CS41 Lab 2: Brute Force and Reductions

September 7, 2022

In typical labs this semester, you'll be working on a number of problems in groups of 3-4 students. You will not be handing in solutions; the primary purpose of these labs is to have a low-pressure space to discuss algorithm design, and to gain experience collaborating with others on algorithm design and analysis. However, it will be common to have some overlap between lab exercises and homework sets.
For Lab 2, you will explore the first two algorithm design techniques of the semester: **Brute Force**, and **Reductions**.

Brute Force. Brute Force is an algorithm design technique that works by *enumerating all possible solutions* to a problem, checking to see if each possible solution is an actual solution. For example, a brute-force sorting algortihm might enumerate over all possible ways of ordering n elements, and for each such way checking to see if the List is sorted. Brute force algorithms tend to be *simple* and *slow*, but sometimes they're the fastest known solutions to a problem.

Reductions. In computer science, a **reduction** is a way of solving one problem using another. Imagine having an algorithm for problem B, and using that algorithm as a subroutine in an algorithm that solves problem A. We say that "problem A reduces to problem B" or that we have a reduction *from* problem A *to* problem B. Reductions are a very deep algorithmic technique that can be used to make connections between problems that might initially look very very different.

1. **Brute Force Stable Matching**. Design and analyze a Brute Force algorithm that solves Stable Matching. Argue at a high level that your algorithm works. What is the runtime of your algorithm?

2. **Sorting to Half-Sorting**. In the HALF-SORT problem, you're given an array of n integers and must return an array that has the first $\lceil n/2 \rceil$ integers in sorted order. For example, if your array is $A = [5, 9, 1, 2, 6, 3]$, then a valid output of HALF-SORT(A) might be $[1, 2, 3, 9, 5, 6]$ since $1, 2, 3$ are the least elements of A.

    - Reduce the sorting problem to HALF-SORT. i.e., imagine you have an algorithm $\mathcal{A}$ for HALF-SORT, and use it to design a sorting algorithm.
    - Reduce HALF-SORT to the sorting problem. i.e., imagine you have a sorting algorithm $\mathcal{B}$, and use it to design an algorithm for HALF-SORT.
    - Now, suppose that your friend claims to have an algorithm for HALF-SORT that runs in $10n$ time in the worst case. What is the runtime of your sorting algorithm? Is $10n$ a reasonable running time for HALF-SORT?

3. **Hipster Coffee Tours.** A group of n Portland hipsters $H = \{h_1, \ldots, h_n\}$ are touring a set of local coffee shops $C = \{c_1, \ldots, c_n\}$ over the course of $m \geq n$ days. Each hipster h_j has an itinerary where he/she decides to visit one coffee shop per day (or maybe take a day off if $m > n$). However, hipsters are fiercely independent and prefer not to share coffee shops with other hipsters. Furthermore, each hipster is looking for a favorite coffee shop to call his or her own. Each hipster h would like to choose a particular day d_h and stay at his/her current

coffee shop c_h for the remaining $m - d_h$ days of the tour. Of course, this means that no other hipsters can visit c_h after day d_h, since hipsters don't like sharing coffee shops.

Show that no matter what the hipsters' itineraries are, it is possible to assign each hipster h a unique coffee shop c_h, such that when h arrives at c_h according to the itinerary for h, all other hipsters h' have either stopped touring coffee shops themselves, or h' will not visit c_h after h arrives at c_h. Describe an algorithm to find this *matching*.

4. **The Wedding Planner Problem.** Imagine you are a wedding planner. Among other tasks, you must help your customers with their guest lists. Couples come to you with lists of people they might invite to their wedding. They also come with demands – Alice and Bob need to be invited, but if Bob is invited, do not invite Carol (they have history). When Carol, Dave, and Eve get together, they only talk about Justin Bieber (their favorite musician), so don't invite all three. However, any two of them is ok. Call these conditions *constraints*.

   People at weddings are **demanding**. Everything needs to be exactly perfect, and they will blame you if even one constraint is not satisfied. You're not even sure if this is possible, and it when it's not, it would be nice to have a convincing argument for the wedding couple.

   Design an algorithm that takes a list of people, and a list of constraints, and outputs YES if there is an invite list that satisfies every constraint. Otherwise, output NO.