Solutions to Combinatorics Review

We'll talk a lot in this course about **brute force** algorithms: this refers to solving a problem in the most straightforward way, without trying to be clever. Often, it means generating all possible solutions and testing all of them until we find a solution that works. Sometimes, this approach will work for us (and when it does, that's great: brute force algorithms tend to be easy to implement precisely because they aren't clever). But sometimes the number of solutions to check is so large that brute force isn't practical.

Basic combinatorics can be useful in determining, without needing to implement and test a brute force algorithm, whether brute force will be a feasible approach for the problem we're trying to solve. "Combinatorics" is really a fancy word for "counting." We can use some tools from combinatorics to count how many possible solutions exist to a particular problem, which we can use to derive an asymptotic bound on the runtime of a brute force approach to the problem.

This isn't intended to be a comprehensive overview of combinatorics. Rather, we just want to provide enough information that you can easily determine running times of some algorithms that you're likely to encounter in this course and in the future.

A few useful formulas

- Number of ways to sample from k items n times, with replacement: k^n
- Number of ways to order n distinct elements: $n!$
- Number of distinct ways to order items in m distinct groups $g_1, g_2, \ldots g_m$, each consisting of n_i identical objects: $\frac{(\sum_{i=1}^{m} n_i)!}{\prod_{i=1}^{m}(n_i!)}$
- Number of combinations of size k taken from n objects: $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$
- Number of permutations of size k taken from n objects: $\frac{n!}{(n-k)!}$

Sampling with replacement

1. Suppose you have an urn that contains a red ball, a green ball, and a blue ball. You pick a ball out of the urn, put it back in the urn, and pick another ball. If you do this three times, you would observe some sequence of the three colours – for example, you could pick the red ball, the blue ball, and the blue ball again. We'll denote the sequence {red, blue, blue} by RBB.

   If you pick a ball n times, how many possible colour sequences could you observe?

   SOLUTION: Let's start with small numbers and try to find a pattern:

   - After picking 1 ball: 3 possible sequences (R, B, or G).
   - After picking 2 balls: we have 3 possible colours for the first ball, followed by 3 possible colours for the second ball, making for $3 \times 3 = 9$ possible sequences (RR, RB, RG, BR, BB, BG, GR, GB, GG).
   - After picking 3 balls: listing all the possibilities is going to get tedious. But we know that there are 9 possible sequences for the first two balls, and then there are 3 possible colours for the third ball, which makes for $9 \times 3 = 27$ possible sequences.

   Now we can spot a pattern: each ball we add multiplies the total number of possible sequences by three, because the new ball can be any of the three colours. More generally: if we have n balls, the number of possible colour sequences is 3^n (because each of the n balls has three possible colours). If instead of 3 colours we had k colours, the number of possible sequences would be k^n.

2. Consider the Boolean satisfiability (SAT) problem: given n TRUE or FALSE variables $x_1, \ldots, x_n$, and a Boolean expression consisting of those variables joined by AND, OR, NOT, and parentheses, is the formula *satisfiable*? That is, can we assign TRUE or FALSE values to each of the n variables in some way that the entire Boolean expression evaluates to TRUE?

   We want to consider a brute force approach to this problem. How many different possible solutions will we have to check?

   SOLUTION: We have n variables, each of which can have value TRUE or FALSE. This is basically identical to our balls-and-urns scenario, except instead of colours we have TRUE or FALSE (and there are two possible values instead of three). This means that the number of solutions we would have to check is 2^n, and any brute force algorithm will have a runtime of **at least** $O(2^n)$ (it will actually be worse than that, because we can't check whether a given solution satisfies the formula in constant time). Is this practical for big problems? Not so much.

Permutations

3. In how many different ways can you rearrange the letters in the word "dermatoglyphics"[1]?

   SOLUTION: We have 15 choices for the first letter in the ordering. Then, once one letter has been chosen to be first, we have 14 choices for the second letter. Similarly, we have 13 choices left for the third letter, and so on, until we get to the last letter, where have only one choice remaining. We can see that, in total, the number of different orderings is

   $$15 \times 14 \times 13 \times \ldots \times 1 = 15!$$

   In general, if we have n unique elements in a set, there are $n!$ possible ways to order them (or **permutations**).

[1] The scientific study of fingerprints. It's also the longest word in the English language with no duplicated letters – along with "uncopyrightable."

4. Consider the Traveling Salesperson Problem: given a set of n cities, what is the shortest tour that visits all the cities (i.e., what ordering of cities results in the least possible distance traveled)?

   We want to consider a brute force approach to this problem. How many different possible solutions will we have to check?

   SOLUTION: The problem is to determine what **ordering** of cities is best. So, a brute force approach would need to check every possible ordering and report the best solution. Since there are n different cities, we will have $n!$ possible permutations to check, by similar reasoning as in the problem above. Again, this is not practical for large problems: for large n, $O(n!)$ is **much** worse than $O(2^n)$.

Permutations with duplicates

5. In how many different ways can you rearrange the letters in the word "Mississippi"?

   An obvious guess here is that, because Mississippi has 11 letters, there should be 11! permutations. But that isn't quite right. To see why, let's look at a pair of these permutations. First, consider the original ordering of the letters:

   MISSISSIPPI

   Now, consider what happens when we look at another of the 11! permutations that we get if we swap the first P and the second P:

   MISSISSIPPI

   See the problem here? The original guess of 11! counts the original word "Mississippi" twice (at least). In fact, for any permutation of the 11 letters, we can come up with an *exactly identical* permutation by switching the order of the P's.

   So, clearly, we need to divide our 11! guess by 2, since that gets rid of the duplicates we get from having the P's in switched order. But we aren't done here! We also have 4 I's, and 4 S's. This means that for any ordering of the letters, our original 11! orderings also includes $4! = 24$ identical orderings that we can obtain by using the 4! different possible orderings of the I's, and 4! identical orderings we can obtain with all the ways to order the S's. So, to get the number of *distinct* orderings of the letters in "MISSISSIPPI", we need to take 11! and divide it by the number of identical orderings we can get by permuting the P's (2!), the I's (4!), and the S's (4!). This means that the number of distinct permutations is

   $$\frac{11!}{2! \cdot 4! \cdot 4!}.$$

   To define this more mathematically: suppose we have m distinct groups $g_1, g_2, \ldots g_m$, each consisting of n_i identical objects. The number of distinct ways to permute these objects is

   $$\frac{(\sum_{i=1}^{m} n_i)!}{\prod_{i=1}^{m}(n_i!)}.$$

   The symbol on the bottom is product notation: it's similar to the sigma notation used for sums, but we multiply the terms together instead of adding them.

   To put our "MISSISSIPPI" example into this notation: our $m = 4$ distinct groups of objects are: the letter M ($n_1 = 1$); letters I ($n_2 = 4$); letters S ($n_3 = 4$); and letters P ($n_4 = 2$).

6. How many different possible solutions do we need to consider in a brute force algorithm for the Resident Hospital Problem (RHP) with n residents and m hospitals each containing n_i slots? (We did also answer this in class; but if you were at all confused by that explanation, this is a good time to think about it a bit more and/or ask for some clarification if you're still having trouble.)

   SOLUTION: As we discussed in class, for our solution to RHP, we care about which residents end up at which hospital, but we **don't** care about which slot they get, as we assume that all slots at a

given hospital are identical. So this is exactly like our Mississippi example above: the letters are like hospital slots, and the **duplicated** letters are like different slots at the **same** hospital.

Having recognized this, we can apply the formula above to the scenario described by RHP and determine that the number of possible solutions is

$$\frac{(\sum_{i=1}^{m} n_i)!}{\prod_{i=1}^{m}(n_i!)} = \frac{n!}{\prod_{i=1}^{m}(n_i!)}.$$