CSE 214 Homework 2

Due October 11th 11:59 pm

Stacks, Queues, and Recursion

10 points

Implement your own Stack and Queue classes, using java.util.LinkedList. Stack should have push, pop, peek, isEmpty, and length. Queue should have enqueue, dequeue, peek, isEmpty, and length. Each of these must run in O(1).

10 points

Write a method that takes in a string and uses your Stack class to determine if it is a palindrome (the same forwards and backwards) in O(n) time.
Example:

```
"racecar" -> True
"not a palindrome" -> False
```

15 points

Make a subclass of Stack called MinStack that adds a getMin method which returns the minimum value in the stack in O(1) time while still being able to do all the other stack operations in O(1).
Example:

```
Push(3)
Stack=3 min=3
Push(5)
Stack=35 min=3
Push(4)
354 min=3
Push(2)
3542 min=2
Pop()
354 min=3
```

Note that it is not enough to keep a single value for the current minimum because we need to be able to change it when the minimum is popped.

20 points

Build a simple simulation of process scheduling on a multi-core processor. We will do round robin scheduling by creating a queue and adding to this queue several processes that need to be run. Each process has a size associated with it which indicates the amount of CPU time it needs to complete. A while loop will simulate each time step on the CPU. At each iteration, each core of our virtual CPU will either grab a process off of the queue, or work on a process it is already holding. If a process is finished (work done so far = process size) then print the job along with the overal timestep at which it finished, then the core that finished it will not requeue it and will grab a new process in the next timestep. If a core is working on the same job for X time steps in a row, then in the next time step it must stop and instead add the process at the end of the queue. Then in the next timestep it can grab another process to work on.

The main method of CpuSimulator reads in the simulation parameters from the command line. See the code skeleton for more details. Here is an example input/output.

```
3
2
3
A,3
B,5
C,6
```

When this runs it will output as it finishes the tasks. It should end up like this.

```
0, A, 4
1, B, 8
1, C, 12
```

This last line tells us the total time to completion.

5 points
Add an `int avgTurnaroundTime()` method to CpuSimulator.
The turnaround time is how long it took to finish a process. We printed this value out when we finished it. Save these values with the CpuSimulator object so that afterwards we can call `avgTurnaroundTime()` to get the average for the run of the simulation.

5 points
Add a `double getUtilization()` method to CpuSimulator.
We will say that a step on a core which actually does some work on a process is a productive step. A step on an idle core (no processes to work on) or one which is busy swapping with the queue is a wasted step. Since the cores are running in parallel the total number of steps, or the cpu time is $cpuTime = numCores \times realTime$ where $realTime$ is the step counter we printed when we finished the last process. The number of productive steps plus the number of wasted steps should equal the total cpu time. `getUtilization()` should return the number of productive steps divided by the total cpu time for the previous run of the simulator. This tells us how efficiently we are using the CPU resources. If we never cut off work on a process to sechedule another one, then our utilization would be nearly 1.0. However, the tradeoff is that some greedy, long-running processes could hog the cores and increase the turnaround time for everything that comes after. With round-robin scheduling, every process gets a fair share of the CPU.

10 points
Levenshtein distance is one way of comparing the similarity of two strings. It is also known as edit or spell check distance because it can help identify typos as strings very similar to dictionary words. It is defined as the number of character insertions/deletions/substitutions needed to transform one string into another. We can also define this recursively in this way...
For string x with prefix of length i and string y with prefix of length j,
$D(0, j) = j$
$D(i, 0) = i$

$$D(i,j) = min\begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+\delta(x[i-1],y[j-1]) \end{cases}$$

$$\delta(a,b) = \begin{cases} 0 \text{ if } a = b \\ 1 \text{ otherwise} \end{cases}$$

Translate this recursive definition to a recursive method in java that returns the edit distance between two strings a and b.
If you decide to add parameters in your implementation, put them in a helper method so that the tests can just call `editDistance(String a, String b)`.
Example:

```
abcdefg
aqcefgj
```

$\rightarrow 3$

10 points
Similar to fibonocci which we looked at in class, the edit distance recursive algorithm does a lot of repeated work which makes it very slow. Write a second implementation `fastEditDistance(String a, String b)` (same note as above about potential helper methods) which uses dynamic programming to reduce redundant recursive calls. Hint: build a 2 dimensional table to store values in.

15 points
Coding style with possible 5% extra credit. Same as homework 1.