P06 TA Hiring

Overview

Pretend you are an instructor of a large class (say CS300), who needs to hire many TAs to hold office hours. Each candidate TA has provided you with a schedule that lists the times throughout the day that they will be available to hold office hours, and the university administration has set a maximum number of TAs that you are allowed to hire. There are two problems to solve: decide **which TAs to hire** to **(1) maximize the total number of hours** throughout the day that **at least one TA** will be available to hold office hours, and **(2) minimize the TA budget** while providing a **guaranteed number of office hours**.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **10/29**
+5%	**Bonus Points**: students whose ***final*** submission to Gradescope has a timestamp earlier than **5:00 PM CDT** on **WED 11/01** will receive an additional 2.5 points toward this assignment, up to a maximum total of 50 points.
12 points	**Immediate Automated Tests**: accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
19 points	**Additional Automated Tests**: these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.
14 points	**Manual Grading Feedback**: TAs or graders will manually review your code, focusing on algorithms, use of programming constructs, and style/readability.

Learning Objectives

After completing this assignment, you should be able to:

- **Formulate** a recursive solution to a problem by describing the base cases, recursive cases, and how to decompose it into smaller subproblems
- **Implement** a recursive solution to a problem by making recursive calls to solve the subproblems, and combining the results
- **Compare** the recursive formulations of similar problems
- **Explain** why recursion can be a useful problem-solving tool
- **Develop** unit tests to verify the correctness of your algorithms

Additional Assignment Requirements and Notes

Keep in mind:

- Pair programming is **ALLOWED** for this assignment. If you choose to work with a partner, you must review the partnership guidelines and declare your partnership before **11:59 PM** on **Sunday 10/29** using this form.

- The ONLY external libraries you may use in your program are:
  ```
  java.util.{ArrayList, Comparator, Random}
  ```
  Use of *any* other packages (outside of java.lang) is NOT permitted.

- You are allowed to define any **local** variables you may need to implement the methods in this specification (inside methods). You are NOT allowed to define any additional instance or static variables or constants beyond those specified in the write-up.

- All methods in the **Hiring**.java class must be **static**. You are allowed to define additional **private static** helper methods.

- Only the **`HiringTester`** and **`HiringDriver`** classes may contain a main method.

- All classes and methods must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.

- Any source code provided in this specification may be included verbatim in your program without attribution.

- **Run your program locally before you submit to Gradescope**. If it doesn't work on your computer, *it will not work on Gradescope*.

Need More Help?

Check out the resources available to CS 300 students here:
https://canvas.wisc.edu/courses/375321/pages/resources

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Academic Conduct Expectations and Advice, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - What do I do about hardware problems?
  - and more!

- Course Style Guide, which addresses such questions as:
  - What should my source code look like?
  - How much should I comment?
  - and more!

1. Getting Started

1. Create a new project in Eclipse, called something like **P06 TA Hiring**.
   a. Ensure this project uses Java 17. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.

2. Download four (4) Java source files from the assignment page on Canvas:
   a. **Candidate.java**
   b. **CandidateList.java**
   c. **HiringDriver.java**
   d. **HiringTestingUtilities.java**

3. Create two (2) Java source files within that project's src folder:
   a. **Hiring.java** (does NOT include a main method)
   b. **HiringTesting.java** (contains a main method)

All methods in this program that you implement will be **static** methods, as this program focuses on procedural programming using recursion.

2. Overview of Problem and Provided Classes

In this program you will be implementing the class `Hiring` with three static methods, as well as a tester class `HiringTester`. The three static methods in the Hiring class **must be implemented using recursion**, either by directly calling itself (possibly more than once), or by utilizing a private static helper method which is recursive. **You may use a looping construct** such as a while-loop or for-loop in these methods, however you must still utilize recursion by setting up base cases which immediately return, and recursive cases which make progress towards a base case via recursive call(s). We will be manually grading these methods to ensure you have done so properly.

Each of the three methods in the `Hiring` class will solve a variant of the hiring problem. In this problem, we imagine that you have a list of candidates TAs to hire, the times that they are available throughout the day, and their pay rates. There are two related problems we would like to solve:

1. To choose a list of candidates which will **maximize the number of hours** that **at least one** candidate is available, where we have a **limited maximum non-negative number of hires**.
2. To choose a list of candidates which will **minimize our budget** while covering some **required minimum non-negative number of hours** that **at least one** candidate is available.

In each of these problems we make the (very unrealistic) assumptions that we only need one TA to handle all requests during a given hour, and that all candidates are equally qualified so we only care about the hours they are available to work and their pay rate.

You can find the JavaDocs for this project here.

2.1 Candidate Class

The first provided file, **Candidate**.java, contains the **Candidate** class. This class represents a candidate TA, including their **unique candidate ID**, **the times they are available to hold office hours**, and their **pay-rate**. Check the class JavaDocs for a more detailed description of the methods this class implements. You are not required to implement any methods in this class, but you may find it helpful to review their implementations and documentation.

2.2 CandidateList Class

The second provided file, **CandidateList**.java, contains the **CandidateList** class. This class represents a list of possible candidates to hire, or a list of candidates that we have decided to hire. It extends the class ArrayList<Candidate>, so you can call all of the methods that the ArrayList class implements, such as add(), remove(), and size(). Additionally CandidateList contains a custom toString() method, a method **coveredHours()** which calculates the total number of hours that at least one candidate in the list is available to hold office hours, and a method **totalCost()** which calculates the total cost to hire all

candidates in the list. Check the class JavaDocs for a more detailed description of these methods including example calculations. You are not required to implement any methods in this class.

2.3 HiringTestingUtilities Class

The third provided file, **HiringTestingUtilities**.java, contains the **HiringTestingUtilities** class, which contains only static methods. These methods are intended to aid in your implementation of HiringTesting.java (more below). Here are some hints:

- **makeCandidateList** – convenient for creating Candidate objects with the given availability and pay rate for tests
- **generateRandomInput** – useful for the "fuzz" tests – see 6.3 HiringTesting
- **compareCandidateLists** – compare lists of candidates and produce nice output if they differ. Useful for checking if your actual and expected output differ.
- **allOptimalSolutions** and **allMinCoverageSolutions** – reference implementations that solve the same problems you will solve recursively (except they do it iteratively). When you test your implementations, you should compare your output against our reference implementations. See 6.3 HiringTesting
- Nearly all of the methods in **HiringTestingUtilities** are overloaded – one overloaded implementation includes a pay rates for testing the minCoverage version of the problem and the other overloaded implementation does not.

See the Javadocs for HiringTestingUtilities for more information. You are not required to implement any methods in this class. You are also not required to read or understand the methods in this class (though you are encouraged to try).

3. Greedy Hiring

3.1 Method Description

In your **Hiring**.java class, create a new method **greedyHiring** with the following signature.

```
public static CandidateList greedyHiring(CandidateList candidates, CandidateList hired, int hiresLeft)
```

In your **HiringTester**.java class, create two test methods for greedyHiring with the following signatures.

```
public static boolean greedyHiringBaseTest()

public static boolean greedyHiringRecursiveTest()
```

The **greedyHiring** method will **recursively** solve the hiring problem by using a **greedy approach**. This means that at each step of solving the problem, we will choose to hire the one candidate which **increases the total number of hours** that our list of hired TAs are available at **by the largest amount,**

assuming that the candidates in the hired parameter are already hired. For instance, to decide who to hire first (i.e. when the hired CandidateList is empty), we simply choose the candidate who is available for the maximum number of hours! This **does not necessarily guarantee that the chosen list of candidates will be completely optimal[1] (i.e. there may be a set of candidates that can cover even more hours)**, but it is a good first step to understand the problem. If there is no solution to the problem you should return a null value. If there is a solution with no hires (e.g. when initially hiresLeft = 0) you should return an empty CandidateList.

The greedyHiringBaseTest method is responsible for testing the greedyHiring method in the case where no recursive calls are made, and the greedyHiringRecursiveTest method is responsible for testing the greedyHiring method in the case where recursive calls will be made. After you read the example in Section 3.2, think of your own examples and implement them as test cases in these tester methods after finding the correct solution by hand. Your test methods should **NOT care about the ordering** of the candidates in the list, as this does not change the hiring decisions. The utility method **HiringTestingUtilities.compareCandidateLists** can be used to compare two CandidateLists without respect to their orders.

The **greedyHiring** method **must be implemented recursively**, either by calling itself, or by calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

See the below visualization of how the greedy algorithm should progress, and for an explicit example of why the chosen solution may not always be the best. **Start coding only after you've worked through this example, created and solved a few examples of your own, and have integrated them into your tester methods.** Consider planning your base cases, recursive cases, and recursive calls by describing or drawing the approach you'll take before starting to code.

3.2 Example

Consider the following scenario where we have three TA candidates, Alice, Bob, and Carol. Suppose we have six 1-hour time slots we would like to cover for office hours (for this assignment, all office hour shifts will start exactly on the hour and will be a whole number of hours long). Alice is available at 4 PM, 7 PM, 8 PM and 9 PM. Bob is available at 4 PM, 5 PM, and 8 PM. Carol is available at 6 PM, 7 PM, and 9 PM. We have not yet selected any candidates (indicated by (0) next to each candidate's name), and so there are no hours at which any selected candidates are available. In total we would like to hire 2 out of the 3 candidates.

[1] Although it **will** be fairly good. In fact, it's guaranteed to return a solution in which the TAs are available for **at least** a (1 - 1/e) ≈ 0.632… proportion of the number of hours they're available for in the best possible solution.

Candidate (Selected)	Availability					
	4 PM	**5 PM**	**6 PM**	**7 PM**	**8 PM**	**9 PM**
Alice (0)	1	0	0	1	1	1
Bob (0)	1	1	0	0	1	0
Carol (0)	0	0	1	1	0	1
TAs Available	0	0	0	0	0	0

In the greedy algorithm, we will always choose the next candidate to **maximize the number of hours covered if we were to hire only that one more candidate**. For instance, as we have not decided to hire anyone yet, hiring Alice would add 4 hours of availability, hiring Bob would add 3 hours of availability, and hiring Carol would add 3 hours of availability. Therefore we first choose to hire Alice, as she maximizes the total availability.

Candidate (Selected)	Availability					
	4 PM	**5 PM**	**6 PM**	**7 PM**	**8 PM**	**9 PM**
Alice (1)	1	0	0	1	1	1
Bob (0)	1	1	0	0	1	0
Carol (0)	0	0	1	1	0	1
TAs Available	1	0	0	1	1	1

With the decision to hire Alice, we now have at least one TA available for the hours of 4 PM, 7 PM, 8 PM, and 9 PM. At this point, if we hired Bob then we would add 1 new hour of availability, at 5 PM, and if we hired Carol then we would add 1 new hour of availability, at 6 PM. As both of these choices result in the **same number of additional available hours, the greedy algorithm can make either choice**. We will choose to hire Bob, the first candidate, in this case.

Candidate (Selected)	Availability					
	4 PM	5 PM	6 PM	7 PM	8 PM	9 PM
Alice (1)	1	0	0	1	1	1
Bob (1)	1	1	0	0	1	0
Carol (0)	0	0	1	1	0	1
TAs Available	2	1	0	1	2	1

As we've hired 2 out of 3 candidates, the greedy algorithm can stop and return the list of chosen candidates, Alice and Bob. Using this approach, we managed to cover a total of 5 out of 6 of the hours **with at least one TA**. We do not care about the total number of TAs covering each hour, only that there's at least one. It's important to note that this is **not the overall best hiring decision, and that's OK** (for now!). This is because if we instead hired Bob and Carol, we could cover all 6 hours with at least one TA.

4. Optimal Hiring

4.1 Method Description

In your **Hiring**.java class, create a new method **optimalHiring** with the following signature.

```
public static CandidateList optimalHiring(CandidateList candidates, CandidateList hired, int hiresLeft)
```

In your HiringTester.java class, create three test methods for optimalHiring with the following signatures.

```
public static boolean optimalHiringBaseTest()

public static boolean optimalHiringRecursiveTest()

public static boolean optimalHiringFuzzTest()
```

The **optimalHiring** method will **recursively** solve the hiring problem of finding the set of candidates which cover the most hours by **exploring all possible sets of candidates to hire, assuming that the candidates in the hired parameter are already hired.** This means that at each level of solving the problem, we must separately explore all possible candidates which are still available to hire. This **method must return a solution that is completely optimal** (i.e. there does not exist a different set of hires that can cover more hours, though there may be alternate solutions that hire the same number of hours), unlike the previous greedy method. If there is no solution to the problem you should return a null value.

If there is a solution with no hires (e.g. when initially hiresLeft = 0) you should return an empty CandidateList.

The optimalHiringBaseTest method is responsible for testing the optimalHiring method in the case where no recursive calls are made, and the optimalHiringRecursiveTest method is responsible for testing the optimalHiring method in the case where recursive calls will be made. After you read the example in Section 4.2, think of your own examples and implement them as test cases in these tester methods after finding the correct solution by hand.

The optimalHiringFuzzTest method tests your implementation using random inputs and compares the output against an iterative reference implementation. See Section 6 for more info.

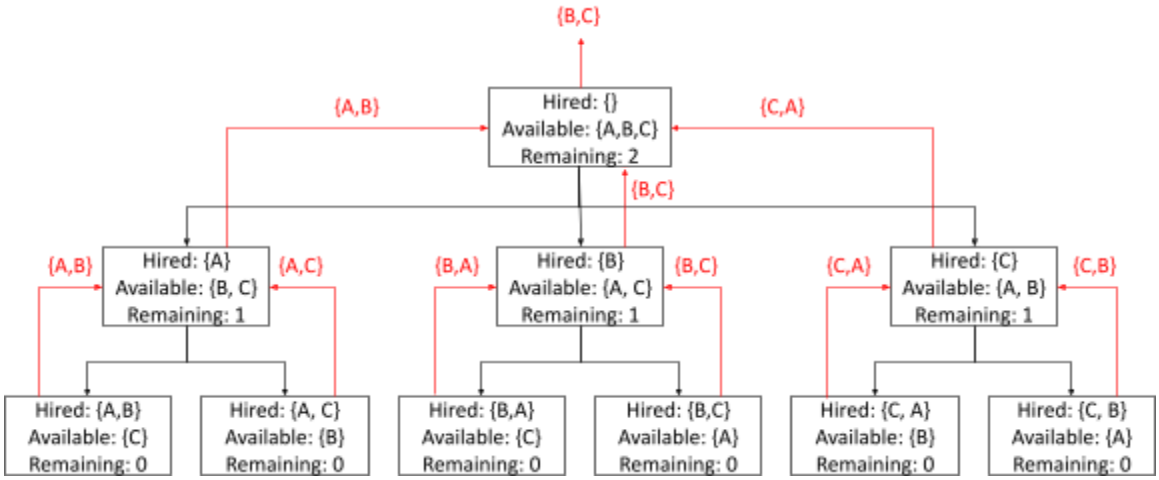
The **optimalHiring** method **must be implemented recursively**, either by calling itself, or calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

See the below visualization of how the call-tree of the algorithm may look like. **Start coding only after you've worked through this example, created and solved a few examples of your own, and have planned your base cases, recursive cases, recursive calls, and how to solve the whole problem using solutions to the subproblems.**

4.2 Example

Consider the same scenario as before with (**A**)lice, (**B**)ob, and (**C**)arol. We will illustrate the process of solving this example optimally using a recursive call tree where each instance of a call to the method is represented by a rectangle in the diagram that shows (1) the list of TAs we have already **hired**, (2) the list of candidates who are still **available** to be hired, and (3) the number of **remaining** candidates that we will need to hire.

Each recursive call is represented by a downward black arrow, pointing from the calling stack frame of the method towards the stack frame that is being called. For instance, the top-most method instance with "Hired: {}, Available: {A,B,C}, Remaining: 2" makes three recursive calls. Each method instance that is not a base case has to **make all possible recursive calls that lead to a valid sub-problem**, and **return the best solution to the overall problem**. The top method instance makes three recursive calls because at that point, either Alice, Bob, or Carol may be hired first. The bottom method instances do not make any recursive calls as they are at a base-case, where we have already hired the maximum number of candidates.
The return-values of the method calls are shown in red text, and there is a red arrow pointing in the opposite direction of the method calls to indicate which method receives the return value from the recursive call. For instance, the top-most method instance first receives the hiring list {A,B} from the left-most recursive call, as this is the **optimal solution to the sub-problem** where we first hire Alice.



Walk through this visualization slowly on your own, ensuring that you understand **why** each recursive call is being made, **what** the subproblem represented by the recursive call is, and **how** the return values from the recursive calls are combined into a solution to the overall problem.

5. Minimum Budget Hiring

OK, now that you've got some recursion under your belt (or whatever the cool kids wear these days), we have a final variant of the problem for you to solve. As before, we have a set of candidates that we want to hire from, and we know the availability of these candidates. However, now we actually have to pay the TAs (darn labor laws!).

In addition to their availability, each candidate also has a pay rate. Different candidates may be paid differently. We will treat this pay rate as a TAs total compensation or salary, so you **should not** multiply the pay rate by the number of hours they are available, and they will get paid the same amount regardless of which hours they get scheduled for.

Suppose we have a minimum number of office hours we want to offer throughout the week. Our goal is to hire **the least expensive set of candidates** that still offers **at least the minimum required number of office hours**. In other words, the number of hours when **at least one** TA is available must be **at least the required number of office hours**. We can hire as many candidates as we need to to achieve this goal.

5.1 Method Description

In your **Hiring**.java class, create a new method **minCoverageHiring** with the following signature.

```
public static CandidateList minCoverageHiring(CandidateList candidates, CandidateList hired,
                                              int minHours)
```

In your HiringTester.java class, create three test methods for minCoverageHiring with the following signatures.

```
public static boolean minCoverageHiringBaseTest()

public static boolean minCoverageHiringRecursiveTest()

public static boolean minCoverageHiringFuzzTest()
```

The **minCoverageHiring** method will **recursively** solve the hiring problem of finding the set of candidates which cover the required number of hours with the least cost by **exploring all possible sets of candidates to hire, assuming that the candidates in the hired parameter are already hired.** This means that at each level of solving the problem, we must separately explore all possible candidates which are still available to hire. This **method must return a solution that is completely optimal** (i.e. there does not exist a less expensive set of hires that can cover the minimum number of hours, though there may be alternate solutions that have the same cost). If there is no solution to the problem (e.g. the minimum required number of hours cannot be met) you should return a null value. If there is a solution with no hires (e.g. when initially hiresLeft = 0) you should return an empty CandidateList.

The minCoverageHiringBaseTest method is responsible for testing the **minCoverageHiring** method in the case where no recursive calls are made, and the minCoverageHiringRecursiveTest method is responsible for testing the minCoverageHiring method in the case where recursive calls will be made. After you read the example in Section 5.2, think of your own examples and implement them as test cases in these tester methods after finding the correct solution by hand.

The minCoverageHiringFuzzTest method tests your implementation using random inputs and compares the output against an iterative reference implementation. See Section 6 for more info.

The **minCoverageHiring** method **must be implemented recursively**, either by calling itself, or calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

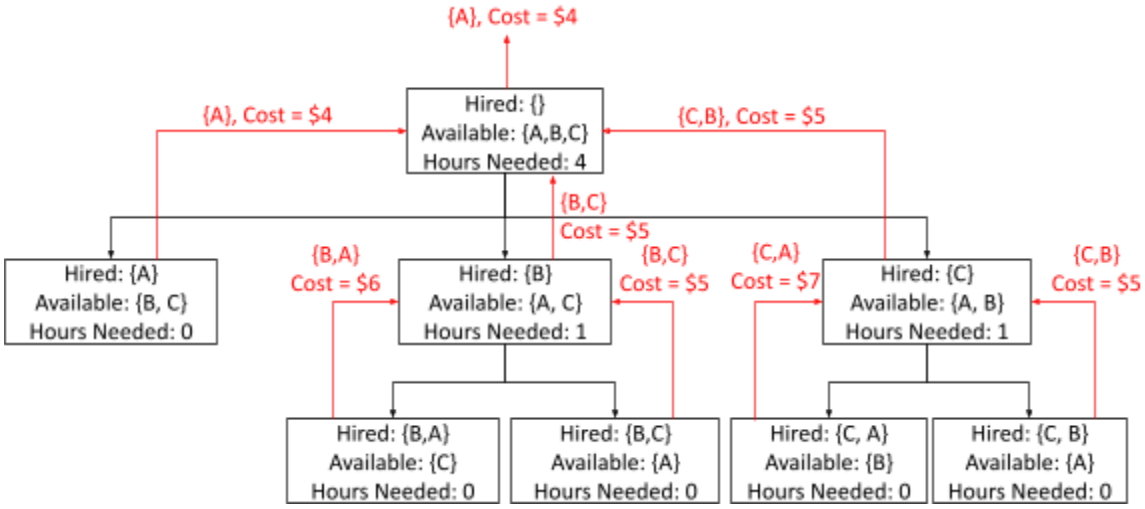
See the below visualization of how the call-tree of the algorithm may look like. **Start coding only after you've worked through this example, created and solved a few examples of your own, and have planned your base cases, recursive cases, recursive calls, and how to solve the whole problem using solutions to the subproblems.**

5.2 Example

Alice, Bob, and Carol are back, except we are adding their pay rates now:

Candidate	Pay Rate	Availability					
		4 PM	5 PM	6 PM	7 PM	8 PM	9 PM
Alice	$4	1	0	0	1	1	1
Bob	$2	1	1	0	0	1	0
Carol	$3	0	0	1	1	0	1

Suppose we wish to find the least expensive set of TAs that can cover at least 4 hours. The following diagram illustrates the recursive calls to find the solution, similarly to Section 4.2.



Note that in this example case, we never needed to explore the possibility of hiring all three candidates because it happens in this example that hiring any two candidates covers at least 4 hours, the minimum we wanted. This will not always be the case. Your implementation should explore all possible combinations of candidates that meet the minimum number of hours, even if that means exploring all possible combinations of candidates.

6. Testing Your Implementations

You will write some tests using the techniques we've been using so far in all of our projects, such as working through an example by hand and then hard-coding the solution in the tester class. However, in many cases, this doesn't exercise all possible paths of our code very thoroughly.

For that reason, in this assignment, you will also use two new testing techniques for testing our implementation. More tools for your programming toolbox!

6.1 Background: Comparing to a Reference Implementation

The first technique is to compare the results of your implementation against a reference implementation. We have provided reference implementations for two of the methods you are writing.

HiringTestingUtilities.allOptimalSolutions() – this method returns an array containing ***all*** optimal solutions that could be returned by your **optimalHiring()** implementation. In other words, if the CandidateList returned by your method is not in the list returned by **allOptimalSolutions()**, then you have a bug in your implementation. By comparing the output of your implementation against the reference implementation, you can check its correctness without hard-coding the solution to your tests.

HiringTestingUtilitiesallMinCoverageSolutions() – serves the same purpose, but for the **minCoverageHiring()** version of the problem.

You might be asking… if we have a reference implementation, then why do we need to implement another implementation? When would we already have an implementation to compare against in the real world?

Actually, this is a surprisingly common situation when you are writing a more efficient version of an existing implementation or trying out a new algorithm to solve a problem or are adding new features to your program but want to make sure you didn't break old functionality. Having a simple, possibly slow, known-correct implementation is a useful way to check your new version.

6.2 Background: Fuzz Testing

The second new testing technique we will introduce is *fuzz testing* or *fuzzing*. Fuzzing means generating a large number of random inputs for your program and checking that the program behaves as expected. Using many random inputs allows you to exercise a bunch of the different code paths in your implementation – ideally, we would exercise every possible path control flow could follow in your program with many different inputs, including edge cases.

Interesting aside: fuzz testing was invented here at UW-Madison (the professor is actually still here at the university) – it is a widely used technique for finding security vulnerabilities.

6.3 HiringTesting

In HiringTesting, you will write simple case tests for greedyHiring(), optimalHiring(), and minCoverageHiring(), as you have been doing for previous assignments. Specifically, you must explicitly test both the base case and the recursive case for your methods.

In addition, you will then write tests in **optimalHiringFuzzTest**() and **minCoverageHiringFuzzTest**() that combine fuzzing and the reference implementation. (NOTE: these will take a few seconds to run rather than finish instantly because we are testing a lot of cases automatically.) In a loop, do the following:

1. Generate a random input to the problem. The **HiringTestingUtilities** class has some methods you may find useful for this. NOTE: if you create your own **Random** instance, **make sure that you set the *seed* value**. This will ensure that every time you run your tester, you will get the same "random" sequence of values, which will make debugging waaaay easier. Your fuzz tests should test **at least 100**, but **not more than 200** randomly generated problem instances (not more than 200 so that the autograder does not time out!). Each randomly generated instance should have (some of these requirements are satisfied by using the provided utility methods, but not all; read the provided code and see what it does!):
    a. a randomly generated number of hours in the schedule in the range **[1, 5]**
    b. a randomly generated number of candidates in the range **[1, 10]**
    c. (optimalHiring) a randomly generated number of desired hires in the range **[1, numCandidates]**
    d. (minCoverageHiring) a randomly generated minimum number of hours to schedule in the range **[1, numHours]**
    e. each candidate should have randomly selected availability (e.g., by flipping a coin for each hour to determine if they are available). (Hint: you may find the different versions of **generateRandomInput** useful here…)
    f. (minCoverageHiring) each candidate should have a randomly generated salary. We recommend keeping the maximum salary relatively small (< number of candidates / 2) to minimize the chance that the optimal solution just picks one candidate. (Hint: again, you may find **generateRandomInput** useful here… note that it is overloaded!)
2. Solve the generated input using your implementation and the appropriate reference implementation in **HiringTestingUtilities**.
3. Compare the output of your implementation against the output of the reference. There are some useful methods for this in **HiringTestingUtilities**.

Assignment Submission

Hooray, you've finished this CS 300 programming assignment!

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, submit your source code through Gradescope.

For full credit, please submit ONLY the following files (source code, *not* .class files):

- Hiring.java
- HiringTesting.java

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Students whose final submission is made before 5pm on the Wednesday before the due date will receive an additional 5% bonus toward this assignment. Submissions made after this time are NOT eligible for this bonus, but you may continue to make submissions until 10:00PM CDT on the due date with no penalty.

Copyright notice