Ethan Wong
COM SCI 180
Discussion 1B
UID 305319001

Homework 1

1. The lower bound of the Gale-Shapley algorithm is also in the order of n^2. This can be demonstrated by the following preference list:

   M1 = {W1, W2, W3} W1 = {M3, M2, M1}
   M2 = {W1, W2, W3} W2 = {M2, M1, M3}
   M3 = {W1, W3, W2} W3 = {M1, M2, M3}

   Note: These preference lists will be stored as a linked list, and in the order of highest preference to lowest preference. For instance, M1 prefers W1 as his highest preference and W3 is his lowest preference.

   Running the algorithm will go as follows:
   - M1 proposes to W1 — they are paired
   - M2 proposes to W1 — they are paired, M1 is now unmatched
   - M3 proposes to W1 — they are paired, M2 is now unmatched
   - M1 proposes to W2 — they are paired
   - M2 proposes to W2 — they are paired, M1 is now unmatched
   - M1 proposes to W3 — this is the only woman available now, they are paired

In this scenario we see that that maximum number of times that a man can get rejected is twice. There are three women so after getting rejected twice, they are guaranteed to be matched with the final woman. We will call this maximum M. M1 gets unmatched the M times (twice). M2 gets unmatched (M-1) times. M3 gets unmatched (M-2) times. Because of this, we can see that the runthrough of the algorithm will run for $\Omega(n^2)$iterations. The encapsulating while loop will need to run n times. The preference list for M1 will also need to be traversed to the very end. There will be n list traversals for M1's preference list. This example of a possible preference list shows that the Gale-Shapley algorithm will run for $\Omega(n^2)$iterations.

2. An algorithm to solve this problem of hospitals and students can be derived by modifying the original Gale-Shapley Algorithm. In this algorithm, the hospitals will be extending job offers to the students — similarly to how men propose to women. The students will play a similar role to the women — they can either accept the "proposal", or job offer, or reject it in favor of another one. The algorithm is as follows:

Matching Algorithm:
Let the set of pairs S = ∅
Initially every job slot in every hospital 'h' is vacant and every student 's' has no offers
While there's a hospital h that hasn't offered to every student and has one vacant slot:
Select a hospital h that has at least one vacant job slot
s := h's most preferred student in their preference list that they haven't offered to yet
If s is free (has not accepted any prior offers):
s accepts offer from h, number of vacant job slots in h decreases by 1
(h, s) will become a pair, update S
Else if s has accepted a prior offer from another hospital h':
If s prefers h' to h:
s rejects offer from h and the job slot remains vacant, (s. h') remain paired
Else if s prefers h to h':
s will accept offer from h and h will have one less vacant job slot
s will reject previously accepted offer from h', h' now has one more vacant job slot
(h, s) will become a pair, update S
Return S, along with any students who were unmatched

This algorithm returns a stable matching and this can be proven by contradiction. Assume that there is a matching M that the algorithm returned that was unstable. For instance, in the first type of instability, s is matched with h, s' is unmatched, and h prefers s' to s. According to the algorithm above, this is not possible. The algorithm matches a student with their first offer. The fact that s' was unmatched at the end of the algorithm means she never received an offer. We know that the hospital h will make offers in order of preference, and s' is higher preference than s for this particular hospital h. There is a contradiction here — h would have extended an offer to s' before offering to s. This contradiction proves that this instability is impossible.

In the second type of instability, s is assigned to h, s' is assigned to h', h prefers s' over s, and s' prefers h over h'. According to the algorithm above, this is not possible. The hospital h would have extended an offer to s' before s because s' was a higher preference. And because s' prefers h over h', there would never be any reason for s' to reject an offer from h to accept an offer from h'. It does not make sense that s' would ultimately get assigned to h'. The fact that s' was not ultimately assigned to h means that h must have actually preferred s to s'. This contradicts the initial statement that h prefers s' over s. This contradiction proves that this type of instability can not occur.

These two proofs by contradiction show that the algorithm above indeed outputs a stable assignment of students to hospitals.

3. a)

Intuitively, this modification to the Stable Matching Problem where men and women can be indifferent between options is not all that different from the original Stable Matching Problem. For instance, say that a w_2 has both m_3 and m_4 tied as her favorite men. Also, w_2 is m_3's favorite woman and m_4's second favorite woman. Because w_2 is indifferent between m_3 and m_4, w_2 will be content with either of these men. Simply modify the Gale-Shapley algorithm so that when there is a tie between preferences, a random option is chosen (since the person proposing should be indifferent as there is a tie between preferences). Basically, if there is a tie between preferences they can choose randomly between the options who are all equal. The Gale-Shapley algorithm does not create any instabilities (proved in lecture) so there should be no problem with strong instabilities.
The algorithm is as follows:

Matching Algorithm:

```
Let the set of pairs S = ∅
Initialize all m ∈ M and w ∈ W to free
While there exists a free man m who still has a woman w he hasn't proposed to yet:
    w := first woman on m's preference list to whom m has not yet proposed*
    if w is free:
        (m, w) become engaged, update S
    else if a pair (m', w) already exists
        if w prefers** m to m':
            m' becomes free
            (m, w) become engaged, update S
        else
            (m', w) remain engaged
  Return S
```

*If there is a tie between the first woman on m's preference list to whom m has not yet proposed (i.e the man is indifferent between w_1 and w_2), just choose a random one between the ones who are equal.
**The definition of "prefer" comes from the problem — "w prefers m to m' if m is ranked higher than m' on her preference list (they are not tied)".

3. b) There does not always exist a perfect matching with no weak instability. This can be proved with an example of a set of men and women and their preference lists.

m: {w = w'} w: {m', m}
m': {w = w'} w': {m', m}

The first possible matching:

m ——- w
m' ——- w'

In this scenario, m' and w' are satisfied because w' prefers m' and m' is indifferent between w' and w. In the other engagement, m is indifferent between w and w', but w' prefers m'. This represents a weak instability.

In the second possible matching:

m ——- w'
m' ——- w

In this scenario, m' and w are satisfied because w prefers m' and m' is indifferent between w' and w. In the other engagement, m is indifferent between w and w', but w' prefers m'. This represents a weak instability.

Based on these given examples, there does not always exist a perfect matching with no weak instability. Both of the above cases contained a weak instability despite there being a perfect matching.

4. In ascending order of growth rate:
    - $f_2(n)$
    - $f_6(n)$
    - $f_5(n)$
    - $f_3(n)$
    - $f_4(n)$
    - $f_1(n)$

This order could be found by simplifying all the given functions and comparing their Big-O time complexity. For $f_1(n)$ and $f_6(n)$, I took advantage of the fact that $log(n) > n^k$ whenever $k > 0$. This means that $f_1(n)$ would be the slowest, as it was bounded by n^4. $f_6(n)$ would be second fastest as it was only bound by n. For $f_5(n)$, I used the fact that a log to any base raised to any power is always upper bounded by n^k for all $k > 0$. This is based off of the theorem that I mentioned previously. This led me to conclude that $f_5(n)$ has a time complexity less than $O(n^2)$. For $f_2(n)$, this function was quite simple and it was easy to see that it had a time complexity of $O(n^{1/2})$. For $f_3(n)$, it could be written as $(1/2) * n^2$, which leads to a time complexity of $O(n^2)$. Finally for $f_4(n)$, a little bit of simplification led to the time complexity of $O(n^3)$. I sorted these by ascending order of growth rate and it led to the list above.

5. a) A smart way to find the highest safe rung would be to drop a jar from $\sqrt{n}$ height. If the jar does not break, continue onwards — drop the jar from the $2\sqrt{n}$ height, $3\sqrt{n}$ height, etc until the jar breaks. We will call the rung from which the jar breaks the "$y\sqrt{n}$ height". In the worst case scenario, it should take $\sqrt{n}$ total steps to find the $y\sqrt{n}$ height. Once we find the height from which the jar will break when dropped, we have one jar remaining. We know that the highest safe rung will be in between the $y\sqrt{n}$ height and the $(y-1)\sqrt{n}^{\text{th}}$ height. Use the second jar and drop it at each of the rungs in between the $(y-1)\sqrt{n}^{\text{th}}$ and the $y\sqrt{n}$ height. In the worst scenario, this approach will find the highest safe rung in $2\sqrt{n}$ steps. The rung at which the jar breaks will be the highest safe rung. This approach has a time complexity of O($\sqrt{n}$).

   b) A somewhat similar approach can be used as in part A of this problem. Since k > 2 here, it would be best to first drop a jar from $n^{(k-1)/k}$ height intervals. This means the first jar will be dropped at most $2n^{1/k}$ times, because $[2n/n^{(k-1)/k}] = 2n^{1/k}$. Just like before, repeat this process until the breaking point is found — this will be called the $y * n^{(k-1)/k}$ height. The safe rung will then be between $y * n^{(k-1)/k}$ and $(y-1) * n^{(k-1)/k}$ height. Another jar can be used to determine the highest safe rung by dropping it at the rungs in this height range.
   As mentioned earlier, the first jar will be dropped at most $2n^{1/k}$ times. Now to find how many times the (k-1) jar will be dropped, we get $f_{k-1}(n^{(k-1)/k})$. This means that less than 2(k-1)$(n^{(k-1)/k})^{1/(k-1)}$, or $2(k-1)n^{1/k}$ total drops will be needed. This shows $f_k(n) < 2kn^{1/k}$
   .

6. The strategy behind solving this problem is to use the divide-and-conquer method that is frequently seen in algorithms. Basically the algorithm will pair two linked lists together and merge them. This process will repeat continuously until there is only one list left that contains all the elements. The time complexity for this algorithm is O(nKlogK). There is only O(n) complexity for merging two sorted linked lists, so the helper function does not contribute to this. In the finalMarge function, the outer loop will run log(k) times and it will process n*K elements per iteration.

```
struct Node{
  Node* next;
  int value;
};


//Helper function used to merge two sorted lists
Node* mergeTwo(Node* x, Node* y)
  if (y == NULL):
    return x;
  if (x == NULL):
    return y;
  Node* final = NULL;
  if ((x->value) > (y->value)):
    final = y;
    final->next = mergeTwo(x, y->next);
  else:
    final = x;
    final->next = mergeTwo(x->next, y);
  return final;



//Merges K sorted lists, uses helper function
Node* finalMerge(Node* cell[], int final)
  while (final != 0):
    int a = 0;
    int b = final;
    while (a < b):
      cell[i] = mergeTwo(cell[a], cell[b]);
      b = b - 1;
      a = a + 1;
      if (b <= a):
        final = b;
   return cell[0];
```