Homework 1 – Solutions

Solutions to Assigned Problems

Following are the problems to be handed in, 25 points each.

1. (**Resident Matching**, 2-page limit – your solutions should fit on two sides of 1 page).

The situation is the following. There were m teams at Google, each with a certain number of available positions for hiring interns. There were n students who want internships at Google this summer, each interested in joining one of the teams. Each team had a ranking of the students in order of preference, and each student had a ranking of the teams in order of preference. We will assume that there were more students who want an internship at Google than there were slots available in the m teams.

The interest, naturally, was in finding a way of assigning each student to at most one team at Google, in such a way that all available positions in all teams were filled. (Since we are assuming a surplus of potential interns, there would be some students who do not get assigned to any team.) We say that an assignment of students to Google teams is stable if neither of the following situations arises.

- The first type of instability that can occur is that there is a team t, and there are students s and $s\prime$, so that
  - s is matched with t, and
  - $s\prime$ is assigned to no team, and
  - t favors $s\prime$ over s.
- The second type of instability that can occur is that there are teams t and $t\prime$ and students s and $s\prime$ so that
  - s is matched with t, and
  - $s\prime$ is matched with $t\prime$, and
  - t favors $s\prime$ over s, and
  - $s\prime$ favors t over $t\prime$.

So we basically have the Stable Matching Problem, except that, one, teams generally want more than one intern, and, two, there is a surplus of students who want internships at Google. Show that there is always a stable assignment of students to Google teams, and give an algorithm to find one.

Please give a clear description of your algorithm. Don't forget to prove its correctness and analyze its time and space complexity.

Solution 1:

The algorithm is very similar to the basic Gale-Shapley algorithm from the text. At any point in time, a student is either "commited" to a team, or "free". A team either has available positions, or it is "full." The algorithm is the following:

```
While some team t_i has available positions
    t_i offers a position to the next student s_j on its preference list
    if s_j is free then
        s_j accepts the offer
    else (s_j is already committed to a team t_k)
        if s_j prefers t_k to t_i then
            s_j remains committed to t_k
        else s_j becomes committed to t_i
                the number of available positions at t_k increases by one.
                the number of available positions at t_i decreases by one.
```

The algorithm terminates in $O(mn)$ steps because each team offers a positions to a student at most once, and in each iteration, some team offers a position to some student. The space complexity is just $O(mn)$ because we need to maintain the preferences lists for the teams and students.

Suppose there are $p_i > 0$ positions available at team t_i. The algorithm terminates with an assignment in which all available positions are filled, becuase any team that did not fill all its positions must have offered one to every student; but then, all these students would be committed to some team, which contradicts our assumption that $\sum_{i-1}^{m} p_i < n$.

Finally, we want to argue that the assignment is stable. For the first kind of instability, suppose there are students s and $s\prime$, and a team t as above. If t prefers $s\prime$ to s, then t would have offered a position to $s\prime$ before it offered one to s; from then on $s\prime$ would have a position at *some* team, and hence would not be free at the end - a contradiction.

For the second kind of instability, suppose that (t_i, s_j) is a pair that causes instability. Then t_i must have offered a position to s_j, for otherwise it has p_i residents all of whom it prefers to s_j. Moreover, s_j must have rejected t_i in favor of some t_k which he/she preferred; and s_j must therefore be committed to some t_l (possible different from t_k) which he/she also prefers to t_i.

2. (**Time complexity**, 2-page limit – your solutions should fit on two sides of 1 page). Part (a) has 15 points and part (b) has 10 points. The top of your solution for part (a) should have the functions in order by their letter, with no spaces, commas, etc. between them. (For example, abc). If you do not include this you will automatically lose 75% of the credit. (Functions that are equivalent should be in alphabetical order)

   (a) Rank the following functions by increasing order of growth, that is, find an arrangement $g_1, ...$ of the functions satisfying $g_1(n) = O(g_2(n)), g_2(n) = O(g_3(n)), ...$ Break the functions into equivalence classes so that f and g are in the same class if and only if $f(n) = \Theta(g(n))$. Note that $\log(\cdot)$ is the base 2 logarithm, $\log_b(\cdot)$ is the base b logarithm, $\ln(\cdot)$ is the natural logarithm, and $\log^c(n)$ denotes $(\log(n))^c$ (for example, $\log^2(n) = \log(n) \times \log(n)$).

a. $\ln(\ln n)$	**b.** $n \log n$	**c.** $14 \log_3 n$	**d.** $\sum_{i=5}^{n} \frac{(i+1)}{2}$	**e.** $\log^2(n)$
f. n^2	**g.** $\sum_{i=1}^{n} \left(\frac{1}{2}\right)^i$	**h.** $\log(n!)$	**i.** 3^n	**j.** $n^{\log 7}$
k. $\sum_{i=1}^{n} 3^i$	**l.** $2^{\log^2(n)}$	**m.** $2^{\log n}$	**n.** $n!$	**o.** n
p. $2^{\log_4 n}$	**q.** $\sqrt{n}$	**r.** $\log(n^2)$	**s.** $4^{\log n}$	**t.** $(\frac{5}{4})^n$

(b) For each of the following statements, decide whether it is always true, never true, or sometimes true for asymptotically nonnegative functions f and g. If it is always true or never true, give a proof. If it is sometimes true, give one example for which it is true, and one for which it is false.

i. $f(n) + g(n) = \Omega(\max(f(n), g(n)))$
ii. $f(n) = \omega(g(n))$ and $f(n) = O(g(n))$
iii. Either $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ or both.

(a) Here is the ordering, where functions on the same line are in the same equivalence class, and those higher on the page are ω of those below them. In some cases, we added a simpler function in the same equivalence class, to help you understand the ordering.

Solution: gacrepqmobhdfsjltikn
$\sum_{i=1}^{n}(\frac{1}{2})^i = 1 - \left(\frac{1}{2}\right)^n = \Theta(1)$
$\ln(\ln n)$
$14 \log_3 n = \frac{14 \log n}{\log_2 3}$ and $\log(n^2) = 2 \log n$ (both are $\Theta(\log n)$)
$\log^2(n)$
$\sqrt{n} = 2^{\log_4 n}$
$n = 2^{\log n}$
$n \log n$ and $\log(n!)$
$n^2 = 4^{\log n}$ and $\sum_{i=5}^{n} \frac{i+1}{2} = \frac{(n-4)(n+7)}{2}$
$n^{\log 7}$
$2^{\log^2(n)} = n^{\log n}$
$(\frac{5}{4})^n = 2^{(\log_2 5 - 2)n}$
$\sum_{i=1}^{n} 3^i = \frac{3^{n+1} - 1}{2}$ and 3^n
$n! = 2^{\Theta(n \log n)}$

(b) i. Always true. Proof: Since f and g are asymptotically nonnegative, there exist n_f and n_g such that for all $n > n_f$, $f(n) \geq 0$ and for all $n > n_g$, $g(n) \geq 0$. Let $n_0 = \max(n_f, n_g)$ and $c = 1$. For any $n > n_0$, the sum $f(n) + g(n)$ is greater than $f(n)$ (since $g(n)) \geq 0$) and greater than $g(n)$ (since $f(n) \geq 0$), and hence greater than the larger of the two. Thus, for $n > n_0$, we have $f(n) + g(n) \geq c \max\{f(n) + g(n)\}$, as desired.

ii. Never true. Proof: $f = O(g)$ implies that for $n > n_1$, we have $f(n) \leq c_0 g(n)$ for some $c_0 > 0$. But by the definition of ω, for this c_0 there must exist n_2 such that for all $n > n_2$, we have $f(n) > c_0 g(n)$. Now consider some $n > \max(n_1, n_2)$. We get $f(n) \geq c_0 g(n) < f(n)$, which implies $f(n) < f(n)$, a contradiction.

iii. Sometimes true and sometimes false. We've seen many examples where the statement is true (for example, $f(n) = n$ and $g(n0 + n^2)$. For an example where it is false, we must find f and g such that $f(n) \notin O(g(n))$ and $f(n) \notin \Omega(g(n))$. For example

$$f(n) = n \quad \text{and} \quad g(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases}.$$

(1) Suppose that $f = O(g)$, so that $f(n) \leq cg(n)$ for some $c > 0$ and all $n \geq n_0$. Let n be the first *even* number greater than n_0. At that point $f(n) > 0$ and $cg(n) = 0$, which is a contradiction to $f(n) \leq cg(n)$. So $f \notin O(g)$. (2) Suppose that $g = \Omega(g)$. so that $f(n) \geq cg(n)$ for some $c > 0$ and all $n \geq n_0$. Let n be the first *odd* number that is greater than $\max(n_0, \frac{1}{c} + 1)$. At that point $f(n) = n$ and $cg(n) = cn^2 \geq cn(\frac{1}{c} + 1) > n$, which is a contradiction to $f(n) \geq cg(n)$. So $f \notin \Omega(g)$.

3. (**Induction**, 2-page limit – your solutions should fit on two sides of 1 page). Part (a) has 10 points and part (b) has 15 points.)

(a) (**Uniform shuffling**) Let $A[1, \cdots, n]$ be an array of integers. A uniform shuffle of A is a set of n random elements from A (without replacement), such that the probability of selecting any such set is the same. Consider the following algorithm to generate a uniform random shuffle:

UNIFORMSHUFFLE(A)

```
1  for i ← n downto 1
2        do j ← random integer such that 1 ≤ j ≤ i
3           exchange A[i] and A[j]
4  return A
```

Prove that the algorithm indeed generates a uniform random shuffle of A. What is the running time of the algorithm, given that generating random integer takes time $O(1)$? *Hint: Start by thinking of what a uniform shuffle means in terms of probability.*

(b) Point out the error in the following proof by induction.

Claim: Given any set of b buses, all buses lead to the same destination.

Proof: We proceed by induction on the number of buses, b.

Base case: If $b = 1$, then there is only one bus in the set, and so all buses in the set lead to the same destination.

Induction step: For $k \geq 1$, we assume that the claim holds for $b = k$ and prove that it is true for $b = k + 1$. Take any set B of $b + 1$ buses. To show that all buses lead to the same destination, we take the following approach. Remove one bus from this set to obtain the set B_1 with just b buses. By the induction hypothesis, all the buses in B_1 lead to the same destination. Now go back to the original set and remove a different bus to obtain a the set B_2. By the same argument, all the buses in B_2 lead to the same destination. Therefore all the buses in $B = B_1 \cup B_2$ must lead to the same destination, and the proof is complete.

(a) The claim can be proven by induction on the number of elements in A. First, observe that what we aim to prove is that, for any array A that has n elements, each of the $n!$ permutations of the array should be equally likely to be the output of the algorithm.

Therefore, the claim that we prove by induction is that:

$S(j)$: Given an n element array A, after the j-th run of UniformShuffle(where $1 \leq j \leq n$), the elements in the positions $A[n-j+1], \ldots, A[n]$ store a uniformly random sample of size j from A. In other words, this sample occurs with probability $\frac{(n-j)!}{n!}$.

Observe how in this proof, we fix the size of the array A to be n, and we proceed to do induction on j.

Base case: After the first run, when $i = n$ and $j = n - i + 1 = 1$, the last element of the array will be a uniformly random sample of size 1 from A. In other words, after the first run, any of the elements in the array can be on the last position.

Induction hypothesis: We assume that for a given array A of length n, after the j-th run of the algorithm, the elements on positions $A[n-j+1], \ldots, A[n]$, represent a uniformly random set of j elements from A. This set occurs with probability

$$\frac{1}{n} \cdot \ldots \cdot \frac{1}{n-j+1} = \frac{(n-j)!}{n!}$$

Induction step: In the next step of the algorithm i.e. when $i = n - (j+1) + 1 = n - j$, any element from the subset $A[1], \cdots, A[n-j]$ can end up in position $A[n-j]$. Thus, there are $n - j$ choices each occuring with a probability of $\frac{1}{n-j}$. Therefore, using the induction hypothesis, we know that the new set of size $j + 1$ will occur with probability

$$\frac{1}{n-j} \cdot \frac{(n-j)!}{n!} = \frac{(n-j-1)!}{n!}$$

This completes the proof.

If it's not yet clear why this is the case, think of the last run of the algorithm, when $j = n$. This means that for $j = n$, the last run yields a subset of size n with probability

$$\frac{(n-n)!}{n!} = \frac{1}{n!}$$

The running time of the algorithm is obviously $O(n)$.

(b) The error here is that, no matter what the number of buses, the intersection of the two sets B_1 and B_2 must be non-empty, and this does not hold for $b = 2$. To see why, consider by $B = \{b_1, b_2\}$. Obviously in the two subsets, $B_1 = \{b_1\}$ and $B_2 = \{b_2\}$, all the buses go to the same destination. However, their intersection is empty and therefore we cannot make a connection between the two subsets. b_1 can go in a different direction from b_2 and thus the induction claim fails.

4. (**Divide and Conquer**, 2-page limit – your solutions should fit on two sides of 1 page).

After dating for several years, Jack and Anthony have finally decided to move in together. As part of this process, each of them wants to bring his n alphabetically sorted books over to the new place. Due to some weird reason, they want to find out who owns the median book of the

joint book collection, which has $2n$ books. In this joint book collection, the median would be the n-th book among the union of the $2n$ alphabetically sorted books.

Because their original book collections are already sorted, they manage to find out who owns the median in $\Theta(\log n)$. They did not have to reorder the joint book collection, but rather it was enough for them to just query individual values from their original book collections. What algorithm did they use? Prove that this algorithm is correct. Find the recurrence relation and show that it resolves to $\Theta(\log n)$.

This problem might seem complicated but it is in fact quite simple. Let's say that we keep Jack's books in an array A and Anthony's books in an array B. Both A and B are sorted arrays of length n, and our task is to find which array contains this median.

```
MEDIAN(A, A_start, A_end, B, B_start, B_end)
 1  ▷ returns the median of A[A_start..A_end] and B[B_start..B_end]
 2  ▷ to find the median of A and B, you would call MEDIAN(A, 1, A.length, B, 1, B.length)
 3  midA ← A_start + ⌊(A_end − A_start)/2⌋
 4  midB ← B_start + ⌊(B_end − B_start)/2⌋
 5  if A_start = A_end ▷ only one element in each array
 6     then
 7          if A[A_start] > B[B_start]
 8          return Jack
 9     else
10          return Anthony
11  if A[midA] > B[midB]
12     then
13          return MEDIAN(A, A_start, midA, B, B_start + ⌈(B_end − B_start)/2⌉, B_end).
14     else
15          return MEDIAN( A, A_start + ⌈(A_end − A_start)/2⌉, A_end), B, B_start, midB).
```

The basic idea is to compare the two middle elements of each array. Let us say without loss of generality that the element from A is larger. Then all the elements in the right half of A are larger than the elements in the left half of A and of B (of which there are at least n). Therefore, those elements in the right half of A cannot be the median. Similarly, the left elements of B are smaller than the right elements of both B and A, meaning that there are at least $n+1$ smaller elements. Hence, those left elements of B cannot be the median either. One has to be careful to consider what can be said about the middle elements themselves. Also, the code takes into account that n can be both even or odd.

The running time is the same as that of binary search and is thus given by

$$T(n) = T(n/2) + \Theta(1)$$

This is solved by $T(n) = \Theta(\log(n))$.