Solutions to Homework 1

1. (Review of Divide and Conquer) As with binary search, the idea is to maintain an interval \([\ell, u]\) that contains the secret number. No matter what my previous guess was, I can always find a new guess which will allow me to halve the length of the interval: if my previous guess was \(\text{prev}\), then my new guess will be \(\text{new} = 2\text{mid} - \text{prev}\), where \(\text{mid} = (\ell + u)/2\) is the midpoint of my current interval. We can verify that this indeed cuts the interval in two, since the distance between \(\text{mid}\) and \(\text{prev}\) is the same as the distance between \(\text{mid}\) and \(\text{new}\). Specifically, we have \(\text{mid} - \text{prev} = \text{new} - \text{mid}\). Our new interval will be contained in either \([\ell, \text{mid}]\) or \([\text{mid}, u]\). The formula above will only give us trouble in the case that \(\text{mid} = \text{prev}\) since then \(\text{next} = \text{prev}\) and query won’t give us any new information.

In the algorithm, we get around this by forcing \(\text{mid}\) to not be an integer. This ensures that we make progress on every iteration. It also means that we don’t have to worry how the guess function behaves when the real number is exactly half-way between \(\text{prev}\) and \(\text{new}\). See Algorithm 1.

Analysis We have to show that on each iteration of the algorithm the interval \(\ell, u\) contains the secret number, and that this interval shrinks sufficiently on each iteration. The following useful invariant simplifies our analysis:

Loop invariant: At the beginning of each iteration of the while loop, we have: (a) the secret number is contained in \(\{\ell, ..., u\}\) and (b) the number \(\text{prev}\) is an integer.

It’s easy to check that this holds the first time though. Let s prove by induction that it holds on every subsequent step. Let \(\text{med}_-\) and \(\text{med}_+\) be the integers immediately below and above \(\text{mid}\). Since \(u > \ell\), these integers are always in the interval \(\{\ell, ..., u\}\). If \(\text{prev} < \text{mid}\), then the integers \(\{\ell, ..., \text{mid}_-\}\) are closer to \(\text{prev}\) than \(\text{new}\), and the integers \(\{\text{mid}_+, ..., u\}\) are farther away. If \(\text{prev} > \text{mid}\), then the integers \(\{\ell, ..., \text{mid}_-\}\) are farther from \(\text{prev}\) than \(\text{new}\), and the integers \(\{\text{mid}_+, ..., u\}\) are closer. Either way, our algorithm correctly reassigns the bounds of the interval so that it still contains the secret number. Moreover, \(\text{new}\) is an integer since \(2\text{mid}\) and \(\text{prev}\) are integers; hence, \(\text{prev}\) will again be an integer on the next pass through the loop.

The algorithm terminates when either the niece tells us we’ve guessed correctly or the interval has been reduced to a single integer (in which case we’re certain of the answer). Moreover, the length of the new interval is always at most \(\lceil (u - \ell)/2 \rceil\). Thus the total number of iterations is at most \(\log_2(n) + 2\). We make one guess at the beginning plus at most one guess per iteration, for a total of at most \(\log_2(n) + 3\) guesses.

1
Algorithm 1: Guessing with a giggly niece

**Input:** Integer \( n \) and access to function “guess”

\[
\text{answer} \leftarrow \text{Guess}(\lfloor n/3 \rfloor);
\]

if \( \text{answer} = \text{correct} \) then
  \[ \text{Return} \lfloor n/3 \rfloor; \]
else
  \[ \ell \leftarrow 1, u \leftarrow n; \text{prev} \leftarrow 1; \]
  \[ \text{while} \ u - \ell > 0 \text{ do} \]
  \[ \quad \text{mid} \leftarrow \lfloor (\ell + u) / 2 \rfloor + 1/2; \]
  \[ \quad \text{new} \leftarrow 2\text{mid} - \text{prev}; \]
  \[ \quad \text{answer} \leftarrow \text{Guess}(\text{new}); \]
  \[ \quad \text{if} \ \text{answer} = \text{correct} \text{ then} \]
  \[ \quad \quad \text{Return} \text{new}; \]
  \[ \quad \text{if} \ (\text{answer} = \text{closer} \text{ and} \ \text{prev} < \text{mid}) \text{ or} \ (\text{answer} = \text{farther} \text{ and} \ \text{prev} > \text{mid}) \text{ then} \]
  \[ \quad \quad \ell \leftarrow \text{mid} + 1/2; \]
  \[ \quad \quad u \leftarrow \text{mid} - 1/2; \]
  \[ \quad \quad \text{prev} \leftarrow \text{new}; \]
  \[ \text{Return} u; \]

2. **(Stable matching with indifferences)** Chapter 1, problem 5.

(a) **(Strong instabilities)** Yes, there always exists a stable matching with no strong instabilities.

There are two ways to solve this problem. The first is to modify the Gale-Shapley algorithm so that women and men break ties arbitrarily, but ensuring that men never propose to the same woman twice. The second is to run the original Gale-Shapley algorithm on a modified input.

**First solution: Modified Gale-Shapley**

A modification of Gale-Shapley will output a matching with no strong instabilities (see Algorithm 2).

The only difference between this and the original G-S algorithm is that one man can only break another’s engagement when his ranking is (strictly) higher in the woman’s preference list.

**Correctness/termination:**

As with G.-S., the algorithm will always terminate because no man proposes to the same woman twice. The algorithm outputs a perfect matching, as before, since each man has every woman on his preference list and each woman will remain engaged once proposed to.

The correctness of G.-S. was shown in class. The modified algorithm is similarly correct because a strong instability in the input is, in particular, an instability
Algorithm 2: Gale-Shapley for no strong instability matching

| Input: Sets $M$, $W$ and preference lists |
| Output: List of engagements |
| Mark all $m \in M$ and $w \in W$ as “free”; |
| while $\exists m \in M$ who is free do |
| find some woman $w$ such that $m$ has not proposed to $w$ and no other woman not yet proposed to ranks higher than $w$; |
| if $w$ is free then |
| $(w, m)$ are engaged; |
| else |
| $m$ ranks higher than $w$’s current partner $m’$; |
| Break $(w, m’)$’s engagement; |
| $(w, m)$ are engaged; |
| (Note: If $w$ is indifferent between $m$ and current partner, do nothing.) |
| Output the engagement lists, send out the invitations and order the champagne; |

for the original stable matching problem with ties broken arbitrarily between indifferent groups of men and women.

**Time complexity** is $O(n^2)$ where $n$ represents the total number of men or women. This is because preference list takes $O(n^2)$ space, and in the worst case, each man checks all the women in his list. For each proposal, we can implement the necessary checks in $O(1)$ time (after $O(n^2)$ preprocessing), for a total of $O(n^2)$ time.

**Space Complexity** is also $O(n^2)$: as with G.S., the algorithm requires space to store the women’s inverted preference lists, and one pointer for each man indicating the last woman proposed to.

**Second solution: Reduction to standard stable matching.**
The other approach is to break ties in the input arbitrarily at the beginning, and then run any stable matching algorithm on these fully-ordered preference lists.

Algorithm 3: Second solution

| for each man $m$ do |
| create a totally ordered list of women (no indifferences) in the order of $w$’s preferences, listing women to whom he is indifferent in arbitrary order; |
| for each woman $w$ do |
| create a totally ordered list of men (no indifferences) in the order of $w$’s preferences, listing men to whom she is indifferent in arbitrary order; |
| Run Gale-Shapley on these modified preference lists; |
| Output resulting stable matching $\pi$; |

**Correctness** Consider the output $\pi$ of this algorithm. First, observe that every
strong instability in π with respect to the original preference lists is in fact a standard instability in π with respect to the modified preference lists. Given this observation, the correctness of the algorithm follows directly since the Gale-Shapley produces a stable matching for the modified preference lists.

**Time and space** This algorithm uses $O(n^2)$ time and $O(n^2)$ space. Processing the original inputs requires linear time since they can be simply copied to create the modified lists (this actually depends on the exact format of the input). Running Gale-Shapley takes $O(n^2)$ time, as seen in class. The space used is that required to store the inputs $O(n^2)$) and that required by Gale-Shapley (also $O(n^2)$).

(b) (Weak instabilities) No, there need not always exist a matching with no weak instabilities.

Example: Suppose there are two men \{$m_1, m_2$\} and two women \{w_1, w_2\}, where both men prefer $w_1$ to $w_2$, and each woman is indifferent between the two men. There are only 2 kinds of matchings: either $m_1 \leftrightarrow w_1$, $m_2 \leftrightarrow w_2$, or $m_2 \leftrightarrow w_1$, $m_1 \leftrightarrow w_2$. Both matchings have a weak instability since both men prefer $w_1$ and she is indifferent between the two of them.

3. **(Order of Growth Rate)** Chapter 2, problem 3.

- $f_2(n) = \sqrt{2n}$; $f_3(n) = n + 10$; $f_6(n) = n^2 \log n$; $f_1(n) = n^{2.5}$; $f_4(n) = 10^n$; $f_5(n) = 100^n$

Chapter 2, problem 4. Please add $g_6(n) = n!$ to the list of functions in #4.

- $g_1(n) = 2^{\sqrt{\log n}}$; $g_3(n) = n(\log n)^3$; $g_4(n) = n^{4/3}$; $g_5(n) = n^{\log n}$; $g_2(n) = 2^n$; $g_8(n) = n!$;
- $g_7(n) = 2^{2n}$; $g_6(n) = 2^{2n}$

4. **(Understanding big-O notation)** Chapter 2, problem 5.

- **(a)** This is true **under the condition that** $g(n) \geq 2$ **for all** $n > 0$. By definition, there exist positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. We can rewrite this log $f(n) \leq \log(c \cdot g(n)) = \log c + \log g(n)$. Since $g(n) \geq 2$, the ratio $\frac{\log(c)}{\log g(n)}$ is at most $\max(1, \frac{-\log(c)}{2})$. Let $c'$ be the constant $1 + \max(1, \frac{-\log(c)}{2})$. Then $\log(f(n)) \leq c' \log(g(n))$ for all $n \geq n_0$, and so $\log f(n) = O(\log g(n))$.

However, **this is false in general, since** $g(n)$ **may be close to 1 (or even less than 1)**. Counterexample: $f(n) = 4 \cdot 2^{1/n}$ and $g(n) = 2^{1/n}$. Taking $c = 1/4$ and $n_0 = 1$, we see that $f, g$ satisfy $f(n) = O(g(n))$. However, $\log f(n) = 2 + 1/n$ and $\log g(n) = 1/n$. The ratio $\frac{\log f(n)}{\log g(n)} = \frac{2 + 1/n}{1/n} = 2n + 1$ is unbounded, and thus there is no constant $c'$ such that $f(n) \leq c' \cdot g(n)$ for sufficiently large $n$.

- **(b)** False. Counterexample: $f(n) = 2n$, $g(n) = n$, since $2n = O(n)$ but $2^{2n} \neq O(2^n)$.

- **(c)** True. By definition of big-O notation, “$f(n) = O(g(n))$” means that there exist positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Then $f(n)^2 \leq (c \cdot g(n))^2 = c^2 \cdot g(n)^2$ for these $c$ and $n_0$. Thus, $f(n)^2 = c' \cdot g(n)^2$ for $c' = c^2$ and all $n \geq n_0$, and hence $f(n)^2 = O(g(n)^2)$.