COSC 223: PROBABILITY AND COMPUTING
PROJECT 1: CACHING

Due Friday, March 8, 5:00pm

1 Find a Partner: Due Sunday, February 24, 11:59pm

This is a partner project: you will work with one other student in the class, and you may choose your parter. Your first task is to find a partner and let me know with whom you will be working. By **Sunday, February 24, 11:59pm** you and your partner must submit a single text file (one per team) containing both of your names. For example, if I decided to work with Prof. Alfeld on this project, he and I would submit one text file containing:

```
Kristy Gardner
Scott Alfeld
```

You can submit this through the CS department submission site: `https://www.cs.amherst.edu/submit`, or, from `remus/romulus`, using the `cssumbit` command:

```
cssubmit myteam.txt
```

And then use the menu to navigate to this class and assignment.

If you haven't found someone to work with by Sunday night, submit a text file with just your name and I will find a partner for you.

2 Intellectual Responsibility

You will be working with a partner on this project and all aspects of the project can (and should) be done together. Do not share code or written work with anyone besides your partner, and do not look on the internet for any solutions. You may discuss the concepts involved in this project with other students in the class who are not your partner; if you do, please note at the top of your writeup with whom you consulted, and what you discussed.

3 Submission

There are three different components to submit for this project (see Section 4 for details):

1. All code that you wrote for this project
2. A `README` file explaining how your code works and how I can run it
3. A "research paper" writeup describing your experiments and results

You'll submit all of these files in a SINGLE submission by uploading them via the CS department submission web site: `https://www.cs.amherst.edu/submit`. This assignment is due Friday, March 8 by 5:00pm.

4 Your Tasks

In this project, you will study how the cache replacement policy affects the hit rate (and consequently, mean access time). You will consider four cache replacement policies, all of which we've discussed in class:

- Random: each item is equally likely to be removed from the cache
- First-In First-Out (FIFO): the item that has been in the cache the longest is replaced
- Least Recently Used (LRU): the item that has been accessed least recently is replaced
- Least Frequently Used (LFU): the item that has been accessed the fewest times is replaced

In the course of this assignment, you will:

- Measure the relative performance of different caching policies
- Develop insights and intuitions about why the policies perform the way they do
- Practice writing simulation code
- Identify and understand the implications of modeling assumptions
- Practice writing a "research paper" describing your experiments and findings

4.1 Implementation and Experiments

Your goal is to write a caching simulator that will allow you to measure the hit rate under different cache replacement policies. When we're simulating a system, we typically aren't interested in implementing every single detail of how the system actually works in practice. Instead, we want to keep track of just enough information to measure our quantity of interest—in this case, the hit rate. For example, imagine our RAM has 10 addresses and our cache can store 4 pieces of data. Our simulator needs to keep track of which addresses' data are stored in the cache (is it the data from addresses 2,5,6, and 9, or the data from addresses 1,2,4, and 7?) but we don't actually need to know *what* data was stored at address 2 (our hit rate will be the same if that data value was 01101011 as it would be if the data value was 10011100). We also (depending on the cache replacement policy) might need to keep track of things like when each address in the cache was last requested, or how many times each address has been requested.

You aren't required to use any particular programming language for your simulator, with the caveat that the experiment in Section 4.3 requires you to generate a sequence of requests using some Java code that I've provided. If you choose to use a language other than Java, you'll need to do a little extra work to integrate my Java code with your code.

4.2 Some Experiments

The purpose of the first set of experiments is to answer the following research question:

How does the cache replacement policy affect the hit rate, for different request distributions and for different cache sizes?

We'll assume that our RAM has 1000 addresses, i.e., there are 1000 different pieces of data that could be requested. Let D denote the data item that is requested, where D takes on values between 1 and 1000. We'll consider two different distributions for D:

- $D \sim$ **Uniform**, in which each data item is equally likely to be requested. That is, $p_X(i) = \frac{1}{1000}$ for $i = 1, \ldots, 1000$.
- $D \sim$ **Zipf**(α), in which each data item's popularity is inversely proportional to its index. That is, $p_D(i) = \frac{(1/i)^\alpha}{\sum_{j=1}^{1000}(1/j)^\alpha}$. We will specifically consider the case $\alpha = 1$, so $p_D(i) = \frac{1/i}{\sum_{j=1}^{1000} 1/j}$. This says that the most popular item is requested about twice as often as the second most popular item.

We will assume that each request is drawn independently from all previous requests and independently from the cache contents.

Here are some things you'll need to do for your simulator and experiments:

- Write some code to generate a sequence of requests. You'll need two separate cases, one for the Uniform request distribution and one for the Zipf request distribution.
- Implement the four cache replacement policies: Random, FIFO, LRU, LFU.
- Write some code to run experiments using your cache simulator. For each experiment, you'll want to run a long sequence of requests (on the order of 10^5 or 10^6) and throw out the first 10^4 or so of them (this is so you ignore any transient effects from your cache starting empty, and instead only measure "steady state" performance). For each of the two request distributions, run experiments for all four policies for different cache sizes ranging from 10 to 200.

4.3 Another Experiment

So far we've been operating under the assumption that each request is drawn independently, i.e., the next data item to be requested has nothing to do with the previous data item that was requested. In practice, this assumption often does not hold. Often request sequences exhibit *temporal locality*, meaning that if some data item is requested, it is more likely to be requested again in the near future. This kind of pattern can come up, for example, in code that contains lots of loops. The same data will be needed over and over again on each iteration of the loop. But after the loop has completed, we might not need to read that data again for a very long time (until the loop executes again, for example).

The purpose of this experiment is to answer the research question:

What is the impact of temporal locality on the relative performance of different cache replacement policies?

I've provided some code to generate a sequence of requests, where now each request is *not* independent of the previous request. Download my `RequestGenerator` using the following command:

```
$ wget -nv -i http://tinyurl.com/reqgen
```

Or download directly from: `https://kgardner.people.amherst.edu/courses/s19/cosc223/projects/project1/RequestGenerator.class`.

You should now have a file called `RequestGenerator.class`, which contains (compiled) Java code to generate a sequence of dependent requests. You can use this code by declaring a new instance of the class:

```
RequestGenerator rg = new RequestGenerator();
```

and then calling the `generateRequest()` method repeatedly to generate a sequence of requests:

```
rg.generateRequest();
```

Note: do NOT create a new instance of `RequestGenerator` in between calls to `generateRequest()`. The line `RequestGenerator rg = new RequestGenerator();` should only appear ONCE in your code.

Now run the same experiment as in Section 3.2, this time using my request generator instead of the Uniform or Zipf request distributions.

5 Deliverables

You'll submit three things for this assignment:

1. All of the code that you've written for this project
2. A `README` file explaining how your code works and how I can run it. The main purpose of this document is to enable me to use your code. You should explain what command to type to run your code (and what, if any, command line parameters are needed) and explain the input, output, and purpose of each method. You do not need to describe line-by-line what your code is doing.
3. A writeup of your experiments, results, and what you've learned about caching from your results (see Section 4.1)

5.1 The Writeup

The goal of your writeup is to provide a complete description of what you did in your project, why you did it, and what you found. This will be structured similarly to a research paper; after all, your project was a mini research study.

Your writeup should include the following sections:

1. **Introduction.** Explain the problem that you are studying, why it's important, and the goal of your project.

2. **Experimental setup.** Describe the experiments that you ran. How many data items were there, and how big was the cache? What distributions and parameter settings did you use? How long was your sequence of requests in each experiment? What cache replacement policies did you test? The purpose of this section is to give your readers enough information to enable them to replicate the exact same experiments you ran. In this project, I told you in Section 3 how to set up your experiments. But if you had designed the experiment yourself, you would need to explain to your audience exactly what you did. We're including this section in the writeup for this project to give you practice describing your experiments, even though I already know what you did.

3. **Results and Discussion.** What happened in your experiments? Include graphs showing the data you collected in your experiments, and explain what patterns you see in the graphs. What insights into system performance do your results reveal? This section is where you provide answers to your research questions and provide some explanation of why the results look the way they do. We'll do some analysis of these policies in class—do your experimental results match up well with what the analysis predicted would happen, or were you surprised by what happened in practice? If the analysis and the experiments differed, why might that have occurred? What kinds of request patterns do you think were in my temporal locality request generator that led to the results you saw? Was anything particularly surprising or particularly interesting about your results?

4. **Conclusion.** In this section you'll summarize the main findings of your experiments and discuss any limitations of your experiments or remaining questions that you might want to study. Were there assumptions you made in your system model that might not hold in practice? What do you think would change about the results if you repeated your experiments without these assumptions, or with different assumptions? What questions do you still have about the system you're studying?