CPSC 320: Intermediate Algorithm Design and Analysis

Tutorial: Week 3

Author: Susanne Bradley

At the time of this week's tutorial, we were approaching the end of our stable matching unit and about to start asymptotic analysis. Topics covered include:

- Combinatorics review, and how this is useful for analyzing brute force algorithms;
- Logarithm rules review;
- A brief overview of reductions.

This writeup also includes a summary sheet on the last page.

1 Combinatorics review

We'll talk a lot in this course about *brute force* algorithms: this is a very general, non-clever way to solve a problem where we generate all possible solutions, and test 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 mathematical 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 review 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.

Sampling with replacement

Suppose you have an urn that contains a red ball, a green ball, and a blue ball (combinatorics deals a lot with balls and urns). 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? 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.

Case study: exponential-time algorithms When we sampled n coloured balls with replacement, we ended up with a number of possible colour sequences that was exponential in n. A lot of problems have brute force algorithms run in exponential time, and it's often because generating the set of possible solutions involves sampling with replacement in some way.

For a well-known problem of this type, 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?

Without knowing anything about whether there might be clever ways to solve this, we can start thinking about a brute force approach. If we wanted to generate all possible solutions, what would that look like? 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.[1]

Permutations

A *permutation* refers to arranging the members of a set into some order. A question: How many permutations are there of the letters in the word "CAT"? In this case, we can count them out by hand and see that there are six: CAT, CTA, ACT, ATC, TCA, TAC. But, let's try to come up with a general formula, so that you'll know what to do if, instead of giving you an easy word like "cat," somebody asks you for the number of permutations of the letters in "dermatoglyphics."[2]

For "dermatoglyphics," we have 15 choices for the first letter in the permutation. 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 permutations 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 permutations.

Case study: factorial-time algorithms We've already encountered a problem where the brute force approach has to search through $n!$ possible solutions, and that's the Stable Marriage Problem. In general, we see size-$n!$ search spaces whenever the set of solutions involves all possible orderings of

[1]Asymptotic analysis lesson 1: exponential runtimes are bad!

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

a set. Another example of this class of problem is the Traveling Salesperson (TSP) problem: given a set of cities, what's the shortest tour that visits all the cities (i.e., what ordering of cities results in the least possible distance traveled?). And, as we discussed in the stable marriage worksheet, this is also not practical for large problems.[3]

Permutations with duplicates

Suppose now that I ask you the number of ways to reorder the letters in "MISSISSIPPI"? After reading the previous subsection, you might think that, because "MISSISSIPPI" has 11 letters, there are 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 I look at another of the 11! permutations that I get when I swap the first P and the second P:

MISSISSIPPI

Clearly, there is a problem with our original guess of 11!, because it counts the original word "MISSISSIPPI" twice (at least). In fact, for any permutation of the 11 letters, I 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 ($n_4 = 2$).

[3] Asymptotic analysis lesson 2: factorial runtimes are even worse!

Case study: factorial-time algorithms II In terms of algorithm search space sizes, this case is less common than the previously discussed cases. But, we did encounter this scenario in our worksheet on the Resident/Hospital Problem. There, the residents at each hospital were like the duplicated letters in "MISSISSIPPI": they weren't all identical, but their *order* within the solution didn't affect the actual solution. Essentially, this case arises when we need to examine different orderings of elements in a set, but the order of elements within certain subgroups (like residents matched to the same hospital) doesn't matter.

The solution space size in these cases is *often* $\Theta(n!)$, but not always. (Can you think of an instance of the Resident/Hospital problem where the number of possible solutions is polynomial in n?)

A caution...

The examples we covered in this subsection dealt almost exclusively with brute force algorithms that run in exponential or factorial time. But, this is *not the case* for all brute force algorithms, even if you need to use combinatorial expressions to determine the size of the solution search space.

Question: Give a Θ-bound of a brute force algorithm to solve the SAT problem in n variables if we assume that **exactly 3 of the variables must be TRUE?** (Assume that you can check whether a particular set of variable assignments satisfies the expression in linear time.) Hint: Recall that the formula to choose a combination of k elements from a set of n (where "combination" means that we sample k items without replacement and the order of the items doesn't matter) is

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!},$$

and try to simplify the expression for this particular case.

2 Logarithm rules review

Most of these are just formulas that you will have learned in high school, which I've included in the summary sheet at the back of this document.

Case study: fun with log bases You all (hopefully) know the binary search algorithm: given an ordered list, we find a target value by checking the middle element of that list, determining whether the target is in the lower or upper half of the list (by checking whether it's smaller or larger than the middle element), and recursively searching the half of the list that contains the target. The array indexing and comparisons take $O(1)$ time, and we have to do this at most $\log_2 n$ times (because that's how many times you can divide a list of size n in half), which gives a total runtime of $O(\log_2 n) = O(\log n)$.

Suppose now that your friend proposes a *trinary* search algorithm: for a sorted array of size n, we check the values at position $n/3$ and $2n/3$. From that we determine whether the target is in the bottom, middle or top third of the list, and recursively search that third of the list.

What's the runtime of this algorithm? At each step, we have two array accesses, and up to two comparisons (is target greater than or less than the value at index $n/3$, and is it greater than or less than the value at index $2n/3$?), which is $O(1)$ work altogether. The greatest number of steps we can take is $\log_3 n$, because that's how many times we can divide the array into thirds. So our total runtime is $O(\log_3 n)$ – but because logarithms in different bases only vary by a constant factor, this is still $O(\log n)$, just like binary search.

Question: Binary search and trinary search both run in $O(\log n)$ time. But I claim that one of them is still about 20% faster than the other. Which one is faster, and why?

3 Reduction introduction

I'll very quickly introduce the concept of a reduction, which is a concept that we'll use a bit at the beginning of the term, and then do to death in the last month or so of classes. (And in case you were wondering: yes, I *do* plan to have a section in a later tutorial document called "Reduction re-introduction.")

Basically, a reduction means solving a problem by converting it to some other problem and then solving that instead. When we tell you to "reduce from problem A to problem B," we mean:

1. Take an instance of problem A and convert it to an instance of problem B. (You must specify how to do this part.)
2. Assume you have a black-box solver that solves problem B. **You do not know anything about this black box and can make no assumptions about how it works.** All you know is that, given a valid input (i.e., an instance of problem B), it will give you a correct output (i.e., the correct/optimal solution to problem B). The black box will solve the instance of problem B that you gave it in step 1.
3. You take the returned solution of problem B, and convert it to the solution to problem A. (You must specify how to do this part.)

A couple of important notes on reductions:

- Unless we state otherwise, your reduction must work for *all* legal instances of problem A.
- We say that a reduction is "correct" (or "optimal") if, for any instance of problem A that you're given in step 1, the solution returned in step 3 is a correct (or optimal) solution to that instance of problem A. A reduction is *incorrect* if there exists an instance of A for which the reduction returns the wrong answer. (And recall that we always assume that the solver for B returns the correct answer, given an instance of problem B.)

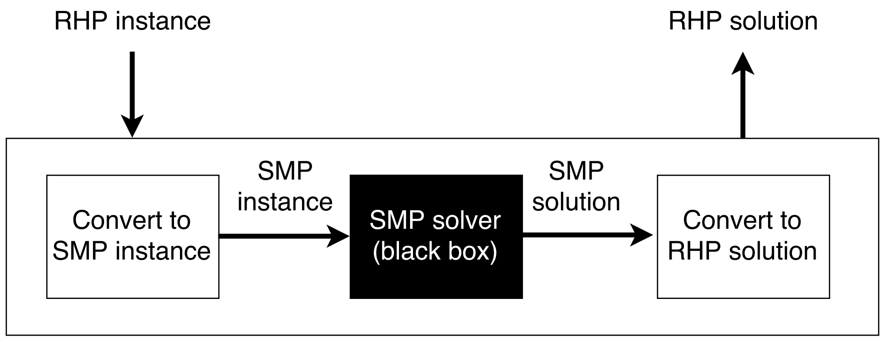


Figure 1: Illustration of the RHP to SMP reduction seen in class. For details of how the steps in the white boxes work, refer to the sample solution of the "Reductions and Resident Matching" handout (which has not yet been released at the time this document was written – so depending on when you read this, you may just have to wait).

Week 3 tutorial cheat sheet

Combinatorics

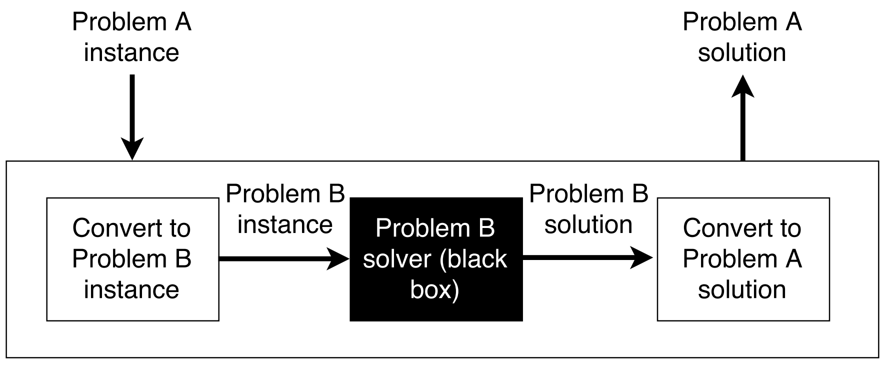
- Number of ways to sample from k items n times, with replacement: k^n
- Number of ways to order n distinct elements: $n!$
- Number of distinct ways to order items in m distinct groups $g_1, g_2, \ldots g_m$, each consisting of n_i identical objects: $\frac{(\sum_{i=1}^{m} n_i)!}{\prod_{i=1}^{m}(n_i!)}$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

Logarithm rules

- $\log(xy) = \log(x) + \log(y)$
- $\log(x/y) = \log(x) - \log(y)$
- $\log(x^y) = y \log(x)$
- $\log_a(a^x) = x$
- $\log_a x = \frac{\log_b x}{\log_b a}$

Reductions

A "reduction from A to B" looks like:


Problem A instance
Problem A solution
Convert to Problem B instance
Problem B instance
Problem B solver (black box)
Problem B solution
Convert to Problem A solution

Your job: describe how the two white boxes work.