CS180 Homework 1

Due: April 19, 11:59pm

1. (15 pt) (Kevin) In the class we showed that the number of iterations (number of proposals) in the Gale-Shapley algorithm is upper bounded by $\leq n^2$ for n men and n women, since each man can only make $\leq n$ proposals. However, we haven't shown a lower bound on number of iterations. To show the lower bound is also in the order of n^2, please give a way to construct the preference lists for n men and n women such that the Gale-Shapley algorithm will run for $\Theta(n^2)$ iterations. (For simplicity, you can assume that the algorithm always chooses the unmatched man with the smallest index at each iteration).

   Solution: One possible solution: All men have the same list $w_1 > w_2 \cdots > w_n$, and all the women have the same list $m_n > m_{n-1} \cdots > m_1$.

   Overview:
   m_1 picks w_1.
   m_2 picks w_1, kicking off m_1. Next iteration sees m_1 unmatched, who then goes on to w_2.
   m_3 picks w_1, kicking off m_2. m_2 moves on to w_2, kicking of m_1, who moves on to m_3.

   Base case: m_1 is the first unmatched, so goes with w_1. Inductive claim: The first time m_i is the next unmatched, a chain reaction of i iterations happens, leaving m_k is with w_{i-k+1} for $k \in [1, i]$. (The rest of the men and women are free)

   Inductive proof: The first time m_i is the next unmatched, he proposes to w_1, and is her best choice since he's the highest m_i seen so far. She drops m_{i-1}, who is now the lowest unmatched. He moves on to w_2, kicking off m_{i-2}. This chain reaction continues, ending with m_1 being dumped by w_{n-1}, and then next iteration m_1 ends up with w_i. So after m_i proposes and the chain reaction settles, i more steps happen, and m_k is with w_{i-k+1} for $k \in [1, i]$.

   In total, this is $\sum_i^n i = n(n+1)/2 = \Omega(n)$. Since we already have $O(n)$, this gives $\Theta(n)$

2. (20 pt) (Lucas) Gale and Shapley published their paper on the Stable Matching 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 Matching Problem, 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 algorithm to find one.

Solution: The same as the original Gale-Shapley stable matching algorithm shown in class, with a few modifications. First, treat the hospitals as the "men" (group that proposes) and the students as the "women" (group that accepts or rejects). Second, don't consider a hospital "matched" until they have filled all their open positions. (So even if they are already matched with a few students, they keep proposing. Just like the original algorithm, if they are full and one of their students decides to leave, they start proposing again.)

To prove stability, first we have to prove this gives a matching where all hospitals are filled. This is easy to see as if a hospital was left not full, it would have had to proposed to every student. But in order for that to occur every student must already be matched to another hospital, which contradicts our assumption that there is a surplus of students.

Second we need to prove that both types of instability can not exist. The first instability can't exist because if h did prefer s', then h would have proposed to s' before s. But this means that s' must be committed to some hospital, which is a contradiction from the instability assumption that s' is not assigned.

The second instability is essentially the same proof used in the original algorithm. If h prefers s' to s, they must have proposed to s' at some point because otherwise they would have never proposed to s. But if s' didn't end up matching with h, that means they must be with a hospital they prefer more than h, meaning that they prefer h' to h. This contradicts our original assumption.

Because we have given an algorithm that is guaranteed to terminate and give a stable matching, we have also shown that it always exists.

3. (20 pt) (Yuanhao) The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be indifferent between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with $n = 4$), a woman could say that m_1 is ranked in first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in last place. We will say that w prefers m to m' if m is ranked higher than m' on her preference list (they are not tied).

   With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

   (a) A *strong instability* in a perfect matching S consists of a man m and a woman w, such that each of m and w prefers the other to their partner in S. Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

   (b) A *weak instability* in a perfect matching S consists of a man m and a woman w, such that their partners in S are w' and m' , respectively, and one of the following holds:

   - m prefers $w$to w', and w either prefers m to m' or is indifferent between these two choices; or
   - w prefers m to m', and m either prefers w to w' or is indifferent between these two choices.

   In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

Solution: (a) Yes, there always exists a perfect matching with no strong instability. Assuming the preference list with tie is P, we can break ties arbitrarily to get a full preference P' and then run $G-S$ algorithm.

A stable matching in P' will imply a strong stable matching in P, and we know $G-S$ will always give a stable matching for any P' so it can also find us a strong stable matching on P.

(b) No, there does not always exist a weak stable matching. Counterexample: m_1: $w_1 > w_2$, m_2: $w_1 > w_2$; w_1: $m_1 = m_2$, w_2: $m_1 = m_2$. For either matching, a weak instability exists.

4. (10 pt) (Noor) Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

   - $f_1(n) = \frac{n^4}{\log n}$
   - $f_2(n) = \sqrt{2n}$
   - $f_3(n) = n^2/2$
   - $f_4(n) = 2^{3\log n}$
   - $f_5(n) = n(\log n)^{100}$
   - $f_6(n) = 2^{\log n - \log\log n}$

   Solution: $f_2, f_6, f_5, f_3, f_4, f_1$

   We can deal with functions f_1, f_2, and f_4 very easily, since they belong to the basic families of exponentials, polynomials, and logarithms. In particular, $f_2(n) = O(f_5(n))$, $f_5(n) = O(f_3(n))$, $f_3(n) = O(f_1(n))$. For f_4, it's equal to $(2^{(}logn))^3 = n^3$ so it's between 1 and 3. For f_5, it's $2^{(logn)}/2^{loglogn} = \frac{n}{logn}$ so it's between 3 and 5 (since 2 can be viewed as $\frac{n}{n^{1/2}}$ and $n^{(\frac{1}{2})} \geq logn$ when n is large enough).

5. (20 pt) (Xuanqing) You're doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not have it break. We call this the highest safe rung.

   It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $n/4$ or $3n/4$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer.

   If your primary goal were to conserve jars, on the other hand, you could try the following strategy. Start by dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar – at the moment it breaks, you have the correct answer – but you may have to drop it n times (rather than log n as in the binary search solution). So here is the trade-off: it seems you can perform fewer drops if you're willing to break more jars. To understand better how this tradeoff works at a quantitative level, let's consider how to run this experiment given a fixed "budget" of $k \geq 1$ jars. In other words, you have to determine the correct answer – the highest safe rung – and can use at most k jars in doing so.

   (a) Suppose you are given a budget of $k = 2$ jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. (In other words, it should be the case that $\lim_{n\to\infty} f(n)/n = 0$.)

   (b) Now suppose you have a budget of $k > 2$ jars, for some given k. Describe a strategy for finding the highest safe rung using at most k jars. If $f_k(n)$ denotes the number of times you need to drop a jar according to your strategy, then the functions f_1, f_2, f_3, . . . should have the property that each grows asymptotically slower than the previous one: $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ for each k.

   Solution: (a) Strategy for two jars: suppose we drop the first jar at level k, if not broken then try the next level at $k + (k-1)$, if still not broken then $k + (k-1) + (k-2)$, and so on. If the first jar broken at any time (let's say at level $k + (k-1) + \cdots + (k-i)$, then we know that the highest safe rung is between $k + \cdots + (k-i+1)$ and $k + \cdots + (k-i+1) + (k-i)$. Then we drop the second jar from $k + \cdots + (k-i+1)$ up to $k + \cdots + (k-i+1) + (k-i)$ one level each time. In total the first jar takes i drops and the second jar

takes $k-i$ drops. In total $f(n)=k$. This strategy can cover at most $k+(k-1)+\cdots+1=k(k+1)/2$ levels, while we have n levels. So we must choose k such that

$$k(k+1)/2 \geq n \Longrightarrow f(n)=k \approx \sqrt{2n}=\mathcal{O}(\sqrt{n}) \Longrightarrow f(n)/n \to 0 \text{ as } n \to \infty.$$

Comments 1: In fact, we don't need to come up with this optimal strategy in order to solve this problem. Simply drop the first jar at $\lfloor\sqrt{n}\rfloor+1$ should also work. Because it only takes $n/(\lfloor\sqrt{n}\rfloor+1) \approx \lfloor\sqrt{n}\rfloor$ drops in the first round. After that, the possible range shrinks to $\lfloor\sqrt{n}\rfloor$ levels. So it takes another $\lfloor\sqrt{n}\rfloor$ drops. In total it is $\mathcal{O}(\sqrt{n})$.

Comments 2: Students coming up with strategies with $\mathcal{O}(\sqrt{n})$ with either optimal or non-optimal such as the one in comments 1 should get all credits.

(b) If we have k jars, inspired by (a) we can have simpler strategy: first jar drops at multiple of $n^{(k-1)/k}$, second jar at multiple of $n^{(k-2)/k}$, etc. So first jar will drop at most $n/n^{(k-1)/k}=n^{1/k}$ times. But after jar 1 breaks, we successfully shrink the range to $[i\cdot n^{(k-1)/k},(i+1)\cdot n^{(k-1)/k}]$. So because the 2nd jar is tested at the step size of $n^{(k-2)/k}$, it will be tested at most $n^{(k-1)/k}/n^{(k-2)/k}=n^{1/k}$ times, and so on. The last jar (k-th) will be tested at the step size of $n^{(k-k)/n}=1$, so it will surely find the exact position of the highest safe rung. In total the number of experiments if we have k jars are $f_k(n)=k\cdot n^{1/k}$ and similarly $f_{k-1}(n)=(k-1)\cdot n^{1/(k-1)}$. $\frac{f_k(n)}{f_{k-1}(n)}=\frac{kn^{1/k}}{(k-1)n^{1/(k-1)}}=O(n^{-1/k(k-1)})$ converges to 0.

6. (15 pt) (Xiangning) Given K sorted linked lists, each one contains n numbers in the ascending order. Give an algorithm to merge these K lists into a single sorted list. Your algorithm should run in $O(nK\log K)$ time.

Solution: Go through the lists, use a heap to maintain the current element of each list; pop the smallest one. Say $e_{ij},\ i=1,...,n,\ j=1,...,K$ represents the i-th element of the j-th list.

Step 1: Initialize the heap with the first element of K lists, which is e_{1j}.

Step 2: Pop the smallest element from the heap, output the element, and push the next element from the same list if exists. For example, if we pop e_{13} then we insert e_{23} to the heap. If we pop the last element of a certain list we don't need to push another element to the heap.

Step 3: Continue such process until the heap is empty.

There are at most K elements in the heap at any time, so popping the smallest one costs $O(\log K)$. Since we need to push and pop all the nK elements so the time complexity is $O(nK\log K)$.

Alternate solution mentioned by student: Merge sort, merge pairs, repeat. nK per iteration, log K iterations (???)

★ Homework assignments are due on the exact time indicated. Please submit your homework using the Gradescope system. Email attachments or other electronic delivery methods are not acceptable. To learn how to use Gradescope, you can:

- 1. Watch the one-minute video with complete instructions from here: https://www.youtube.com/watch?v=-wemznvGPfg
- 2. Follow the instructions to generate a PDF scan of the assignments: http://gradescope-static-assets.s3-us-west-2.amazonaws.com/help/submitting_hw_guide.pdf
- 3. **Make sure you start each problem on a new page.**

★ We recommend to use LaTeX, LyX or other word processing software for submitting the homework. This is not a requirement but it helps us to grade the homework and give feedback. For grading, we will take into account both the correctness and the clarity. Your answer are supposed to be in a simple and understandable manner. Sloppy answers are expected to receiver fewer points.

★ Unless specified, you should justify your algorithm with proof of correctness and time complexity.