CS211 Spring 2007

Assignment 4 — Asymptotic Complexity, GUIs and The Event Model, Java Threads and Concurrency, Client/Server and Peer-to-Peer Communication

Due Friday, 13 April 2007, 11:59:59pm

1 General Instructions

1.1 Purpose

This assignment will involve written exercises on asymptotic complexity, big-O notation, and heaps, and programming exercises using the Java swing classes, Java event model, GUIs, threads and concurrency, and client/server and peer-to-peer communication. It is a fairly substantial assignment involving a fair amount of code, but we supply most of the code. Even though the final solutions do not involve much code, it will still be a challenge to find it. Get started early!

1.2 Grading Criteria

Same as before.

1.3 Partners

As before, you may work in pairs. Please follow the instructions on the website for selecting a partner. You must create a group in CMS for yourself and your partner.

2 Asymptotic Complexity and Big-O Notation

Purpose: Understanding asymptotic complexity and using big-O notation
Estimated time: 1 hour
Points: 10

1. Arrange the following functions on n in order of increasing asymptotic growth rate. If there are any ties, indicate them.

   $n^2 \log n$, $n^{1/3}$, 10181986, $2^{n/4}$, n^3, $2^{50} + n$, n^{2n}, $(\log n)^{1/2}$, $n!$, $\log_2 n$, $\log_3 n$

2. Say whether the following statements hold. In positive cases, provide a witness pair, and in negative cases, argue that no witness pair exists.

   (a) n^2 is $O(n^{1/2})$

   (b) $\log(n)$ is $O(n)$

   (c) $(\log n) \cdot (\log n)$ is $O(2 \log n)$

   (d) $n^{\log n}$ is $O(n^5)$

3. Determine the runtime for the following pieces of code as a function of n.

   (a)
   ```
   int sum = 0;
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n/4; j++)
         sum++;
   ```

(b)
```
int sum = 0;
int k = n;
while (k > 0) {
  k = k/2;
  sum++;
}
```

(c)
```
int sum = 0;
for (int i = 0; i < n; i++) sum++;
for (int j = 0; j < n; j = j + 2) sum++;
```

(d)
```
void weirdfunction1(int n) {
   if (n <= 1) System.out.println("Finished thinking!");
   else {
      for (int i = 1; i <= n; i++) System.out.println("thinking");
      weirdfunction1(n - 1);
   }
}
```

(e)
```
void weirdfunction2(int n) {
   if (n <= 1) System.out.println("Done!");
   else {
      weirdfunction2(n/2);
      for (int i = 1; i <= n; i++) {
         for (int j = 1; j <= n; j++) {
            System.out.println("Please wait");
         }
      }
   weirdfunction2(n/2);
   }
}
```

Submit your solutions in a file `Asymptotic.txt`.

3 Heaps

Purpose: Understanding heaps
Estimated time: 1 hour
Points: 10

1. Suppose we have an array-based heap containing the integers 3, 17, 14, 19, 28, 26 in locations 0,1, . . . ,5, respectively. Draw the array after performing *each* of the following operations.
   (a) add 23
   (b) extract min
   (c) extract min
   (d) add 15
   (e) add 0
2. Describe how to use an array-based heap to merge k sorted lists containing a total of n elements in $O(n \log k)$ time.

Submit your solutions in a file `Heap.txt`.

4 File Sharing

Purpose: Java swing classes, the Java event model, GUIs, threads and concurrency, client/server and peer-to-peer communication
Estimated time: 25 hours
Points: 75

4.1 Overview

This problem will introduce you to the Java swing classes for creating GUIs, the Java event model, concurrency and Java threads, and network connections and message passing between hosts.

The application you will work with is a simple peer-to-peer (p2p) file sharing system very much like Napster. It consists of a *server* and several *clients*. The server listens for and responds to requests from clients coming in over a network. Clients would normally be running on different machines attached to the network. They can connect to the server and communicate with it by passing messages. These connections are called client/server connections. Clients can also connect to each other for the purposes of sharing files. These connections are called peer-to-peer (p2p) connections.

For ease of debugging, you can run the server and as many clients as you wish all on the same machine. If you like, you can also run them on different machines if they are connected to the same local network and you know their IP addresses. The IP address consists of four unsigned bytes separated by periods, such as 192.168.1.101. In Windows you can find out the IP address of your machine by typing `ipconfig` at the command line. In Java, you can find out by calling `java.net.InetAddress.getLocalHost()`. If you are only connecting between clients and servers on the same machine, you can use the IP address 127.1.1.0.

The file-sharing system works as follows. The server continuously listens for connection requests from clients. After a client connects to the server, it uploads to the server a list of the names of all files in its current directory on its local machine. These are the files that the client wishes to share with the rest of the world. The server maintains a database of all such file names of all clients currently connected to it. So when the client connects to the server, its file list is integrated into the master file list on the server. The client can then download this master list to view what files are available from other clients. The files themselves are *not* uploaded to the server.

Now say a client C1 wishes to download a file from another client C2. To do this, C1 establishes a p2p connection to C2 and requests the file, which C2 then sends to C1 over the same p2p connection.

To see how this works, download the archive `A4Release.zip` and unpack it. This will create a directory `A4Release` with a subdirectory `a4` containing several `.java` files. Put a bunch of text files in the `A4Release` directory. Navigate to the `a4` subdirectory in a command window, then type `javac *.java` to compile. After you have compiled, your class files should be in the directory `a4`. Go up to the parent directory `A4Release` and type the following four lines:

```
javaw a4/P2P server 100 200
javaw a4/P2P client 5001 700 200
javaw a4/P2P client 5002 700 450
javaw a4/P2P client 5003 700 700
```

This will start up the server and three clients. The arguments 100 200, 700 200, 700 450, and 700 700 specify the positions of the windows on the screen. The numbers 5001, 5002, 5003 specify the ports on which the three clients are listening for p2p connections. These numbers must be different and must be different from 5000 if you are running on the same machine. The server always listens on port 5000 for connections from clients (although you can change this).

Click `start` on the server to start it. After you have done this, click `connect` on the three clients. They should connect to the server. If all goes well, click `refresh` on the server. This will list all the files of all the clients' current file-sharing directories (which right now is `A4Release` for all three of them). You will not see any files if you haven't put any other files in that directory. Now click `refresh` on some client. It will send a message to the server, causing the server to refresh. The server will wait a few seconds for the file lists to come in from its clients. After a few seconds, it will send the entire filelist to the client that clicked `refresh`. That client will then display all the files except its own.

Now the client has a list of available files from other clients, and it can download one. Select a row and click `download`. The client will open a p2p connection with the peer specified in the selected row. That peer will then send the requested file to the requester. The file should appear (with a modified file name) in the `A4Release` directory.

4.2 Structure of the Application

Look at the code to see how the GUI is constructed, how the message passing mechanism works, and how connections are created. You will have to understand this code fairly well in order to do this assignment.

Both the `Client` and `Server` classes are subclasses of the abstract class `Host`. The `Host` class contains code that is common to both client and server. In particular, both clients and servers need to listen on a port for incoming connections from clients.

The `GUI` class constructs the GUI for both the client and server. Most of this code is common except for a few differences.

Anonymous subclasses of `AbstractAction`s are used to define actions shared by buttons and menu items. For example, the `Connect...` action appears both as a toolbar button and in the Action menu of the client.

4.3 Events and Listeners

Both `Client` and `Server` implement `MessageListener`, which is an explicit example of how the Java event model works. Many other event-handling mechanisms in Java, such as button and menu handlers, work the same way. Typically there are a collection of *listeners* associated with various GUI objects, which can handle events of various types associated with those objects, such as a mouse click on the object or when an item is selected from a menu. To be a listener for a certain type of event, an object usually has to implement an interface associated with that type of event. The interface requires the listener to contain a method to do something useful when that event occurs. A listener becomes a listener by registering itself to receive those events. For example, an object registers itself to receive message events by calling `addMessageListener(this)`. Whenever such an event occurs (such as a mouse click), the system runs the event handler of each registered listener in a separate thread, passing it an event object that describes the event. In this code, the `MessageListener` is explicit, so you can see how it is put together. A message event is dispatched whenever a message is received.

4.4 Threads

A *thread* is a separate process that can perform a task independently and concurrently from other threads. Most programs have only one thread. However, GUI programs typically have another thread to dispatch events. In addition, the programmer can create new threads to execute various independent tasks. A Java `Thread`, or more generally anything that implements `Runnable`, must have a `run` method. It is possible to call the `run` method directly, in which case the thread that did the call will be the one to execute the `run` method. However, it is more common to call `start`, which will spawn a new separate thread to run the `run` method. The `start` method returns immediately, so that the thread that called it can go off and do something else. Threads normally terminate by returning from `run`.

When there are multiple threads running around, one must be careful to avoid concurrent modification of data structures. This can be avoided by *synchronizing* on an object. When a thread wants to modify an object `obj`, and there is a chance that another thread might want to access it concurrently, then the modification code can be surrounded by a synchronized statement `synchronized (obj) {...}`, which will prevent other threads from accessing `obj` until control leaves the synchronization block.

4.5 What You Will Do

You will make four modifications to this code to add extra functionality:

1. Change the representation of the file list data to maintain it in a `JTable` that allows the rows to be sorted by file name, host name, ip address, or port number, simply by clicking on the column header.

2. Add a file selector to the `Action` menu of clients to allow the client to select an alternate file-sharing directory rather than the default current directory.
3. Add a text box and `Search` button in the tool bar to allow the user to search for a row containing the text in the text box.
4. Add a *ping* facility to allow the server to check for disconnected clients and delete them from its database.

Each of these tasks is described in some detail below. However, you will have to do a lot of digging through the Java API documentation and the code that is already there to understand completely how to do what you have to do. To see what we would like the finished product to look like, run the file `a4sol.jar` containing our solution code. Navigate to the directory `A4Release` and type

```
javaw -jar a4sol.jar server 100 200
javaw -jar a4sol.jar client 5001 700 200
javaw -jar a4sol.jar client 5002 700 450
javaw -jar a4sol.jar client 5003 700 700
```

4.6 Constructing a JTable

Right now the data in a file list is maintained in a `Vector<String>` and displayed as a `JList`. These are declared in the `GUI` class. Each row represents one file. The row contains the name of the host where the file can be found, the ip address of the host, the port number that the host is using to listen for p2p connections, and the name of the file, all as one long `String`.

You should replace the `JList` with a `JTable` with four columns, with the data backed by a `TableModel`. Don't worry about sorting just yet, we will get to that later. There is a lot of information on `JTable`s in the Java API documentation, including a tutorial on their use with `TableModel`s.

The `TableModel` should maintain the data for each row of the table and provide methods for accessing its elements. You should declare a new class `MyTableModel` as a subclass of the abstract class `AbstractTableModel`. The `MyTableModel` object must represent the list of rows somehow. You may choose any convenient representation you like. In our solution code, we used a `Vector<RemoteFile>`. The `MyTableModel` class will also have to provide methods for accessing the data:

```
public String getColumnName(int col)
public int getRowCount()
public int getColumnCount()
public Object getValueAt(int row, int col)
public Class<?> getColumnClass(int col)
```

plus any other convenience methods. The column names should be as in our solution. The `getColumnClass(int col)` method should return `Integer.class` for the port and `String.class` for the other columns. This will be essential for correct sorting. The exact implementation of the other methods will depend on the data representation you choose.

Get this working before you go on to sorting.

4.6.1 Sorting

For sorting rows, you do not have to implement the sorting yourself. There is a file `TableSorter.java` available from the Java `JTable` tutorial that will do most of the work for you. You can put a `TableSorter` object in between the `TableModel` and the `JTable`, as described in the tutorial. It is only a few lines. After you set this up, you will be able to click on a column heading and it will sort by the data in that column.

The `TableSorter` does not actually physically sort the elements. It only creates a permutation (one-to-one correspondence) between the row indices as you see on the display and the row indices in the underlying data representation. There is a method `int modelIndex(int viewIndex)` in `TableSorter` that goes from a display index to the corresponding internal index. When you select an element in the table, to retrieve the actual element, you will need to go from the display index to the internal index. You should never need to go in the other direction.

4.7 Selecting a Directory

Right now a client looks for files to share in the current working directory. This directory can be obtained by a call to `System.getProperty("user.dir")`. Add a `JFileChooser` to select an alternate directory. This should be activated by a new menu item in the Action menu. Set the file selection mode to `JFileChooser.DIRECTORIES_ONLY` so you can only select directories. Once the directory is chosen, that will become the working directory. All shared files should subsequently come from that directory, and all downloaded files should subsequently go to that directory. There is a tutorial in the Java API documentation on the use of `JFileChooser`.

4.8 Searching for a File

Add a `JTextField` and `Search` button in the client's tool bar to allow the user to search for a row containing the text in the text box. When the user clicks the button, you need to retrieve the text from the `JTextField` and search the file list data for the first occurrence of the text as a substring. If not found, a message should be displayed in the message area. If found, the row should be automatically selected as if the user had clicked on it. This can be done by calling `JTable.changeSelection(...)`.

4.9 Discovering Disconnected Clients

Right now, when a client connects to the server, as far as the server is concerned, that connection exists forever. The connection is entered into a `HashSet` (namely `Host.clients`) and it is never deleted, even if the client shuts down. (This is not an issue for p2p connections: the connection is deleted as soon as the file transfer is done.) One would like to give the server a way to discover when a client has disconnected and delete the connection to avoid the proliferation of dead connections over time.

To do this, the server can periodically "ping" the clients and await a response. If there is no response within a reasonable time, the server will assume that the client has disconnected and will delete the connection.

Create a new `Command`, `Command.PING`. Every 30 seconds, the server will broadcast a `Command.PING` to all the clients. This can be done with a `Timer` and `TimerTask`, which is a way to start up a thread to do something either on a one-time basis or periodically.

Look at the inner class `Server.DelayedFileListMessage` for how to do this. This class uses a one-time-only timer to wait for 3 seconds after requesting file lists from all the clients before sending out the master file list. You should create an inner class `Server.Ping` that pings the clients every 30 seconds. The code will be very similar to `Server.DelayedFileListMessage`, except that it schedules a recurring periodic event instead of a one-time-only event, the delay is 30 seconds instead of 3 seconds, and of course the action will be different. Also, in that same code, you should delete all the clients that have not responded since the previous broadcast 30 seconds ago. You will need to keep track of the responses from the clients in some data structure. After deleting the unresponsive clients, the data structure should be reinitialized for the next time.

On the client side, add a clause for `Command.PING` to the message handler `messageReceived(...)`. The client should simply return a `Command.PING` message back to the server on the same channel.

Back on the server side, add a clause for `Command.PING` to the message handler `messageReceived(...)` that marks the client from which the `PING` was received as still connected, so that it will not be deleted.

4.10 Caveats

We have experienced difficulties attempting to run multiple communicating clients and servers simultaneously on the same virtual machine. That is the reason for the four different commands in a command window, which run four copies of the Java virtual machine in different processes. In particular, it is best not to run them in Eclipse, because these are all running on the same virtual machine. You can edit and compile in Eclipse, but if you have multiple clients on one machine, we recommend against trying to run in Eclipse. For Windows users, we have provided `.cmd` scripts with these commands for your convenience.

It's a little harder to debug if you cannot do a `System.out.println(...)` to see what is going on at a particular point in the program. You can get around this by writing to the message area or putting up an error dialog box just for debugging purposes. The `GUI.error(...)` method does this for you.

Due to the encoding we use for file transmission over a network, our file-sharing system only works with text files. This is intentional. Please do not use this code to violate intellectual property rights.

5 Submitting Your Work

Points: 5

Submit the following files in CMS:

- Exercise 2: `Asymptotic.txt`
- Exercise 3: `Heap.txt`
- Exercise 4: `Client.java`, `Command.java`, `GUI.java`, `Message.java`, `MyTableModel.java`, `Server.java`

Do not submit any other files.