CMPS 10 Lecture Notes: Lecture 18 (3-3-2016)

Hey folks. So, once again, Ben was unable to take notes on lecture, this time because he was out of town. However, after speaking with Dimitris, there were two major topics that he felt it important that you know: "Combinations and Permutations" and "Merge Sort." So let's chat about those concepts, and some fundamental ones that build up to them!

FACTORIAL

This should hopefully just be a quick reminder. When you see something that looks like this:

7!

You can read that as "Seven Factorial." I know I have a penchant for exclamation points, but I'll try to be mindful of their use from here on out. Seven Factorial is the mathematical equivalent of this:

7*6*5*4*3*2*1

Which some quick math will tell you evaluates out to 5,040. So you can see it gets pretty big, pretty quickly. Okay, now with that under our belts, we can move on to...

PERMUTATIONS AND COMBINATIONS

First of all, before we worry about any math about computing these guys, let's make sure we're on the same page about what they actually are. These two concepts are very similar to each other, as they both are dealing with possible groupings, so it's easy to get them mixed up with one another. But the basic difference is this:

Permutations CARE about ordering.

Combinations DO NOT care about ordering.

Let's go through these two guys one by one, starting with Permutations.

Permutations

Let's imagine that you live in a world in which you have eight friends. And then let's say that you decide to hold a big foot race in your backyard and invite all your friends to compete. At the end of the race, you want to give your friend that got first place a gold medal, second place gets a silver medal, and third place gets bronze. Everyone else gets nothing (besides the joy of spending an afternoon with friends).

The question then becomes: what are all the different ways that you could distribute your three medals? Note that this is a matter of **PERMUTATIONS**, as ordering matters! Alice getting first, Bob getting second, and Carl getting third is *a very different result* than Carl getting first, Bob getting second, and Alice getting third (at least it is for Carl and Alice).

We'll cover the "official function" to solve permutations in just a second, but for now, the trick to solve Permutation problems can be thought of like this:

Figure out how many "options" there are for the first "choice." Then multiply that to how many "options" there are for the second "choice." And then multiply THAT for how many options there are for the third choice. And keep on doing that for as many choices you have.

So here, our "choices" are our medals – we have 3 of them, and our "options" are our friends – we have 8 total. In theory, any of our friends could get in first place, so there are 8 possible options for first place.

However, once we have that, there are now only 7 possible options for second place (because the person who got in first can't ALSO get in second!). So we multiply 8 and 7, and we get 56). And then there are only 6 possible options

for third place (because our 1st and 2nd place friends are out of the running). So we multiply that to our running total, for 56 * 6 = 336 total possible ways of distributing your medals.

A useful way of thinking of the above: **We had to ORDER 3 people out of 8**.

So now that we have the home-remedy way to solve permutations, let's dive into the "real" function. 8*7*6 *KINDA FEELS* like factorial. But, of course, we know that it isn't exactly factorial, as 8! = 8*7*6*5*4*3*2*1 = 40,320 which is waaaay too much! How can we stop it? Well...

The first thing to notice is that we like the 8*7*6 part, BUT we don't like the 5*4*3*2*1 part. But wait a second! 5*4*3*2*1... isn't that just.. yes, of course! It's 5! So then, what would happen if we calculated 8!/5! ?

Well, you end up with...

$$\frac{8!}{5!} = \frac{8*7*6*5*4*3*2*1}{5*4*3*2*1} = 8*7*6 = 336$$

You can see that 5*4*3*2*1 from the top and bottom of the function cancel each other out! And it leaves you with 8*7*6, which is what we were hoping for in the first place. And remember, we got 5! in the first place because that was how many people we had left after we were done ordering the amount that we cared about. In this example, that means that we had 8 people total, and after we had ordered the three that we cared about, we were left with 5 that we didn't care about. And so this brings us to the actual permutation function:

To Order k Things Out of n Total Things, you use this function:

$$\frac{n!}{(n-k)!}$$

Or, put another way:

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

And voila! That's the magic right there! When computing permutations, that function is all that you need, but hopefully now you have a little bit of an understanding as to WHY you need it! Now, let's move on to Combinations!

Combinations

Now don't forget what Combinations are: **Groups in which ordering DOES NOT matter**. Let's imagine that you still have 8 friends, and that you still have a race. But instead of having three distinct medals (gold, silver, and bronze), you instead just have three blue ribbons, and the top three racers will each get one of these ribbons.

Note that all of a sudden we've entered a world in which ordering DOES NOT matter. Alice, Bob, and Carl each getting a blue ribbon IS THE SAME as Carl, Bob, and Alice getting a blue ribbon. So, note that here, Alice, Bob and Carl is equal to Carl, Bob, and Alice. This can be considered a *redundancy.*

To figure out the total number of redundancies, it's actually just as easy as taking the factorial for the number of things you are dealing with! That is, if we want to know how many ways we can arrange Alice, Bob, and Carl, we note that we have 3 people, so we simply take 3!, or 3*2*1, or 6. And it isn't hard to see that what they are specifically:

Alice, Bob, Carl
Alice, Carl, Bob
Bob, Alice, Carl
Bob Carl, Alice
Carl, Alice, Bob
Carl, Bob, Alice

So there are six different ways that we can arrange these three friends, and when working with combinations, we treat all of them as equivalent to each other, because again, Ordering DOES NOT MATTER with combinations.

And so it turns out that if we want the combinations, we actually **First find the permutations** and then **Divide out the redundancies**. That ends up yielding a function that looks like this:

$$C(n,k) = \frac{P(n,k)}{k!}$$

And that P(n,k) function is just the Permutation function that we first learned! So we can expand that out to get this:

$$C(n,k) = \frac{n!}{(n-k)!} * \frac{1}{k!}$$

Where that left fraction is the Permutation function, and the right fraction is dividing out the redundancies. Usually you'll probably see those two fractions combined like this:

$$C(n,k) = \frac{n!}{(n-k)! * k!}$$

Which is exactly the same thing. And so, to complete our example, if we plug in "8" for n (our total number of friends) and "3" for k (the number of ribbons we have, the size of our group), we can see we get

$$C(8,3) = \frac{8!}{(8-3)! * 3!} = \frac{8*7*6*5*4*3*2*1}{(5)! * 3!} = \frac{8*7*6*5*4*3*2*1}{(5*4*3*2*1)*3*2*1} = \frac{8*7*6}{3*2*1} = \frac{336}{6} = 56$$

And I realize that that might look long and scary, but each step is either just expanding the factorials out to the multiplications, canceling out like terms from the numerator and denominator, or actually carrying out the multiplication. And so we see that we wind up with 56 different groups of three friends, when ordering doesn't matter!

Pretty fun, huh?

Another good reference for this is This Webpage

Permutation and Combination Recap

So, I'd say there are five major take aways from the above notes. If you can do the following five things, you should be good for anything that may appear on a final exam:

1. Knowing when something is a Permutation problem vs. a Combination problem (i.e., does ordering matter or not matter?).

2. Knowing *why* the Permutation function works (i.e., the step by step breakdown we went through to derive the Permutation function).

3. Knowing *why* the Combination function works (i.e., the same as above for the combination function).

4. Given an "n" and a "k", figure out all the permutations (i.e., KNOW the permutation function and how to use it).

5. Given an "n" and a "k", figure out all the combinations (i.e., KNOW the combination function and how to use it).

MERGESORT

So, the professor talked about Mergesort on the previous lecture, so you can check those notes for a first introduction to Mergesort. However, a brief reminder is:

Mergesort(A)
If |A| = 1
then

return the unique element in A
else
split the elements of A into two arrays L and R as evenly as possible
SLH $<--$ Mergesort(L)
RLH $<--$ Mergesort(R)
merge the content of SLH and RLH into a single sorted array and return it

So, going through this line by line...

If |A| = 1 — this is checking the length of the array that was passed in, and specifically it is checking if the length is 1. If it is, then we'll go inside the "then" block, which is...

return the unique element in A – there's only one element in A (we know that because we just confirmed the length of A was 1), so let's just return it! One element is already sorted by default! But what if the length wasn't 1 when we checked? Well, we would have jumped into the else case...

split the elements of A into two arrays L and R as evenly as possible – we have an array of some length greater than 1. Let's split it into two arrays, as evenly as we can, and call them L and R.

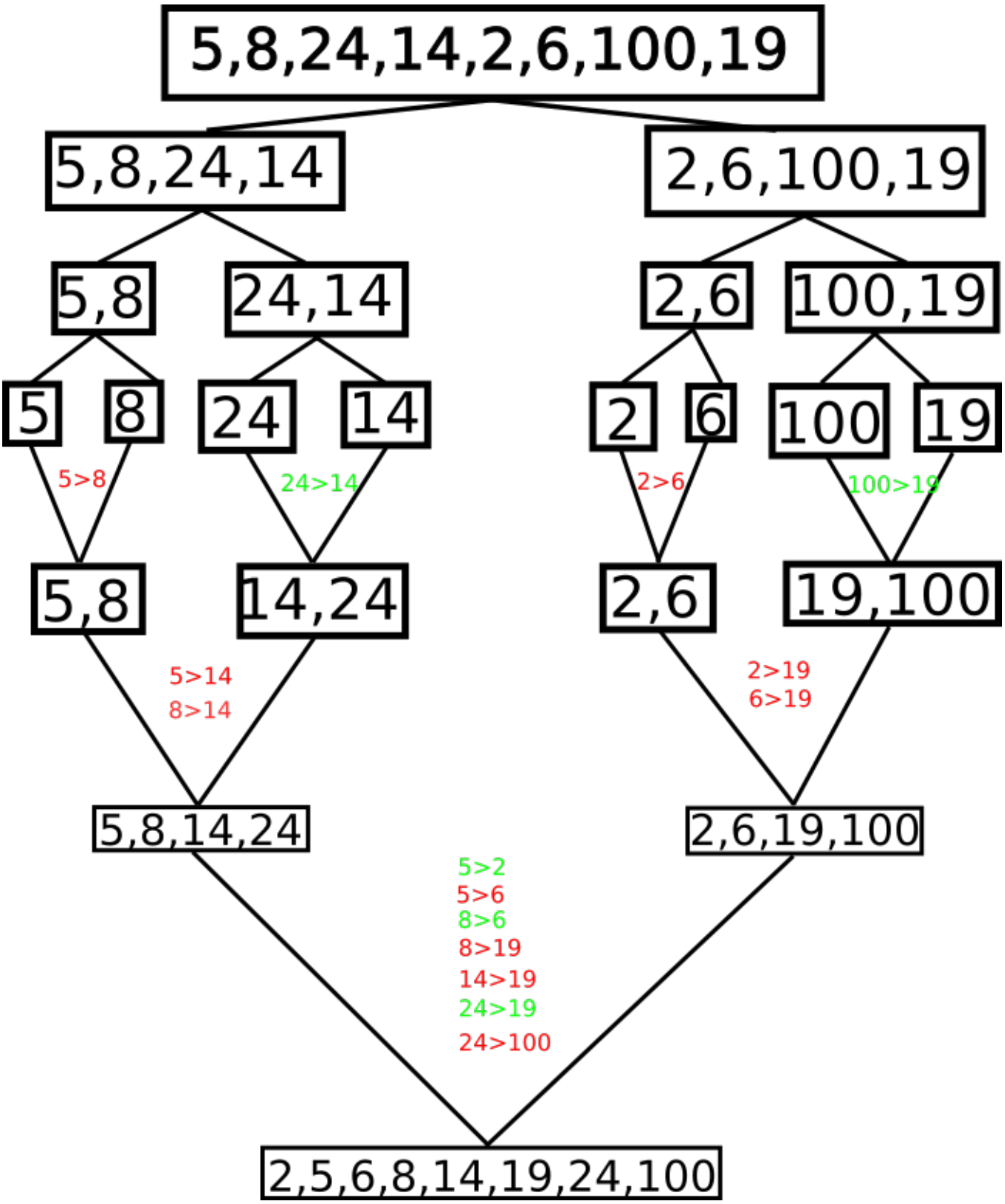
SLH $<--$ Mergesort(L) – NOW that we have our L array, let's USE MERGESORT ON IT (hence recursion! We are using Mergesort INSIDE OF Mergesort!) It'll likely take a while for it to be done, but when it is, we'll store it in a temporary array "SLH" – this will be guaranteed to be sorted!

RLH $<--$ Mergesort(R) – Similarly, we'll use mergesort on our R array too (more recursion! Whoo fun!), and when it finishes, we'll store it in RLH This will also be guaranteed to be sorted!

merge the content of SLH and RLH into a single sorted array and return it – And then, finally, when we have SLH and RLH (which are both sorted individually), we 'merge' them together by going through these two arrays and one by adding, adding the smallest remaining element to a new array, and then returning it.

Basically, you take an array of numbers, you split it into two arrays, and then split both of those into two arrays, and then THOSE into two arrays, forever and ever UNTIL you start getting arrays of length 1. Then you slowly start piecing them back together. At first the piecing process is really easy, since the arrays are just of size one, so it's easy to tell which number should go first in the combined array (i.e., it is just the smaller of the two). As you begin to merge longer arrays together, you need to do more checks, but because the individual arrays are guaranteed to already be sorted by that point, you'll know which two numbers precisely are the ones that need to be compared against each other at any given moment. Once "all" of the numbers of one of the arrays have been "used" (i.e., added to the new merged array), then all of the remaining elements of the other array can be safely appended to the end of the merged array (as again, we know they are already sorted relative to each other).

I think this might be best illustrated with an example. On the next page, see that we have a starting array, it progressively gets broken down into smaller pieces until we have arrays of length 1, and then we slowly piece them back together. The colored numbers are the comparisons that get made. Green means that the number from the Right Hand Array is smaller and should be added to the merged array first. Red means the number from the Left Hand Array is smaller and should be added first.


5,8,24,14,2,6,100,19
5,8,24,14
2,6,100,19
5,8
24,14
2,6
100,19
5
8
24
14
2
6
100
19
5>8
24>14
2>6
100>19
5,8
14,24
2,6
19,100
5>14
8>14
2>19
6>19
5,8,14,24
2,6,19,100
5>2
5>6
8>6
8>19
14>19
24>19
24>100
2,5,6,8,14,19,24,100