Lab 2: Greedy Algorithms, Part 1.

Due date: Tuesday, April 13, at the beginning of lab period.

Lab Assignment

Assignment Preparation

This is an **individual** lab. The goal of this lab to give you an opportunity to solve two problems using greedy algorithms discussed in class.

The Task

You will implement greedy algorithms for solving two problems discussed in class: making change and activity selection. In addition to implementing the algorithms, you will implement and deploy simple evaluation frameworks for each of the two algorithms.

Task 1: Making Change

You will be solving the version of the problem described in the lecture handout:

Problem. A cashier at a store accepts money in payment for goods and needs to make change. The money comes in a finite number of fixed coin and bank note denominations. Assuming that the cashier has access to unlimited quantities of bank notes and coins of every denomination, the cashier needs to give change *using the smallest number of coins/bank notes* given a specific amount of money.

Implementation Notes. You will implement a Java MoneyChanger class with the following features:

- `int NCoins;` instances of the `MoneyChanger` class will have an `NCoins` field, which will store information about the number of different denominations of coins/bank notes used to make change.
- `int[] Money;` instances of the `MoneyChanger` class will have a `Money` field. This field shall contain the list of all available coin denominations for making change. sorted in **descending** order. For example, for U.S. currency, the array will contain the following values:

  { 10000, 5000, 2000, 1000, 500, 100, 25, 10, 5 ,1}[1]

- `MoneyChanger(int N)`: a constructor creating an empty `MoneyChanger` instance with N different currency denominations.
- `void setCurrencyArray(int[] Coins)`: sets the `Money` component of the `MoneyChanger` class to the `Coins` array passed into this method.
- `int[] makeChange(int amount)` is your implementation of the **greedy** algorithm for making change. It returns an array which stores, for each denomination from the `Money[]` array, the number of coins/notes of this denomination in the change returned.

Note, that while you **must** implement the features above, you are also allowed to implement any other features in your `MoneyChanger` class, that you find convenient.

In addition to the `MoneyChanger` class, you should implement (outside of this class) a printChange() method (figure out the return type and the input parameters that fit your implementation best), which, given an array of change (e.g., returned by the `MoneyChanger.makeChange()` method) and an instance of the `MoneyChanger` class as input, will print the information about the change.

Testing and Evaluation. Create a simple testing framework that allows you to do the following:

- initialize `MoneyChanger` instances to either a random set of currency denominations (randomly select the number of denominations, randomly determine the actual denominations, but remember — they must be stored in descending order).
- initialize `MoneyChanger` instances to represent the US currency system as above.
- Generate random requests for change and run your `makeChange()` algorithms on them.

[1] Assuming no \$2 bills and no half-dollar coins, as their circulation is very limited. Also, assuming no \$500 and \$1000 or larger bills, as they are not in active circulation.

Submit a java file named `TestChange.java` which contains a version of your testing framework, which uses two money changers: one for US currency and one - randomly generated and outputs the change given by each money changer for 10 randomly generated amounts (for a total of 20 outputs. Make certain, you specify in the output the value of change to be given, and which currency/money changer is used. Output the denominations in the random money changer.)

Task 2:Activity selection

The activity selection problem you will work on is formulated as follows:

Problem. You are trying to schedule a list of activities to take place in a conference room. A number of activity requests are presented to you. For each activity, the request contains the start and the end time. Your goal is to produce the activity schedule for the conference room that would accommodate as many activities as possible.

Implementation Notes. You will create a Java class `ActivityScheduler` which will have the following features:

- `int NRequests`: an instance variable which will store the total number of activity requests received by the scheduler.
- `int maxTime`: and instance variable specifying the largest possible timepoint for the schedule. Note, that you will always start scheduling at timepoint 0. All values in the scheduling requests will be between 0 and `maxTime` (inclusively).
- `Request[] Requests`: an array of `NRequests` requests. Each request is represented by an instance of a `Request` object:

```
public class Request {

 int start; // start time of the requested activity
 int finish; // end time of the requested activity

 Request(int a, int b) { start=a; finish=b;}

 public int getStart() { ... }

 pubtlic int getFinish() { ...}

  ....
}
```

(fill in the ... as you see fit)

You **may assume** that the `Request` instances in your `Requests` array are always ordered according to the needs of your scheduleing algorithm.[2].

- `ActivityScheduler(int time)` constructs an instance of the `ActivityScheduler` object with `maxTime` component set to the value of parameter `time`.
- `void setRequests(int N, Request[] R)` sets the `NRequests` and `Requests` components of the `ActivityScheduler` object to N and R respectively.
- `int[] findBestSchedule()` generates an optimal activity schedule using a greedy algorithm. It outputs an array of size `NRequests` of zeroes and ones. A value of 0 in position i means that the request `Requests[i]` is denied, a value of 1 in position i means that the request `Requests[i]` is scheduled.

You should also implement a `printSchedule()` method which takes as input a schedule array and an `ActivityScheduler` instance and prints to screen the list of the scheduled activities.

Evaluation. The testing and evaluation framework you have to create for the activity scheduler should perform the following functions:

- Generate a random instance of a `ActivityScheduler` class. You should select a reasonable range of acceptable `maxTime` values (e.g., from 10 to 100, or so).
- Generate a random activity request. Note that the best way is to generate a start time, and sample event duration from a distribution (uniform, or normal) of possible event durations (feel free to determine the acceptable range of durations: it should not be too large, but neither should it be too small).
- Generate a sequence of random requests which can be passed to a instance of `ActivityScheduler` as the `Requests` component (i.e., make certain, they are ordered as you need them to be).
- Call the scheduling algorithm on an instance of `ActivityScheduler` and print (when necessary) its output.

You will submit a Java program `TestActivity.java` which uses your evaluation framework to study the dependency of the number of scheduled activities on the number of requested activities and on the average duration of an activity.

[2]This is because you will be repsonsible for writing code that generates arrays of `Request` objects, and you should be able to generate them appropriately then.

In particular, you will select three different time ranges (i.e., three different values of `maxTime` for your `ActivityScheduler` objects), and for each time range you will do the following:

- Generate 200 instances of Activity Scheduling problem. Each instance is an array of `Request` objects. The number of objects requests should be generated randomly.
- Run the scheduler algorithm on each instance and record the following information:
  - Number of requests in the instance.
  - Number of requests that were granted (satisfied) by the scheduler.
  - Average length of a request in the instance.
- Output **all** information in an easy-to-read and easy-to-use format (e.g., CSV).

With the data collected, prepare the scatterplots depicting (for each `maxTime` value), the relationship between the *number of requests that were granted* (as dependent variable) and the *total number of requests* (as independent variable) and the relationship between the *number of requests that were granted* (as dependent variable) and the *average duration of a request*.

Prepare a short report containing your graphs and some observations that you can make about the obtained results.

Deliverables

This part of the lab has both electronic and hardcopy deliverables. All electronic deliverables shall be submitted by the assignment deadline using the following `handin` command:

Use `handin` to submit:

```
$ handin dekhtyar-grader lab02-349 <files>
```

The hardcopy deliverables shall be submitted to the instructor during the lab period on Tuesday, April 13.

Please note, that we will start a new lab on April 13, so your work shall be completed by then (you are allowed to submit your materials, both electornic and hardcopy during that lab, but you will need to work on the next assignment during the lab.).

Electronic Deliverables. Submit the following files:

- `MoneyChanger.java`
- `TestChange.java`

- Any other JAva files necessary to run your "make change" submission.
- `ActivityScheduler.java`
- `TestActivity.java`
- Any other Java files necessary to run your "Activity scheduling" submission.
- `README` file. The file should contain your name, and any compilation notes that the grader needs to be aware of.
- `ActivityReport.pdf`: your report about the work you `TestActivity.java`. (the orginal format of the file can be anything you want, but please, convert your report to PDF prior to submission).

Hardcopy Submission. A hardcopy of your `ActivityReport.pdf` file should be submitted to the instructor.

Note that we expect that all your programs will compile from the command line. Make sure you test for that.

Good Luck!