Introduction to Algorithms	**Problem Set 1**
CS 482 Spring 2004	**Due February 6, 2004**

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

Reading: Chapter 1 and Appendix A. Appendix A is primarily a review of material from CS 211, 280, and 312; we will not be covering this chapter in lecture, but will expect you to know the material in it (except the subsection on "Finding Cut-Points in a Graph").

(1) Running time analysis of algorithms using $O(\cdot)$ notation (and to a lesser extent $\Omega(\cdot)$ notation) is an important piece of background material for this course, and something that you should have seen in courses like CS 312. This question provides some practice in applying this style of analysis.

Consider the following basic problem. You're given an array A consisting of n integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$ — that is, the sum $A[i] + A[i+1] + \cdots + A[j]$. (The value of array entry $B[i, j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.)

Here's a simple algorithm to solve this problem.

```
For i = 1, 2, ..., n
  For j = i + 1, i + 2, ..., n
    Add up array entries A[i] through A[j]
    Store the result in B[i, j]
  Endfor
Endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n. (I.e. a bound on the number of operations performed by the algorithm.)

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

(c) Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem — after all, it just iterates through the relevant entries of the array B, filling in a value for each — it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n \to \infty} g(n)/f(n) = 0$.

(2) *(This is Problem 1 at the end of Chapter 1.)* Gale and Shapley published their paper on the stable marriage problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

- First type of instability: There are students s and s', and a hospital h, so that
  - s is assigned to h, and
  - s' is assigned to no hospital, and
  - h prefers s' to s.
- Second type of instability: There are students s and s', and hospitals h and h', so that
  - s is assigned to h, and
  - s' is assigned to h', and
  - h prefers s' to s, and
  - s' prefers h to h'.

So we basically have the stable marriage problem as in Chapter 1, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an efficient algorithm to find one. The input size is $\Theta(mn)$; ideally, you would like to find an algorithm with this running time.

(3) Consider the following scenario. n computer science students get flown out to the Pacific Northwest for a day of interviews at a large software company. The interviews are organized as follows. There are m time slots during the day, and n interviewers, where $m > n$. Each student S has a fixed *schedule* which gives, for each of the n interviewers, the time slot in which S meets with that interviewer. This, in turn, defines a schedule for each interviewer I, giving the time slots in which I meets each student. The schedules have the property that

- each student sees each interviewer exactly once,
- no two students see the same interviewer in the same time slot, and
- no two interviewers see the same student in the same time slot.

Now, the interviewers decide that a full day of interviews like this seems pretty tedious, so they come up with the following scheme. Each interviewer I will pick a *distinct* student S. At the end of I's scheduled meeting with S, I will take S out for coffee at one of the numerous local cafes, and they'll both blow off the entire rest of the day drinking espresso and watching it rain.

Specifically, the plan is for each interviewer I, and his or her chosen student S, to *truncate* their schedules at the time of their meeting; in other words, they will follow their original schedules up to the time slot of this meeting, and then they will cancel all their meetings for the entire rest of the day.

The crucial thing is, the interviewers want to plan this cooperatively so as to avoid the following *bad situation*: some student S whose schedule has not yet been truncated (and so is still following his/her original schedule) shows up for an interview with an interviewer who's already left for the day.

Give an efficient algorithm to arrange the coordinated departures of the interviewers and students so that this scheme works out and the *bad situation* described above does not happen.

Example: Suppose $n = 2$ and $m = 4$; there are students S_1 and S_2, and interviewers I_1 and I_2. Suppose S_1 is scheduled to meet I_1 in slot 1 and meet I_2 in slot 3; S_2 is scheduled to meet I_1 in slot 2 and I_2 in slot 4. Then the only solution would be to have I_1 leave with S_2 and I_2 leave with S_1; if we scheduled I_1 to leave with S_1, then we'd have a bad situation in which I_1 has already left the building at the end of the first slot, but S_2 still shows up for a meeting with I_1 at the beginning of the second slot.