Design and Analysis of Algorithms **out:** January 20, 2015

Problem Set 1

Lecturer: Daniel A. Spielman **due: 2:35** PM, January 29, 2015

Instructions. Read Carefully

1. Make sure that your yale **netid** appears on **every** page of your problem set, and that it is written clearly.

2. The problem sets are due at the beginning of class. Solution sets will be provided in class. For this reason, late problem sets are not accepted.

3. You will **hand in each problem separately**, so make sure that they are not stapled together and that no problem appears on the back of another. There will be 4 boxes in the front of the class, one for each problem. Put each problem in the correct box.

4. Alternatively, you may submit the problem set electronically via the Classes server. If you do so, submit it as a **pdf** and make sure that your **netid** appears on **every** page.

5. Cite any resources that you use, other than the textbook, TA and instructor.

6. You **must not** search the web for solutions to similar problems given out in other classes.

Collaboration Policy

You must write the solutions to the problems independently and in your own words.

However, you are allowed to discuss the problem sets in small groups of no more than 4 students. To ensure that you understand the solutions you submit, you are forbidden from taking written notes during your discussions. You must list at the top of the problem set everyone (other than course staff) with whom you have discussed the problem set. Failure to list people with whom you have discussed a problem set is considered a violation of academic honesty.

Problem 1: Perfect Stable Matching by Suitability

We will consider instances of the perfect stable matching problem in which the preferences of people and jobs are dictated by how suitable each person is for each job. Let the n people be $p_1, \ldots, p_n$ and let the n jobs be $j_1, \ldots, j_n$. We will assume that for each person p_a and each job j_b, we are given as input a numerical score $X(a, b)$ indicating how suitable person p_a is for job j_b. A higher score means better suited. We will assume that each person prefers the job for which she/he is best suited and that each job prefers the person best suited to the job.

Formally, we will assume that job j_b prefers person p_a to person $p_{a'}$ if $X(a, b) > X(a', b)$. Similarly, we will assume that person p_a prefers job j_b to job $j_{b'}$ if $X(a, b) > X(a, b')$.

Prove that if all of the numbers $X(a, b)$ are distinct, then the perfect stable matching is unique (that is, there is only one).

Example: We can think of X as a matrix. If it looks like:

$$X(p_1, j_1) = 1 \quad X(p_1, j_2) = 4$$
$$X(p_2, j_1) = 2 \quad X(p_2, j_2) = 3,$$

then the perfect stable matching will be $(p_1, j_2), (p_2, j_1)$.

Note: It is **not** sufficient to give one example for which this assertion is true. You must prove that it is true regardless of n and regardless of the numbers $X(a, b)$, provided that they are all distinct.

Hint: A good way to go about solving a problem like this is to look at a few small examples. Draw some examples with $n = 3$ or $n = 4$. Find the unique perfect stable matching. A natural way to prove the assertion is to describe the unique perfect stable matching in terms of X, and then prove that there is no other perfect stable matching. One would do this by proving that every other perfect matching is unstable.

Problem 2: The difficulty of finding a fair stable matching

It would obviously be desirable to find an algorithm for finding perfect stable matchings that is "fair". There are many ways of defining what it means for the result of an algorithm to be fair. One that has been proposed is to have the algorithm return a random perfect stable matching[1]. For example, we could consider an algorithm that

[1]While this seems better than using the person or job-optimal matching, I actually think that it is not a great definition of fairness. I think it is appealing merely because we do not understand what it will return

first constructs all possible perfect stable matchings, and then chooses one at random to return.

The difficulty with doing this is that some preference lists lead to there being many different perfect stable matchings. You will prove this.

a. Give an example of preference lists for people and jobs that leads to at least two different perfect stable matchings. State what those perfect stable matchings are.

   Note that it is possible to do this with only 2 people and only 2 jobs. (3 points)

b. Show that there exists a constant $c > 1$ so that for an infinite sequence of integers n there are preference lists for n people and n jobs that result in at least $2^{n/c}$ distinct perfect stable matchings.

   The easiest way to do this would be to describe the preference lists, identify the perfect stable matchings, and explain why they are stable.

   In my solution, I do this for even n and $c = 2$. (7 points)

Note: It may be possible to sample a random perfect stable matching without computing the list of all of them. One can do this for many other problems. But, I do not know of any efficient way of doing it for perfect stable matchings.

Problem 3: Scheduling a Supercomputer

Yale has a few supercomputing facilities known as the Bulldog clusters. Faculty who have need for a lot of computing power can submit jobs to be performed on these machines. Whenever one is setting up a facility such as this, one needs to decide on a sensible policy for choosing the order in which jobs run on the machines. For example, if one job will take a week to run and another will only take an hour, it seems reasonable to run the job that will only take an hour before the job that will take a week[2]. The problem is made even more complicated by the fact that these clusters consist of many cores, so that one can run many jobs at once and one can also vary the number of cores dedicated to each computing job. We will consider a complicated version of this problem later in the course. For now, we consider a simple version.

We will assume that we know in advance[3] all of the n jobs that we want to run on the supercomputer, and that only one job will run at a time. We assume that job i will take time t_i, and that the numbers $t_1, \ldots, t_n$ are distinct. We let π give the order in which we

[2]While it is unreasonable to expect to know how long jobs will take, most supercomputer scheduling systems require the users to supply estimates.

[3]Another way in which the real problem is difficult is that we do not know the jobs in advance.

run the jobs. That is, we run job $\pi(1)$ first, then job $\pi(2)$, and so on. If we run the jobs in the order π, then job $\pi(k)$ will finish at time

$$f_{\pi(k)} \stackrel{\text{def}}{=} \sum_{i=1}^{k} t_{\pi(i)}.$$

We define the *unhappiness* of job i (or of the professor to whom it belongs) to be the ratio of when it finishes to how long it takes:

$$u_i \stackrel{\text{def}}{=} f_i/t_i.$$

Note that $u_i \geq 1$ for all i. Your job is to minimize the maximum unhappiness:

$$u_{max} \stackrel{\text{def}}{=} \max_i u_i.$$

a. Describe a polynomial-time algorithm that takes $t_1, \ldots, t_n$ as input and returns an order π that minimizes the maximum unhappiness. If it is not completely obvious that your algorithm runs in polynomial time, explain why it does. (3 points)

   Note: You should explain your algorithm simply.

b. Prove that your algorithm does in fact minimizes the maximum unhappiness. (7 points)

Problem 4: Clustering — the easy case

One of the most important problems in data science is clustering. In a clustering problem, one is usually given many things and is asked to partition them into groups so that those inside each group are similar to each other. This is often a hard problem. In this problem, we will consider an easy variant.

The things we will cluster will be real numbers. You are given as input n real numbers, $x_1, \ldots, x_n$. Formally, a clustering of these numbers is a collection of sets $C_1, \ldots, C_k$ so that the union of C_1 through C_k is $\{1, \ldots, n\}$, and so that $C_i \cap C_j = \emptyset$ (the empty set) for all $i \neq j$. We define the *width* of a cluster C_i to be the maximum difference between two elements of C_i:

$$\mathsf{width}(C_i) = \max_{a,b \in C_i} |x_b - x_a| .$$

We define the *width* of the whole clustering to be the maximum of the width of its clusters

$$\mathsf{width}(C_1, \ldots, C_k) = \max_i \mathsf{width}(C_i).$$

It is easy to find a clustering of zero width: just assign each point to its own cluster. The problem becomes more challenging when you must find a clustering with few clusters.

For example, if the input is

$$x_1 = 1.1, x_2 = 4.1, x_3 = 4.4, x_4 = 6.2, x_5 = 4.9,$$

then the sets $C_1 = \{1, 3\}$ and $C_2 = \{2, 4, 5\}$ give a clustering of width 3.3.

a. Design a polynomial time algorithm that takes as input $x_1, \ldots, x_n$ and a target width w, and partitions the numbers into *as few clusters as possible* so that each cluster has width at most w. This is, you algorithm should find a width w clustering with smallest k. (4 Points)

   Note: You should explain your algorithm simply. If you need to write a lot of pseudo-code, you should explain what the code does. Pseudo-code should look like an imperative language, not like a functional language. The grader should not have to simulate your algorithm to understand it!

b. Prove that your algorithm is correct. That is, prove that it finds a clustering that has width at most w, and prove that among all clusterings of width at most w it finds a clustering with as few clusters as possible. (6 Points)