Introduction to Algorithms	Problem Set 2
CS 482 Spring 2004	Due February 13, 2004

Please hand in each problem on a separate sheet with your name on each.

Reading: Sections 2.1-2.4. (Section 2.3 on Dijkstra's algorithm is a review of material from earlier courses; we will not be covering this chapter in lecture, but will expect you to know the material in it.)

(1) You're working with a group of security consultants who are helping to monitor a large computer system. There's particularly interest in keeping track of processes that are labeled "sensitive". Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a little program called `status_check` that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of `status_check` as lasting for only this single point in time.) What they'd like to do is to run `status_check` as few times as possible during the day, but enough that for each sensitive process P, `status_check` is invoked at least once during the execution of process P.

(a) Give an efficient algorithm that finds as small a set of times as possible at which to invoke `status_check`, subject to this requirement.

(b) While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of k sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke `status_check` at least k times: no one invocation of `status_check` can handle more than one of these processes."

This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that k^* is the *largest* value of k such that one can find a set of k sensitive process with no two ever running at the same time. Is it the case that there must be a set of k^* times at which you can run `status_check`, so that some invocation occurs during the execution of each sensitive process? (In other words, is the kind of argument in the previous paragraph really the only thing forcing you to need a lot of invocations of `status_check`?) Decide on an answer to this question, and justify it with a proof or a counter-example.

(2) *(This is Problem 7 at the end of Chapter 2.)* Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins

swimming the 20 laps; as soon as he/she's out and starts biking, a third contestant begins swimming ... and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts.

What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

(3) Embolded by their success in Problem (1), your security consultant friends have branched out into the financial domain; they're currently advising a client who is investigating a potential money-laudering scheme. The investigation thus far has indicated that n suspicious transactions took place in recent days, each involving money transferred into a single account. Unfortunately, the sketchy nature of the evidence to date means that they don't know the identity of the account, the amounts of the transactions, or the exact times at which the transactions took place. What they do have is an *approximate time-stamp* for each transaction; the evidence indicates that transaction i took place at time $t_i \pm e_i$, for some "margin of error" e_i. (In other words, it took place sometime between $t_i - e_i$ and $t_i + e_i$.)

In the last day or so, they've come across a bank account that (for other reasons we don't need to go into here) they suspect might be the one involved in the crime. There are n recent *events* involving the account, which took place at times $x_1, x_2, \ldots, x_n$. To see whether it's plausible that this really is the account they're looking for, they're wondering whether it's possible to associate each of the account's n events with a distinct one of the n suspicious transactions in such a way that, if the account event at time x_i is associated with the suspicious transaction that occurred approximately at time t_j, then $|t_j - x_i| \leq e_j$. (In other words, they want to know if the activity on the account lines up with the suspicious transactions to within the margin of error; the tricky part here is that they don't know which account event to associate with which suspicious transaction.)

Give an efficient algorithm that takes the given data and decides whether such an association exists.