CSC 495 – Problem Set 6 – Due Tuesday, 21

Problem 6.R1: The Value of Cache

Required Problem
Points: 50 points
5 extra ninja points if you email me the results from running this test on at least 3 different machines — you need to include information on the machines, such as CPU and cache size (if you know it)

Background

As discussed in Column 7 of *Programming Pearls*, a common first step when doing a "back of the envelope" analysis is to take some quick benchmarks of basic operations that are key to the analysis (pinging machines, sending an email, etc.). Two performance measures that are mentioned in various places in the book deal with how "cache-friendly" algorithms are, and the use of single-precision versus double-precision floating point computation. Problem 4 in Column 2 addresses cache performance (particularly in the discussion of the solution in the back of the book), showing an algorithm that makes fewer operations and yet runs slower than an alternative that uses more operations, due to cache behavior. In Column 6, the "Code Tuning" for the n-body simulation included replacing double precision floating point variables with single precision variables, which he reported doubled the performance of the algorithm. What are these variables like on modern machines?

The Problem to Solve

This problem is pretty different from previous ones in this class, since there is *no input*! You are to write code that measures cache effects that vary when accessing memory sequentially versus non-sequentially, and to measure the relative performance of using `float` variables versus `double`. Your program can output anything you'd like to standard error (see why below), but output to standard output should be two lines, the first containing the ratio of time required for non-sequential access to that for sequential access, and the second containing the ratio of the time required for double precision operations to that for single precision operations. The submission server will simply check to make sure your program outputs two values that are in a sensible range of possible values, but it will also enforce a one minute run-time limit on your program.

Hints and Techniques

Your code needs to be able to time itself to solve this problem. In C and C++, you can use the `getrusage()` function, and in Java you can use `java.lang.management.ThreadMXBean` functions. We will do examples with these functions in class. To ensure both accurate measurement and termination in a reasonable amount of time, you should try to have any code that you time take between 5 and 15 seconds. If your code falls in that range on `linux.uncg.edu`, then it should have acceptable running time on our submission/testing server as well.

Hints for testing cache impacts: To test cache performance, consider initializing a large array (say 10,000,000 int elements) by storing values in the array in order, and then having a second test that initializes the same array, but in non-sequential order. Modern machines are very fast, of course, so initializing 10,000,000 elements takes a fraction of a second — enclose this initialization in another loop that repeats the test some number of times so that the running time is long enough to be accurately measured. Note that cache performance is particularly sensitive to other processes running on the machine, so do not expect to get the exact same result if you run your program multiple times.

Hints for testing floating point performance: Consider a loop that computes a floating point value using a lot of single or double precision computations. For example, you can add up the sums of squares of 100,000,000 floating point values. While it is tempting to call math functions for more complex floating point operations, this can be tricky — for example, calling `sqrt()` with a `float` value will result in the value being cast to a `double` and the square root being taken in double precision before casting back to a `float`. There is a `sqrtf()` function that works with single-precision values, but it's probably best just to avoid library functions and stick with fundamental arithmetic operations. Note that the value you compute must be used in some way, or the compiler optimizations (which are turned on at the submission server) may cause you some grief. For example, if you compute the sum of squares of a bunch of values, and then don't use that sum for anything, the compiler will probably remove the entire loop from your code (since it doesn't produce anything useful), and then your time will be zero. One way to "use" a value is to output it to standard error — note that you should put this print statement *outside* your timing loop so it doesn't affect your measured times.

For both cache timing and floating point operations, you will have two cases that differ in a single aspect, and you should take care to make the cases as much alike as possible otherwise. If your loop for non-sequential access updates the position using a mod operation, make sure you include a mod operation in a meaningful way in the sequential access loop. Of all of the properties that can change from one loop to the next, ensure as much as you can that the property you are testing is the only one that actually does change.

Input and Output

There is no input to this program, and standard output should contain just two floating point values. The sample output below reflects one particular machine, but other machines might have quite different results.

Sample Output

```
3.241
1.001
```

Problem 6.R2: Happy Meals

Required Problem
Points: 50 points

Background

In Column 7, Bentley describes "Little's Law" as a method of estimating measures related to a system in which entities enter and leave the system at a stable average rate. This is a simple example of an area known as "queueing theory," which can also deal with considerably more complex problems. In this problem you are to consider a problem from this area that is more complex than can be handled by a "back of the envelope" calculation with Little's Law.

The Problem to Solve

After years of hard work and dedication to your computer science studies, you have graduated and landed your dream job: working at McBurger-fil-a. In between asking customers if they would like fries with their order, a situation arises in which you can use your computer science skills. Here's what happened: the power went out, and customers have been arriving at the store and don't know what to do. There are n customers milling about aimlessly when the power turns back on, and your job is to figure out how to organize the customers so that you minimize the unhappiness of the least happy customer, where "unhappiness" is measured by the amount of time that the customer has to wait between their arrival at the store and the delivery of their McDelicious meal. The store has m lines/registers that work independently, and each meal takes t minutes to prepare. You are given the times at which customers arrive at the store (while the power is out), and the time that the power comes back on. Since you're so smart, you can assume that you instantly put customers in lines when the power comes back on, so there is no time wasted arranging customers. You are to calculate the longest time a customer has to wait for his or her meal in the best possible arrangement (i.e., the minimum possible maximum wait). To make things easier, no customers will arrive after the power has been turned on, so the customers never need to be rearranged after the power comes on.

For example, consider customers that arrive at 1:44, 1:35, and 1:38 (call these customers A, B, and C, respectively), and the power comes on at 1:46. If your store has two lines, and each meal takes 5 minutes to prepare, then your best arrangement is to put customers B and A in one line (in that order), and customer C in the other. Customer B receive his meal after a total wait of 16 minutes, customer A will receive her meal after a 12 minutes, and customer C will receive his meal after a wait of 13 minutes. Therefore the most unhappy customer will be Customer B, who waited a total of 16 minutes.

In this problem, you are guaranteed that $1 \leq n \leq 10,000$ and $1 \leq m \leq 10$. Furthermore, $1 \leq t \leq 50$. Your program should solve any such problem in under 5 seconds.

Input and Output

The first line of the input contains three integers: n (the number of customers), m (the number of lines in the store), and t (the number of minutes required to prepare a meal). This is followed by n lines, each containing a time that a customer arrived at the store, and a final line containing the time that the power turns on. All times will be after 1:00 in the afternoon on the same day (so time wrap-around is not an issue). Your program should output a single line containing the worst wait time for any customer, in minutes. The sample input and output below show the sample problem described above.

Sample Input

```
3 2 5
1:44
1:35
1:38
1:46
```

Sample Output

```
16
```

Problem 6.C1: The Return of Happy Meals

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points

Background

Problem 6.R2 dealt with a queueing problem, but made an assumption: all lines prepare food at the same rate. In this problem we remove that restriction, and for each line give the amount of time it takes to prepare a meal. This can cause quite a difference: For example, consider a store in which one line produces a meal in 2 minutes, and the other line produces a meal in 5 minutes. If two customers arrive at the store at the same time, it's actually better to put them both in the faster line (since two meals are prepared in 4 minutes total) and not use the slower line at all (which would take 5 minutes for a single meal).

The Problem to Solve

This problem is the same as Problem 6.R2, except that instead of one meal preparation time t, there are m times — one for each line.

Hints and Techniques

Note that this problem is a strict generalization of 6.R2, so any program that solves this problem can also be used (with only minor changes) to solve 6.R2. Consider creating a core function that can solve this problem, and then for 6.R2 providing a "shim" that converts that problem's input into this one — both programs can use the exact same core function!

Input and Output

The first line of the input contains two integers: n (the number of customers) and m (the number of lines in the store). The second line contains m integers, which give the meal-producing time for each of the lines. This is followed by n lines, each containing a time that a customer arrived at the store, and a final line containing the time that the power turns on. All times will be after 1:00 in the afternoon on the same day (so time wrap-around is not an issue). Your program should output a single line containing the worst wait time for any customer, in minutes.

See sample input/output on the following page.

Sample Input

```
8 4
2 5 3 8
1:33
1:41
1:36
1:34
1:41
1:35
1:42
1:36
1:46
```

Sample Output

```
16
```

Problem 6.C2: McReservations

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points

Background

After your success in handling the power outage problem at McBurger-fil-a, you have been tasked with implementing a new business model, and you get to use more of your computer science talents. The new business model requires customers to make a reservation the day before, saying when they will arrive at the store, and the latest acceptable time for them to leave with their order. They might have very complicated orders, so the length of preparing the order varies and is specified (or calculated) when the reservation is made. To reduce costs at the store, there is just a single production line, and it must work on a single order at a time (and cannot stop before that order is fully prepared). In addition, in order to protect against wasted food due to no-shows, the production line will not start work on an order until that customer arrives at the reserved time (but production can start the instant the customer arrives). Your job is to take all of the next day's reservations, and determine whether it is possible to fill all orders within these constraints.

Consider these two cases:

Case 1: Possible

Customer	Arrival	Time	Deadline
A	1:08	12	1:28
B	1:11	3	1:18
C	1:15	6	1:33

Case 2: Impossible

Customer	Arrival	Time	Deadline
A	1:08	12	1:25
B	1:11	3	1:18
C	1:15	6	1:33

In the first case, you don't start on A's order immediately, but rather wait for B to arrive, and prepare B's order from 1:11 to 1:14. Then you prepare A's order from 1:14 to 1:26, and C's order from 1:26 to 1:32. With this schedule, all customer deadlines are met.

In Case 2, it is impossible to do so: to fill B's order, the latest you can start is 1:15, which leaves only 7 minutes between A's arrival and when you must start B's order, which is not enough time to complete A's order. Furthermore, the earliest B's order could be ready (freeing up the production line) is 1:14, which leaves only 11 minutes until A's deadline — again, not enough time to complete the order.

The Problem to Solve

You must write a program to determine whether it is possible to fill all orders. You will be given the number of customers n, and information on arrival time, preparation length, and deadline on each of the n customers. Your new model is a boutique service, so you do not have very many customers: you may assume that $1 \leq n \leq 12$, and your program must determine whether scheduling is possible in less than one minute.

Hints and Techniques

Think about the bound on n here: Do you really have to be very clever with the algorithm?

Input and Output

The first line of the input contains an integer `n`, giving the number of customers who have made reservations. The next `n` lines give, for each customer, the time they will arrive (formatted as a time), the length of time required to prepare their order (in minutes, an integer), and the time they must leave (again formatted as a time). As in the previous problems, all times will be after 1:00 in the afternoon on the same day (so time wrap-around is not an issue). Your program should output a single line containing either "`Yes`" (if it is possible to complete all orders) or "`No`" (if it is impossible to do so).

Sample inputs and outputs for the two cases described in the "Background" section are given below.

Sample Input 1

```
3
1:08 12 1:28
1:11 3 1:18
1:15 6 1:33
```

Sample Output 1

```
Yes
```

Sample Input 2

```
3
1:08 12 1:25
1:11 3 1:18
1:15 6 1:33
```

Sample Output 2

```
No
```