CPSC 320 Sample Solution, The Stable Marriage Problem

1 Trivial and Small Instances

1. Write down all the **trivial** instances of SMP. We think of an instance as "trivial" roughly if its solution requires no real reasoning about the problem.

   SOLUTION: If you think of the smallest possible instances, it usually guides you towards trivial instances. In SMP, it's tempting to say that the smallest possible instance has one man and one woman, but we can go smaller! Degenerate cases like "zero men and zero women" are often helpful!

   So, is zero men and zero women trivial? Sure! There's exactly one solution, in which no one is matched with anyone else.

   What about one man and one woman? Regardless of their preferences (which, in fact, must be simply for each other), the only solution is for the one man to marry the one woman.

   So, zero men/women and one man/woman are the trivial instances.

   FROM SOLUTION REPRESENTATION: The solution for a problem with $n = 0$ is the empty set of pairings $\{\}$. The solution for a problem with $n = 1$ is $\{(m_1, w_1)\}$.

   What about two men and women? You might start that here, but you'll quickly realize it belongs in the next slot...

2. Write down two **small** instances of SMP. One should use the preference lists your group gave for the chocolate/donut example above:

   SOLUTION: Here's what we might have come up with. This is just a sample of three men and women and their preferences for each other that could come from the chocolate example:

```
w1: m2 m1 m3       m1: w3 w1 w2
w2: m1 m2 m3       m2: w3 w2 w1
w3: m2 m1 m3       m3: w2 w1 w3
```

   Each woman lists their preferences for men in order from most to least preferred.

   Each man lists their preferences for women in the same order.

   FROM PROBLEM REPRESENTATION: We can rephrase this with our new notation, but honestly, there's not much to do. $n = 3$, clearly. $W = \{w_1, w_2, w_3\}$, but all that changes there is subscripts vs. numbers on the side. $P[w_1]$ is the first list on the left. Similarly, we can see the men and the other preference lists.

   FROM SOLUTION REPRESENTATION: There happens to be only one stable solution to this instance: $\{(m_2, w_3), (m_1, w_1), (m_3, w_2)\}$.

   On to the next question...

   The other can be even smaller, but not trivial:

   SOLUTION: We can go smaller and still have a trivial example. So, let's do so. Two men and two women:

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

```
w1: m1 m2          m1: w1 w2
w2: m1 m2          m2: w2 w1
```

With two men/women, there are only two choices of preference list. So, I gave the women matching preference lists and the men opposite preference lists, just to illustrate both possibilities. That may be useful or may not!

FROM PROBLEM REPRESENTATION: I won't explicitly use our new names here, since little has changed, as noted above.

FROM SOLUTION REPRESENTATION: Again, there happens to be only one stable solution to this instance: $\{(m_1, w_1), (m_2, w_2)\}$. (Want one with more than one solution? Try tweaking w_1's preference list.)

2 Represent the Problem

1. What are the quantities that matter in this problem? Give them short, usable names.

   SOLUTION: You may have come up with more, fewer, or different quantities than me, but here are some useful ones.

   - n, the number of men and the number of women.
   - M, the set of men $\{m_1, m_2, \ldots, m_n\}$ (so, $n = |M|$)
   - W, the set of women $\{w_1, w_2, \ldots, w_n\}$ (again, $n = |W|$)
   - Each man's preference list—which we might call $P[m_i]$ for man i—is a permutation of W, the set of women. Note that I'm forcing the men to have complete preferences for all the women, no ties. That's the simplest version of the problem; so, probably the one to start with. I'll also assume that everyone prefers being married to not being married.
   - Similarly, each woman's preference list, $P[w_j]$, is a permutation of M.
   - It's good to have a notation to indicate whether a woman prefers one man to another (or similarly for a man). I'll use $m_i >_{w_j} m_k$ to mean that w_j prefers m_i to m_k, i.e., m_i occurs earlier in w_j's preference list than m_k.

2. Go back up to your trivial and small instances and rewrite them using these names.

   SOLUTION: See above.

3. Use at least one visual/graphical/sketched representation of the problem to draw out the largest instance you've designed so far:

   SOLUTION: This isn't a problem that suggests a lot of obvious graphical solutions, but I like drawing this as two columns with the preferences on the outside. That leaves space in the middle for us to draw lines in between men and women:

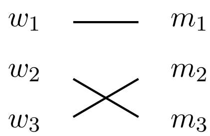
```
m2 m1 m3 :w1       m1: w3 w1 w2
m1 m2 m3 :w2       m2: w3 w2 w1
m2 m1 m3 :w3       m3: w2 w1 w3
```

   You might also create an $n \times n$ grid, with women across the top and men down the side. That would let you put information about each potential couple in each grid cell.

   Different graphical representations will suggest different information in the problem to focus on or ignore or facilitate particular ways of thinking about the problem. We'll keep pushing on this as the course proceeds!

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

FROM SOLUTION REPRESENTATION: Abandoning plain text, let's actually draw this:



4. Describe using your representational choices above what a valid instance looks like:

   SOLUTION: What "shape" is an instance (in programming terms, what inputs of what type constitute an input) and what additional constraints are there on inputs of that "shape" for them to be valid?

   The crucial piece of an instance is the preference lists for the men and women. We need to know how many of those there are.

   So, we might describe an instance as a tuple: (n, P_W, P_M), where n is the number of women (and also the number of men), P_W is a list of n preference lists for the women (where element i is w_i's preferences), and P_M is a list of n preference lists for the men.

   If you're more comfortable thinking in programming input/output terms, you might say that the input is: one line with a (non-negative) integer n, then n lines representing the women's preference lists each with n whitespace-separated numbers forming a permutation of $1, \ldots, n$, and finally n similar lines representing the men's preference lists.

3 Represent the Solution

1. What are the quantities that matter in the solution to the problem? Give them short, usable names.

   SOLUTION: Central to our solution are marriages, which are pairs (m_i, w_j) indicating that man i and woman j are married. A solution, then is a set of pairings (with some constraints we describe next).

2. Describe using these quantities what a **valid** solution looks like:

   SOLUTION: There's no technical weight here for the word "valid." But broadly speaking, we consider a solution invalid if it violates a constraint. In this case, we'll define validity as measuring whether we've successfully married everyone off just once (which was the constraint we were given for the problem). A valid solution is a perfect matching: a set of pairings such that each woman appears in exactly one pairing and each man appears in exactly one as well.

3. We haven't said what a **good** solution to this problem looks like. Brainstorm some different possible definitions of a good solution.

   SOLUTION: The word "good" is even less technically meaningful than the word "valid": it just defines a solution that's high-quality for some reason, and it's up to us to define some way to judge that (ideally, one that makes sense and that will make the problem easy to solve efficiently).

   We will define a good solution as a stable solution (defined below), but here are some alternative ways we might define a good solution:

   - One that has as many people with their first choice partner as possible
   - One that has as few people with their last choice partner as possible
   - One that maximizes a total "happiness" score, based on some way to compute happiness based on preference lists.

   And so on. You may have come up with different alternatives!

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

4. There are many reasonable ways we could define a **good** solution to this problem. However, for the remainder of this worksheet we will use **one particular** definition. **WAIT HERE** for us to provide that definition!

   SOLUTION: We choose to define a good solution as being "self-enforcing" in the sense that no man and woman who aren't married will decide to break the arrangement we suggested. So, a good solution is *stable* in that it contains no *instabilities*.

   Next, what's an instability? An instability can occur for a man m_i and woman w_j who are not matched in the solution ($(m_i, w_j) \notin$ the set of pairings). Let w' be m_i's partner in the solution and m' be w_j's partner. Then, m_i and w_j constitute an instability if $m_i >_{w_j} m'$ and $w_j >_{m_i} w'$. That is, each of m_i and w_j prefers the other to their assigned partners.

5. Go back up to your trivial and small instances and write out one or more solutions to each using these names.

   SOLUTION: See above.

6. Go back up to your drawn representation of an instance and draw at least one solution.

   SOLUTION: Again, see above.

4 Similar Problems

As the course goes on, we'll have more and more problems we can compare against, but you've already learned some. So...

Give at least one problem you've seen before that seems related in terms of its surface features ("story"), problem or solution structure, or representation to this one:

SOLUTION: You may not have enough background in problems to feel like you've seen a lot of similar problems, but you have at least seen problems where you organize a bunch of values by comparisons among them: sorting. If you've worked with bipartite graphs and matching problems, anything associated with them seems promising, especially maximum matching. (We often discuss "goodness" measures that give more points to a first preference than a second and so forth, like the Borda count. You could frame that problem as a maximum matching problem!) This also feels a bit like an election or auction, which takes us toward game theory. Maybe you'd even decide this feels a bit like hashing (mapping a value in one set to a different value in another set).

The point isn't to be "right" yet; it's to have a lot of potential tools on hand! As you collect more tools, you'll start to judge which are more promising and which less.

5 Brute Force?

You should usually start on any algorithmic problem by using "brute force": the most straightforward, non-optimized approach you can think of. In this case (and in my cases), we want to generate all possible solutions and test each one to see if it is, in fact, **the** solution we're looking for.

1. A possible SMP solution takes the form of a perfect matching: a pairing of each woman with exactly one man. We'll call a perfect matching a "valid" (but not necessarily good) solution.

   It's more difficult than the usual brute force algorithm to produce all possible perfect matchings; instead, we'll count how many there are. Imagine lining all the men up in a row in a particular order. How many different ways we can line up (permute) the women next to them?

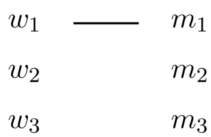
   SOLUTION: There are n women we can line up with the first man. Once we've chosen the first, there are $n-1$ to line up next to the second. Then, $n-2$ next to the third, and so on. Overall, then,

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

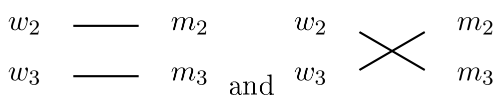
that's $n \times n-1 \times n-2 \times \ldots \times 2 \times 1 = n!$. There are $n!$ perfect matchings, our "valid" solutions. That's already super-exponential, even if it takes only constant time per solution to produce them!

We asked in the challenge problems for an algorithm to produce these. It's unusually challenging to design for a brute force algorithm, but it's useful to think about; so, we'll work through it here.

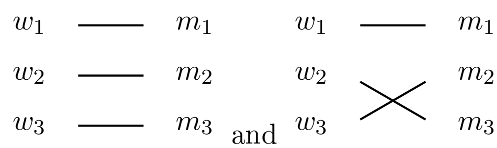
Before we dive into an algorithm, let's just try creating all solutions for an example. We might start by just marrying the first person (say, w_1) off to **someone**. After all, we know she needs to be married to somebody!



We can now set w_1 and m_1 aside, which gives us... another SMP instance that's smaller. As soon as you hear words like "and that leaves us with [something that looks like our original problem] but smaller", you should be thinking of recursion. Let's just assume we can recursively construct all possible solutions. That will give us back a bunch of sets of pairings, in this case two:



We can add our set-aside pairing (m_1, w_1) onto each of these:



That's all the solutions in which w_1 weds m_1. Who else can w_1 marry? Each of the other men. We can use the same procedure for each other possible pairing. Must w_1 marry someone? Yes, because we need a perfect matching. So, that covers all the possibilities for w_1 and, recursively, for everyone else.

Now we're ready for an algorithm. Let's call it AllSolns. It's recursive; so, what's the base case? Our trivial cases are where $n = 0$ or $n = 1$. Let's try $n = 0$ as a base case. Looking back at the trivial cases, I see the solution for $n = 0$ is the empty set of pairings $\{\}$. With that, let's build the algorithm. I'll use **return** when I'm producing the whole set at once and **yield** to produce one at a time. (You could just initialize a variable to the empty set and add in each **yielded** solution, returning the whole set at the end.)

```
procedure AllSolns(W, M)
    if |W| = 0 then                                   ▷ The base case we chose.
        return {{}}              ▷ The set of sol'ns, containing only the empty sol'n.
    else
        choose a w ∈ W                                ▷ Any one, e.g., the first.
        for all m ∈ M do                              ▷ Iterate through the men,
            for all S ∈ AllSolns(W − {w}, M − {m}) do ▷ and the subproblem sol'ns.
                yield {(m, w)} ∪ S                    ▷ Add the set-aside pairing.
            end for
        end for
    end if
end procedure
```

If we use our analysis techniques to count the number of solutions this creates, the analysis will parallel the recursive function itself. In the base case when $n = 0$, AllSolns produces one solution. Otherwise, for each of the n men, it makes a recursive call with $n' = n - 1$ (one fewer man and one

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

fewer woman in the subproblem). For each solution produced by that recursive call, it also generates one solution. If we give the number of solutions a name, we can express this as a recurrence:

$$N(n) = \begin{cases} 1 & \text{when } n = 0 \\ n * N(n-1) & \text{otherwise} \end{cases}$$

So, for example, $N(4) = 4*N(3) = 4*3*N(2) = 4*3*2*N(1) = 4*3*2*1*N(0) = 4*3*2*1*1 = 4!$. And, indeed, this is exactly the definition of factorial. So, $N(n) = n!$. There are $n!$ solutions to a problem of size n.

2. Once we have a possible solution, we must test whether it's the solution we're looking for. Informally, we'll refer to this as asking whether it's a "good" solution.

   A perfect matching is a good solution if it has no instabilities. Design a (brute force!) algorithm that—given an instance of SMP and a perfect matching—determines whether that perfect matching contains an instability. (As always, it helps to **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive. Remember, for brute force: generate each possible solution (possible instability, in this case) and then test whether it really is a solution.)

   SOLUTION: The form of a potential instability is a pair (man and woman). We therefore want to go through each pair of one man and one woman and check that (1) they are **not** already married (or they cannot cause an instability) and (2) they'd rather be with each other that their partners. (I'll assume we have a quick way to find a partner, which shouldn't be hard to create.) That should look like the following, where (n, P_W, P_M) is an instance of SMP and S is a solution to that instance (a perfect match and therefore a set of pairings (m_i, w_j)):

```
procedure IsStable((n, P_W, P_M), S)
    for all w ∈ {w_1, ..., w_n} do
        for all m ∈ {m_1, ..., m_n} do
            if (m, w) ∉ S then
                find m' such that (m', w) ∈ S
                find w' such that (m, w') ∈ S
                if m >_w m' and w >_m w' then
                    return false
                end if
            end if
        end for
    end for
    return true
end procedure
```

3. Exactly or asymptotically, how long does your algorithm take? (Again, you should explicitly name the size of an instance and perform your analysis in temrs of that name!)

   SOLUTION: Let's assume we do an efficient (constant-time) job of operations like comparing w's preferences for the two men and checking if (m, w) is in S. (It's not immediately obvious how to do this, but with some careful data structures and $O(n^2)$ preprocessing, it's doable!) The number of iterations in the inner loop is independent of which iteration we're on in the outer one. The body takes constant time. So, in the worst case (when we find no instability), this takes $|M| * |W| * O(1) = n * n * O(1) = O(n^2)$ time.

4. Brute force would generate each valid solution and then test whether it's good. Will brute force be sufficient for this problem for the domains we're interested in?

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

SOLUTION: Looks like brute force will take $O(n^2 n!)$ time. That's horrendous. It won't do for even quite modest values of n. (But, it is good enough to solve the $n = 3$ example we demonstrated in the classroom.)

6 Lower-Bound (Extra)

SOLUTION: We didn't discuss this in class, but you can often lower-bound the runtime of any algorithm to solve a problem by determining how long it would take simply to read the input to the problem.

By lower-bounding the problem, we have a "goal" to shoot for in finding an efficient algorithm. If we upper-bound the worst-case runtime of some algorithm to be the same as the lower-bound on the problem, then we know that we have an asymptotically optimal algorithm.

Looking back at our most useful instance description (the one that talks about "whitespace-separated" preference list lines), we can see that we'll have one number (n), followed be n lines each with n numbers, followed by another n lines of n numbers each. That's $n + n^2 + n^2 \in \Omega(n^2)$.

(Our analysis also gives an O bound, but it's the Ω bound we care about, since the purpose is to lower-bound runtime of solutions.)

So any algorithm that even reads the input will take $\Omega(n^2)$ time.

7 Promising Approach

Unless brute force is good enough, describe—in as much detail as you can—an approach that looks promising.

SOLUTION: You may have thought of lots of ideas. (E.g., earlier I sketched a Borda count-based approach that uses maximum matching. We've noticed that if a man and a woman both most-prefer each other, we must pair them; that might form the kernel of some kind of algorithm. Etc.)

I won't go into one here. Instead, I refer you to the textbook's description of the rather awesome and Nobel prize-winning Gale-Shapley algorithm.

Keen note about Gale-Shapley: It runs in $O(n^2)$ time. That means it matches our lower bound on the problem's runtime and so is asymptotically optimal!

8 Challenge Your Approach

1. **Carefully** run your algorithm on your instances above. (Don't skip steps or make assumptions; you're debugging!) Analyse its correctness and performance on these instances:

   SOLUTION: For fun, we'll use G-S with **women** proposing.

   G-S correctly terminates immediately on any $n = 0$ example with an empty set of marriages. With $n = 1$, the one woman proposes to the one man, who must accept, and the algorithm correctly terminates with them married.

   Going back to our other two examples:

   (a) Example #1:

   ```
   w1: m2 m1 m3        m1: w3 w1 w2
   w2: m1 m2 m3        m2: w3 w2 w1
   w3: m2 m1 m3        m3: w2 w1 w3
   ```

   G-S doesn't specify what order the women propose. We'll work from top to bottom:

   i. w_1 proposes to m_2, who accepts. $E = \{(m_2, w_1)\}$

   ii. w_2 proposes to m_1, who accepts. $E = \{(m_2, w_1), (m_1, w_2)\}$

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.

iii. w_3 proposes to m_2, who prefers w_3 to w_1. m_2 breaks his engagement with w_1 and accepts w_3's proposal. $E = \{(m_2, w_3), (m_1, w_2)\}$

iv. w_1 proposes to m_1 (2nd on her list), who prefers w_1 to w_2. m_1 breaks his engagement with w_2 and accepts w_1's proposal. $E = \{(m_2, w_3), (m_1, w_1)\}$

v. w_2 proposes to m_2, who prefers w_3 to w_2 and so declines the proposal. $E = \{(m_2, w_3), (m_1, w_1)\}$

vi. w_2 proposes to m_3 (last on her list!), who accepts. $E = \{(m_2, w_3), (m_1, w_1), (m_3, w_2)\}$

vii. The algorithm terminates with the correct solution $S = \{(m_2, w_3), (m_1, w_1), (m_3, w_2)\}$.

(b) Example #2:

```
w1: m1 m2          m1: w1 w2
w2: m1 m2          m2: w2 w1
```

We'll again use G-S with women proposing, working top to bottom.

i. w_1 proposes to m_1, who accepts. $E = \{(m_1, w_1)\}$

ii. w_2 proposes to m_1, who declines (prefers w_1 to w_2). $E = \{(m_1, w_1)\}$

iii. w_2 proposes to m_2, who accepts. $E = \{(m_1, w_1), (m_2, w_2)\}$

iv. The algorithm terminates with the correct solution $S = \{(m_1, w_1), (m_2, w_2)\}$.

2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

SOLUTION: Skipping this, since we've already seen a proof of correctness for G-S!

9 Repeat!

Hopefully, we've already bounced back and forth between these steps in today's worksheet! You usually *will* have to. Especially repeat the steps where you generate instances and challenge your approach(es).

SOLUTION: We bounced back and forth quite a bit, even in this carefully crafted solution.

P.S. No solutions to challenge problems, but feel free to talk to us about them!

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
For license purposes, the author is the University of British Columbia.