CPSC 320: Steps in Algorithm Design and Analysis Solutions[*]

In this worksheet, you'll practice five useful steps for designing and analyzing algorithms, starting from a possibly vague problem statement. These steps will be useful throughout the class. They could also be useful when you find yourself thinking on your feet in an interview situation. And hopefully they will serve you well in your work post-graduation too!

We'll use the **Stable Matching Problem (SMP)** as our working example. Following the historical literature, the text formulates the problem in terms of marriages between men and women. We'll avoid the gender binaries inherent in that literature and use employers and job applicants instead. Imagine for example the task faced by UBC's co-op office each semester, which seeks to match hundreds of student applicants to employer internships. To keep the problem as simple as possible for now, assume that every applicant has a full ranking of employers and vice versa (no ties).

Step 1: Build intuition through examples.

1. **Write down small and trivial instances of the problem.**
   We'll use the words "instance" and "inputs" interchangeably.

   SOLUTION: Here's an instance with three employers and three applicants, with preferences listed in order from most to least preferred.

   ```
   e1: a2 a1 a3        a1: e3 e1 e2
   e2: a1 a2 a3        a2: e3 e2 e1
   e3: a2 a1 a3        a3: e2 e1 e3
   ```

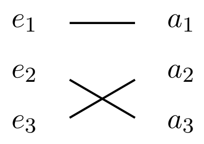
   And here's one with two employers and two applicants:

   ```
   e1: a1 a2           a1: e1 e2
   e2: a1 a2           a2: e2 e1
   ```

   Neither of these are trivial, since there is more than one way to match employers and applicants. It's tempting to say that the smallest possible instance has one employer and one applicant. Can we could go smaller? Degenerate cases are often helpful as base cases in algorithm design and analysis; here the degenerate case has zero employers and zero applicants.

2. **Write down potential solutions for your instances.**
   Are some solutions better than others? How so?

   SOLUTION: One potential solution for our instance of size three is the set $M = \{ (e_1, a_1), (e_2, a_2), (e_3, a_3) \}$. An alternative solution is the set $M' = \{ (e_1, a_1), (e_2, a_3), (e_3, a_2) \}$, drawn below. This one seems better than the first, since both e_3 and a_2 rank each other first and are matched with each other. Admittedly, e_2 gets a lower-ranked applicant in M' than in M.



[*]Copyright Notice: UBC retains the rights to this document. You may not distribute this document without permission.

Step 2: Develop a formal problem specification

1. **Develop notation for describing a problem instance.** What quantities or data (such as numbers, sets, lists, etc.) matter? Give them short, usable names. Think of these as input parameters to the algorithm code. Use your earlier examples to illustrate your notation.

   SOLUTION: You may have come up with different names and quantities, but here are some useful ones.

   - $n \geq 0$, the number of employers and the number of applicants.
   - E, the set of employers $\{e_1, e_2, \ldots, e_n\}$ (so, $n = |E|$)
   - A, the set of applicants $\{a_1, a_2, \ldots, a_n\}$ (again, $n = |A|$)
   - $P_E[i]$, a preference list for each employer e_i. This is a permutation of $[1..n]$.
   - We could use P_E to denote the list of all employer's preference lists (so P_E is a list of lists).
   - $P_A[j]$, a preference list for each applicant a_j, also a permutation of $[1..n]$.
   - We could use P_A to denote the list of all applicant's preference lists.

   The quantity n and lists of lists P_E and P_A fully describe a problem instance:

   ```
   ▷ $n \geq 0$ is the number of employers and also the number of applicants
   ▷ $P_E$ is the collection of complete preference lists ($>_e$) of the employers
   ▷ $P_A$ is the collection of complete preference lists ($>_a$) of the applicants

   function SMP-Algorithm($n, P_E, P_A$)
   ▷ return a stable matching $M$ for the stable matching instance $(n, P_E, P_A)$

   ...
   ```

2. **Develop notation for describing a potential solution.**
   Use your earlier examples to illustrate your notation.

   SOLUTION: A potential solution (or just "solution") for SMP is a set M of employer-applicant pairs, where each employer and each applicant appears in exactly one pair. We call such a set a *perfect matching*. We use (e_i, a_j) to indicate that employer e_i and applicant a_j are matched.

3. **Describe what you think makes a solution *good*.**

   SOLUTION: With respect to a given matching M, an *instability* is a pair $(e, a') \in E \times A$ such that $(e, a) \in M$, $(e', a') \in M$, but e prefers a' to a and a' prefers e to e'. A solution is good if it contains no instabilities, in which case we say that the solution is *stable*. That is, a stable solution is "self-enforcing" in the sense that no employer and applicant who aren't matched will decide to break the matching.

   We'll write $a' >_e a$ to mean "e prefers a' to a" and similarly $e' >_a e$ to mean "a prefers e' to e".

Step 3: Identify similar problems. What are the similarities?

SOLUTION: We haven't seen many problems and algorithms yet, but as the course goes on, we'll have more to draw on here. If you've worked with bipartite graphs and matching problems, anything associated with them seems promising, especially maximum matching. This might also feel a bit like an election or auction, which takes us toward game theory. There are many other possibilities. (The point isn't to be "right"; it's to have potential tools on hand. As you collect more tools, you'll start to judge which are more promising and which less.)

Step 4: Evaluate simple algorithmic approaches, such as brute force.

1. **Design a brute force algorithm for the SMP problem.**
   Given as input a problem instance, a "brute force" algorithm enumerates all potential solutions, and checks each to see if it is good. If a good solution is found, the algorithm outputs this solution, and otherwise reports that there is no good solution. **Flesh out the details: what is a problem instance, and potential solution in this case? How would you test if a potential solution is a good solution?**

   SOLUTION: A brute force algorithm for SMP enumerates all perfect matchings M of employers and applicants. If a stable matching M is found, the algorithm stops and M is returned. Otherwise there is no stable matching. Here is the pseudocode:

```
function SMP-Brute-Force($n$, $P_E$, $P_A$)
    ▷ $n \geq 1$ is the number of employers and also the number of applicants
    ▷ $P_E$ is the collection of complete preference lists ($>_e$) of the employers
    ▷ $P_A$ is the collection of complete preference lists ($>_a$) of the applicants

    for each perfect matching $M$ (i.e., potential solution) do
        if $M$ is stable (i.e., a good solution) then
            return $M$
    return "no stable matching"
```

   Next, we flesh out our algorithm to check if a potential solution is a good solution, i.e., if a perfect matching is stable. This algorithm enumerates employer-applicant pairs (e, a') and checks that (1) they are *not* matched and (2) they'd rather be with each other than with their matches.

```
function IsStable($M$)
    ▷ $M$ is a perfect matching
    ▷ return true if $M$ is a stable matching for instance $(n, P_E, P_A)$, and false otherwise

    for all $e \in E$ do
        for all $a' \in A$ do
            if $(e, a') \notin M$ then
                find $a$ such that $(e, a) \in M$
                find $e'$ such that $(e', a') \in M$
                if $e >_{a'} e'$ and $a' >_e a$ then
                    return false
    return true
```

2. **Analyze the worst case running time of brute force.**

   - **Bound the running time, say $t(n)$, of your procedure to test if a potential solution is a good solution. Assume that the input M is represented as a list of pairs (e, a) such that e is matched with a, where $1 \leq e, a \leq n$.**

     SOLUTION: Traversing a list of matchings to implement the steps in the **if** statement takes $\Theta(n)$ time in the worst case, which is not great. We can do better by initially creating new data structures, before executing the **for** loop.

     - In $O(n)$ time we can build an array Employer-Matches[1..n], whose eth entry is the applicant matched with e. We can use this to determine in $O(1)$ time which applicant a is matched with a given employer e. This enables us to check the condition of the **if** statement, and execute the first **find** statement, in $O(1)$ time.
     - Similarly, in $O(n)$ time we can build an array, Applicant-Matches[1..n] whose ath entry is employer matched with a. This enables us to find which applicant is matched with a given employer e in $O(1)$ time.
     - We can check the condition $a' >_e a$ in $O(1)$ time if we use a 2D matrix, rather than preference lists, to represent employer rankings. Let $Rank_E[e][a]$ store the rank of applicant a in e's list. Then to check if $a' >_e a$ in $O(1)$ time, we can check whether $Rank_E[e][a] < Rank_E[e][a']$.
     - Similarly,we can check the condition $e >_{a'} e'$ in $O(1)$ time if we use a ranking matrix, rather than preference lists, to represent applicant rankings.

     So we pay an initial cost of $O(n)$ time to build the matching arrays, and $O(n^2)$ time to build the ranking matrices, after which the body of the inner loop runs in $O(1)$ time.

     There are n employers and n applicants, so n^2 iterations of the nested **for** loops. So the overall time is $t(n) = O(n + n^2) = O(n^2)$.

- **How many potential solutions are there?** Each potential solution is a perfect matching. Imagine that the n applicants are arranged in some order. How many different ways can we arrange (permute) the n employers next to them?

  SOLUTION: There are n applicants that can be lined up with the first employer. Once we've chosen the first applicant, there are $n-1$ to line up next to the second. Then, $n-2$ next to the third, and so on. Overall, then, that's $n \times n-1 \times n-2 \times \ldots \times 2 \times 1 = n!$. There are $n!$ perfect matchings, or potential solutions. That's already super-exponential! In the worst case, since the order in which solutions are enumerated is not specified, there could be $n!$ iterations of the for loop, e.g., when there is exactly one stable matching. (Can you find an instance of size n with exactly one stable matching?)

- **Putting the previous two parts together, what can you say about the overall worst-case running time of brute force? Assume that the time needed to generate each potential solution is less than that needed to check if a potential solution is a good solution.**

  SOLUTION: In the worst case, all $n!$ permutations of $[1..n]$ are generated. The time to generate each one, and test if it is stable, is $O(n^2)$. So the overall worst case running time is $O(n!t(n)) = O(n!n^2)$. That's horrendous. It won't do for even quite modest values of n.

A lower bound for Stable Matching. 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.

Step 5: Design a better algorithm.

1. **Brainstorm some ideas, then sketch out an algorithm.**
   Carefully try out your algorithm on your examples, and design instances that challenge the correctness of your approach.

   SOLUTION: You may have lots of ideas. For example, you might have noticed that if a employer and a applicant both most-prefer each other, we must match them; that might form the kernel of some kind of algorithm. For our algorithm, we'll work with the Gale-Shapley algorithm, also in textbook.

   Here, we'll work through G-S with **applicants** considering employers. G-S correctly terminates immediately on any $n = 0$ example with an empty set of matchings. With $n = 1$, the one applicant considers one employer, who accepts the match, and the algorithm correctly terminates.

   Going back to our first example:

   ```
   a1: e2 e1 e3        e1: a3 a1 a2
   a2: e1 e2 e3        e2: a3 a2 a1
   a3: e2 e1 e3        e3: a2 a1 a3
   ```

   G-S doesn't specify what order the applicants are chosen. We'll work from top to bottom:

   (a) a_1 considers e_2, who accepts. $M = \{\ (e_2, a_1)\ \}$
   (b) a_2 considers e_1, who accepts. $M = \{\ (e_2, a_1),\ (e_1, a_2)\ \}$
   (c) a_3 considers e_2. e_2 rejects a_1 and accepts a_3's offer. $M = \{\ (e_2, a_3),\ e_1{:}a_2\ \}$
   (d) a_1 considers e_1 (2nd on its list). e_1 rejects a_2 and accepts a_1's offer. $M = \{\ (e_2, a_3),\ (e_1, a_1)\ \}$
   (e) a_2 considers e_2, who prefers a_3 to a_2 and so rejects the offer. $M = \{\ (e_2, a_3),\ (e_1, a_1)\ \}$
   (f) a_2 considers e_3 (last on its list!), who accepts. $M = \{\ (e_2, a_3),\ (e_1, a_1),\ (e_3, a_2)\ \}$
   (g) The algorithm terminates with the correct solution $M = \{\ (e_2, a_3),\ (e_1, a_1),\ (e_3, a_2)\ \}$.

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).