Homework 1 Solution

Due: September 10, in class.

Instruction. There are 6 mandatory problems and 1 bonus problem in this homework. The 6 mandatory problems worth 100 points in total. For all algorithm design problems, you need to provide proofs on the correctness and running time of the algorithms unless otherwise instructed.

Problem 1 [16 pts]

Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

True or false? Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m. Then in every stable matching S for this instance, the pair (m, w) belongs to S.

Solution: The claim is true.

Proof of Correctness: Let us assume there are two men and two women m, m' and w, w' respectively. By contradiction let us consider a stable match S, where the pair (w, m) is not in S which implies that w is engaged to a man m', and m is engaged to a woman w'. But as per the claim, w likes m better than m' , and m likes w better than w' which implies that S contains an instability, contradicting the fact that it is stable.

Problem 2 [16 pts]

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: Here is the algorithm.

1: Set every hospital's candidate set to be empty.
2: **while** everyone of those not in a candidate set of a hospital has no hospitals on their preference lists **do**
3: **for** each student s **do**
4: s is assigned to the highest hospital on his/her preference list.
5: **for** each hospital h **do**
6: h sorts students who are assigned to it according to its preference list and picks up those rank top as a candidate set.
7: For those who are not chosen by h, they will cross out h from their preference lists.
8: Output every hospital's candidate set.

Proof of Correctness: The first thing we can prove is that the algorithm is guaranteed to stop. This is pretty clear. Each student begins with a list containing a finite number of hospitals. Each iteration, at least one hospital gets crossed out from one student's preference list. So the maximum number of iterations the algorithm could last is the number of students times the number of hospitals.

The next thing to prove is that a hospital's candidate set can only remain the same or improve as time goes on. That is to say, the worst student in the candidate set in one iteration will not be better than the worst student in the set in a later iteration. This is because any student in the candidate set one iteration will definitely return in the next iteration. The only reason they will ever be eliminated from the candidate set is if a better student shows up. So the set can't get worse. Also, it is clear that the cardinality of a candidate set will never decrease.

Next we prove that the matching algorithm produces a stable assignment. Suppose there exists first type of instability, such that student s' is assigned to no hospital and h prefers s' but was assigned s. Since s' is assigned to no hospital, s' must have been booted from h at some point. But somehow, h ended up with student s who is worse than student s'. This violates the theorem that h's candidate set cannot get worse. Suppose there exists second type of instability, such that

student s' is assigned hospital h' but prefers hospital h and h prefers s' but was assigned s. Since s' prefers h to h', s' would have visited h before visiting h'. Since s' didn't end up at h, s' must have been booted from h at some point. But somehow, h ended up with student s who is worse than student s'. This violates the theorem that h's candidate set cannot get worse. Thus, the final pairing is stable.

Finally, we prove that all available positions in all hospitals are filled. Suppose h's all available positions are not fully filled. If not, there exists at least one student whose preference list is empty. Let the student be s. Then s must have visited h before. Since s is finally eliminated, we can conclude that h's positions are fully filled which is a contradiction. Therefore, the assumption is false.

Above all, the proof is complete.

Problem 3 [20 pts]

Let $T = \{(1,3),(3,5),(2,7),(4,6),(5,7),(6,10),(8,9),(7,9),(9,10)\}$ denote the (start, finish) times for a collection of 9 tasks. (Note that the intervals are open.) Use the greedy algorithms we learned from class to solve the following problems. Please state the key steps in your simulation of the algorithms.

a) Solve the Interval Scheduling Problem for this collection of tasks (i.e. find the maximum number of tasks that can be scheduled on a single machine, and give a set of compatible tasks that achieves this maximum).

Solution: For the Interval Scheduling Problem, we first sort the intervals by their *finish times*.

j	s_j	f_j
1	1	3
2	3	5
3	4	6
4	2	7
5	5	7
6	7	9
7	8	9
8	6	10
9	9	10

Having done this sorting, then we schedule the tasks in order, omitting those that conflict with already scheduled tasks. In the end, the tasks we will schedule using this procedure will be

$$(1,3),(3,5),(5,7),(7,9),(9,10),$$

for a total of five tasks we can schedule (which is the most we could schedule on a single machine).

b) Solve the Scheduling All Intervals Problem for this collection of tasks (i.e. find the minimum number of machines required to complete all tasks, and give a schedule for doing so).

Solution: This is similar to the above problem, but this time we'll sort the tasks in order by their *start times*.

j	s_j	f_j
1	1	3
2	2	7
3	3	5
4	4	6
5	5	7
6	6	10
7	7	9
8	8	9
9	9	10

Now we schedule the tasks, in order, on machines so that they don't conflict. We will put a task on the lowest numbered machine that is possible. Doing so, we'll end up this schedule.

m_1 (1,3), (3,5), (5,7), (7,9), (9,10)
m_2 (2,7), (8,9)
m_3 (4,6), (6,10)

Thus we need three machines to schedule all tasks, and can do no better than three machines.

Problem 4 [16 pts]

In an **Art Gallery Guarding Problem** we are given a line L that represents a long (straight) hallway in an art gallery. We are also given a set $X = \{x_1, x_2, \ldots, x_n\}$ of real numbers that represent locations where painting are hung in the hallway. Suppose that a single guard at position y can protect all the paintings in the interval $[y-1, y+2]$. We can place guards at any position on the line L to protect paintings.

Design an algorithm with running time that is polynomial in n, for finding a placement of the guards that uses the minimum number of guards to guard all the paintings with positions in X.

Solution: First of all, without loss of generality (by sorting and renumbering if necessary) we will assume that $x_1 < x_2 < \cdots < x_n$.

If it's not clear from the problem description, we're assuming that we can place a guard at any real number (and they need not be placed at a particular painting, but can be placed in between paintings).

With this in mind, we will place the first guard at the real number $x_1 + 1$. This guard will cover the painting at position x_1, and possibly others. So we can find the smallest index j such that $x_{j-1} \leq x_1 + 3$ but $x_1 + 3 < x_j$. (In other words, this first guard can watch paintings at positions $x_1, \ldots, x_{j-1}$, but not the one at position x_j.)

This leaves us with a subproblem consisting of paintings $x_j, \ldots, x_n$ for some $j \geq 2$. Then we repeat the argument above. This will give us the minimum number of guards necessary to guard all the paintings.

Proof of Correctness: Suppose S is a set of some paintings. Let $f(S)$ denote the minimum number of guards necessary to guard all the paintings in S. If $S = \emptyset$, then $f(S) = f(\emptyset)$ is defined as 0.

Let S_0 be $\{x_1, x_2, \ldots, x_n\}$ which is the whole set containing all the paintings and S_1 be $\{x_j, x_{j+1}, \ldots, x_n\}$ where j is defined as above procedure. We are going to prove that

$$f(S_0) = 1 + f(S_1).$$

Since $S_0 = ((S_0 \backslash S_1) \cup S_1)$, then guards guarding $S_0 \backslash S_1$ together with guards guarding S_1 can also guard S_0 which means

$$f(S_0) \leq f(S_0 \backslash S_1) + f(S_1).$$

Also, we can get $f(S_0) \geq f(S_1)$ since S_1 is a subset of S_0. By definition of j, we know that only one guard is needed to guard $S_0 \backslash S_1$. Therefore, so far, we can get $f(S_1) \leq f(S_0) \leq 1 + f(S_1)$.

Actually, it is impossible to guard S_0 with only $f(S_1)$ guards. Suppose it was true, let G_1 be a set of guards with $|G_1| \leq f(S_1)$. After removing all of guards from G_1 that covered x_0, the left guards could still cover S_1 which means we could use at most $f(S_1) - 1$ guards to guard $f(S_1)$ which is impossible.

Above all, we can now focus on finding $f(S_1)$ since $f(S_0) = 1 + f(S_1)$. And this formula guarantees our greedy algorithm leading to the optimal solution.

Proof of Running Time: First, we may need to sort all the paintings and remove duplicate ones, which can be completed in $O(n \log n)$ time complexity. Then, we will start the iterations. For each iteration, the running time is $O(1)$ and at least one painting is removed. Thus, there are at most n such iterations. Hence, the total running time of this greedy algorithm is $O(n \log n) + O(n) = O(n \log n)$.

Problem 5 [16 pts]

Describe an $O(n \log n)$-time implementation of the greedy algorithm we learned in Lecture 02 for the Scheduling All Intervals problem (the algorithm is also in KT 4.1). Prove the running time of your algorithm. [Hint: use a heap/priority queue.]

Solution: The high level idea is that we maintain the intervals with the latest starting time within each label in a min-heap. And the keys are their ending times. Therefore, each time we want to check whether there exists a label in $\{1, 2, \ldots, d\}$ compatible with I_j, we can extract the interval with the earliest ending time in the heap. And, then check whether it is compatible with I_j. If it is, we can just assign this interval's label to I_j. If not, it means no labels can be assigned to I_j. And we should give I_j a new label.

Here is the new algorithm.

```
1: Sort the intervals by their start times, breaking ties arbitrarily and let I_1, I_2, ..., I_n denote the
   intervals in this order
2: Set min-heap H empty
3: d ← 0
4: for j = 1, 2, 3, ..., n do
5:     I ← ExtractMin(H)
6:     if I is compatible with I_j then
7:         Assign the label of I to I_j
8:     else
9:         d ← d + 1
10:        Assign d to I_j
11:        Insert(H, I)
12:    Insert(H, I_j)
```

Proof of Running Time: For sorting, we can use quick sort or heap sort to achieve $O(n \log n)$ time complexity (Line 1). For each j, there are at most one ExtractMin and two Insert(s) of heap H. And the running time for any such operation is $O(\log n)$. Therefore, total running time from Line 4 to Line 12 is bounded by $n \cdot O(\log n) = O(n \log n)$. Hence, total running time of this algorithm is $O(n \log n)$.

Problem 6 [16 pts]

Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point of time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Example: Consider the following four jobs, specified by (*start-time*, *end-time*) pairs.
(6 PM, 6 AM), (9 PM, 4 AM), (3 AM, 2 PM), (1 PM, 7 PM).
The optimal solution would be pick jobs *(9 PM, 4 AM)* and *(1 PM, 7 PM)* which can be scheduled without overlapping.

Solution: Note that this problem is like the interval scheduling problem discussed in the class; except that the jobs are periodic. One way of thinking about this not like an interval, but as a circle. Now each job would correspond to an interval in this circle. Let J be a job which is being done at midnight. Then we could remove all the jobs overlapping with J, and solve the remaining problem. Note that the remaining problem looks like the interval scheduling problem which was solved in the class. This can be solved the regular greedy algorithm. We however do not know the job J. This could be found out easily just by trying out all the possiblities for such a job.

For each job which contains a midnight time, we select it; solve the resulting interval scheduling problem. Finally, we pick the best such solution. The running time would be [Running Time of Greed]×[Number of Processes]. (In the worst case we would be running the greedy algorithm on each process. Thus, we get $O(n^2 \log n)$.

Problem 7 [Bonus, 30 pts]

Let us consider the following algorithm for the Scheduling All Intervals problem.

- Sort all intervals according to the non-decreasing order of finishing times (break ties arbitrarily)
- For each interval I in the order
  - If there exists at least one processor that can process interval I
    - Choose the processor P that can process I with the latest finish time (break ties arbitrarily)
    - Assign I to processor P
  - Else
    - Add a new processor and assign I to the newly added processor

Please either give a counterexample to this algorithm or prove the correctness of the algorithm.

Solution: The algorithm is correct. One possible proof is as follows.

First, it is easy to check the algorithm always output a valid solution. Then we need to prove that the algorithm outputs an optimal solution.

Each time the algorithm introduces a new processor (namely the $(d+1)$-st processor), we will identify a time t so that $d(t)$ (the depth at time t) is at least $(d+1)$. And this would show that Min's algorithm uses at most the number of processors used by the optimal solution (which is at least $d(t)$).

Now let us see how to identify t and $d(t)$. Suppose the interval assigned to the new processor is $(s,\ f)$. At this time there are already d existing processors P_1, P_2, ..., P_d. We will sort these d processors in a way so that the finishing time of the last interval (the interval with the greatest finishing time) is in non-decreasing order. (i.e. finishing time of $P_1 \leq$ finishing time of $P_2 \leq$ ...).

Let the last interval on P_1 be $(s_1,\ f_1)$. Now we set t $= f_1 - \epsilon$ for some extremely small $\epsilon > 0$. We see that at time t, both $(s_1,\ f_1)$ and $(s,\ f)$ are active, where $(s,\ f)$ is active because otherwise (s, f) would be able to added to P_1.

Now we will identify an interval for each P_i where $(i = 2, 3, ..., d)$ so that $d(t)$ is at least $(d+1)$. Fix an $i = 2, 3, ..., d$, we see that the last finishing interval on P_i finishes at time greater or equal to f_1 (because of our sorting). Among all the intervals on P_i with finishing time at least f_1, we choose the one with the smallest finishing time, namely $(s_i,\ f_i)$. We claim that $s_i < f_1$ – otherwise, when the algorithm considers $(s_i,\ f_i)$, it is also compatible with intervals on P_1, and the finishing time on P_1 is greater than the finishing time on P_i (since $(s_i,\ f_i)$ is the first job with $f_i \geq f_1$) – this means that $(s_i,\ f_i)$ should have been added to P_1 (or some other processor) instead of P_i. To summarize,

we have $s_i < f_1$ and $f_i \geq f_1$, which means that $(s_i,\ f_i)$ is active at time t which immediately before f_1.

In total, we have identified $(d + 1)$ intervals active at time t immediately before f_1. These intervals are $(s_1,\ f_1)$, $(s_2,\ f_2)$, ..., $(s_d,\ f_d)$, and $(s,\ f)$. Therefore $d(t) \geq d + 1$ and this shows that the algorithm's output is optimal.