CS9053 Section I2

Assignment 8

Prof. Dean Christakos

March 24th, 2025

Due: April 4th, 2025 11:59PM

**Part I: Threading Count of Divisors**

In the class CountDivisors, there are two static methods, countDivisors(long val) and maxDivisors(long from, long to). countDivisors takes a value and returns the number of divisors the value has. maxDivisors takes a minimum and a maximum value and returns the largest value with the largest number of divisors.

Try it out by executing CountDivisors.maxDivisors(0,1000). Then try CountDivisors.maxDivisors(0,100\_000) and note that it takes longer.

If you try CountDivisors.maxDivisors(100\_000, 200\_000) it takes quite a while.

This can run faster if you divide this process pieces.

In the class ThreadedCountDivisors.java, write in the run() method something which will get the value with the largest number of divisors and that number of divisors– implement the class however you choose – and in the main method, generate the threads, then find the results of each thread that has the value you are looking for. (ie, the number that has the largest number of divisors).

Compare the speed of the multithreaded division of labor with simply running maxDivisors(100\_000, 200\_000). See if the interval affects the result.

There are two ways of doing this. One will get full credit, but the faster method will give you 1 point extra credit.

**Note: You should return the largest number with the largest number of divisors. If two numbers have the same number of divisors, pick the larger. The output should be of the format:**

**<Number with largest number of divisors>=<number of divisors>**

**Eg: 10=2**

**Part II: Thread-Safe Least Recently Used Cache**

An LRU (Least Recently Used) cache stores a limited number of item and evicts the least recently accessed item when full. The API is:

get(key): retrieves an item and mark it as recently used

put(key, value): insert/update an item and mark it as recently used. If capacity is exceeded, evict the least recently used item

What you will do is implement the cache with a HashMap to access data by key and a LinkedList to keep track of usage order (most recently used at the front, least at the back). Every time you access an item in the cache, it promotes the item accessed to the front of the list. When you add an item to the cache, if it is at capacity, it removes the item at the back of the list and adds the new on to the front.

The way this will work is that you have to implement your OWN Doubly Linked List. The HashMap will be indexed by a key, K, and return a reference variable to a Node containing the K, V pair. The Node is part of the Doubly Linked List, which means you should be able to remove the node from its location in the Doubly Linked List and place it at the head of the list.

A diagram of a network

AI-generated content may be incorrect.

This diagram should explain the architecture and ensures that get() and put() methods still take only O(1) because the HashMap gives direct access to the Node in the Doubly Linked List which can then be moved to the front in O(1) while Nodes at the end of the Doubly Linked List being dropped for capacity reasons can be removed in O(1) time.

In the code, you can see that I have created a skeleton structure of a Node, Doubly Linked List, and LRU Cache. I have also written some code where several threads add and get items from the LRUCache.

Finally, I have an CacheMonitor thread that checks to make sure that the Cache never exceeds capacity. **There are definitely many other race conditions you have to account for other than just the capacity check.**

Your job is to implement the LRUCache correctly, including the DoublyLinkedList and make this cache threadsafe. We should have several threads accessing the cache and it should never exceed capacity, and items inserted in the cache should be able to remain in the cache until they age out.

Explain your work.