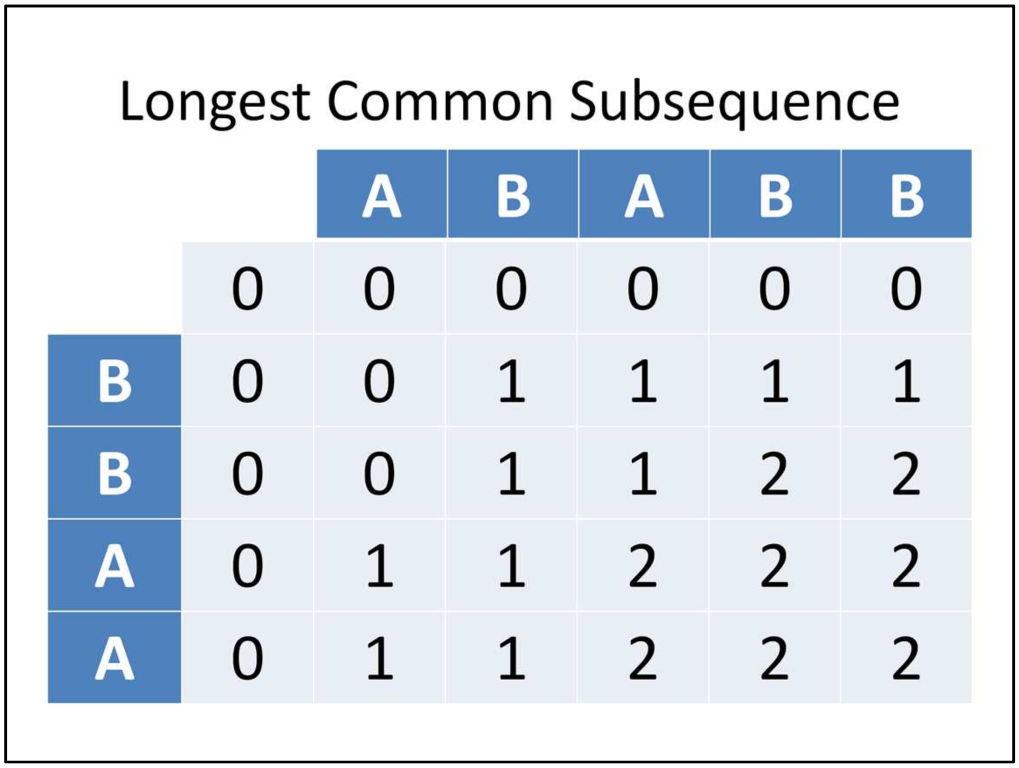
CS161L: Implementation of Algorithms

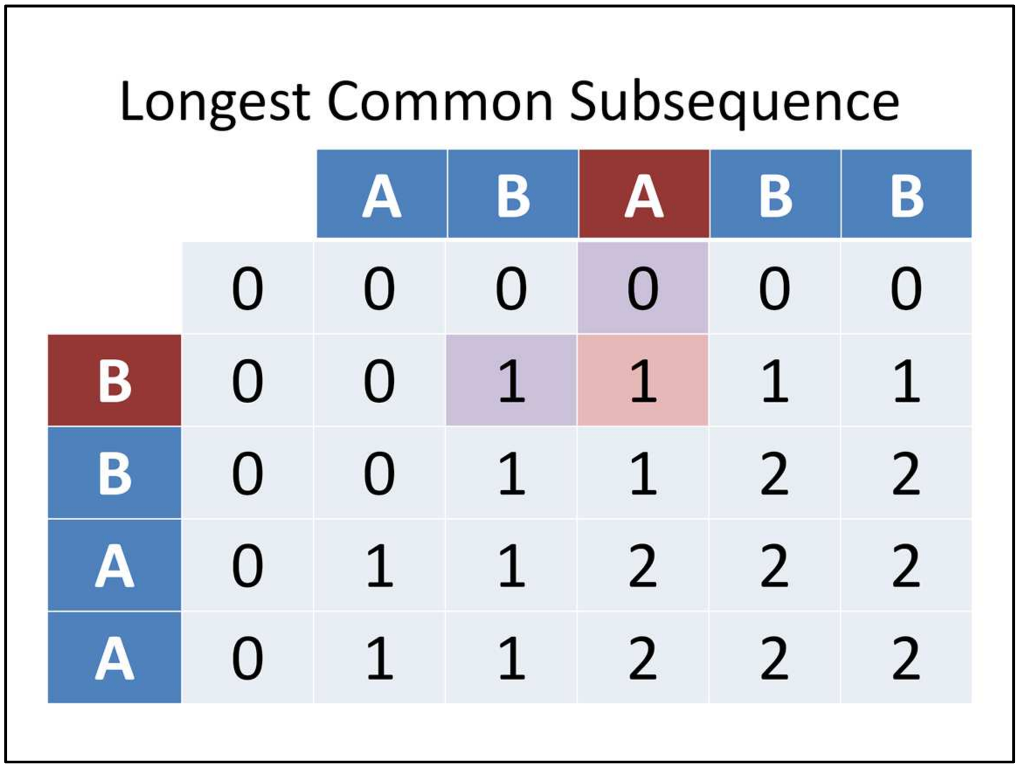
Friday, May 9, 2014



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

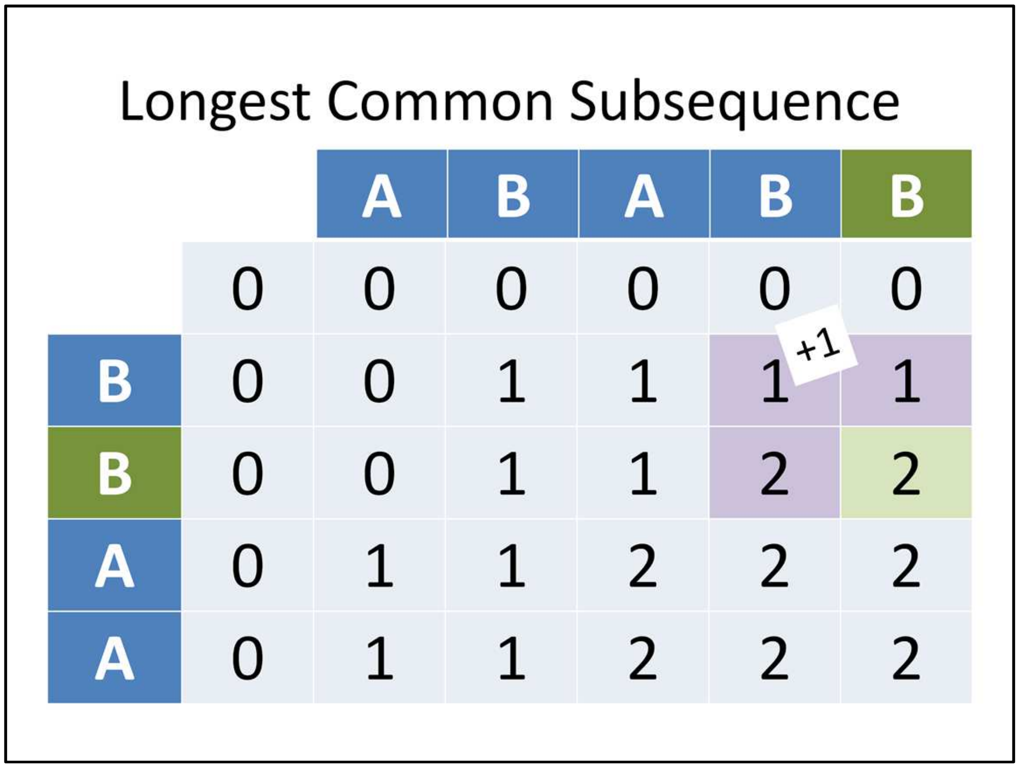
The first problem we're covering today is longest common subsequence, or LCS. This was covered in detail in lecture on Wednesday, but as you might have noticed, it's relevant to the programming project in 161, so I'll review it here. We're given two strings, and we want to find the longest subsequence of characters that they have in common. For example, the subsequence "AA" exists in both of these strings, so it's a common subsequence. The main observation we use is that we can focus on what we do to the last character in each string, knowing that we should take the optimum for the remainder of the two strings. In particular, we can do one of three things: we can match the last two characters to each other, ignore the last character in the first string, or ignore the last character in the second string. If we solve this problem for all prefixes of the two strings, in increasing size of the prefixes, then we'll always have the answers we depend on to compute our current answer. So what we do, given two strings of length m and n, is we make a 0-indexed 2D array where the entry at (i, j) corresponds to the length of the LCS of the first i characters of the first string and the first j characters of the second string.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

Then we walk through the table from top to bottom, left to right, filling in the entries. The top and left entries are all 0s. As for the other entries, if the characters corresponding to the entry don't match, then we just take the bigger of the values immediately to the left of or above it.

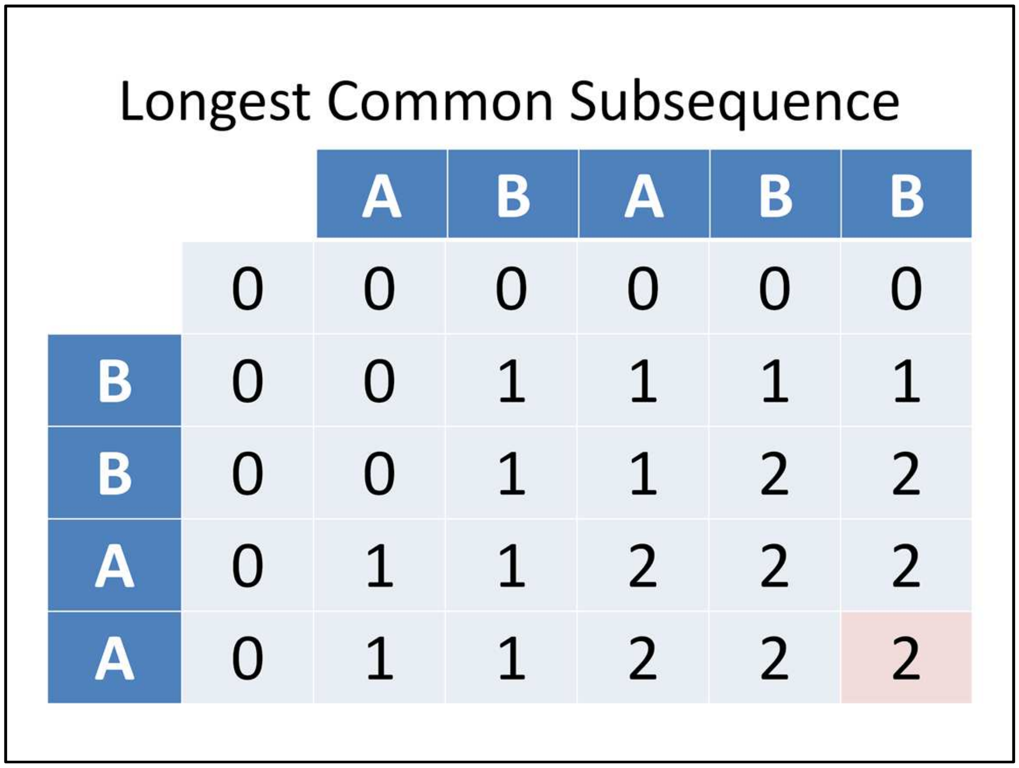


Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

+1

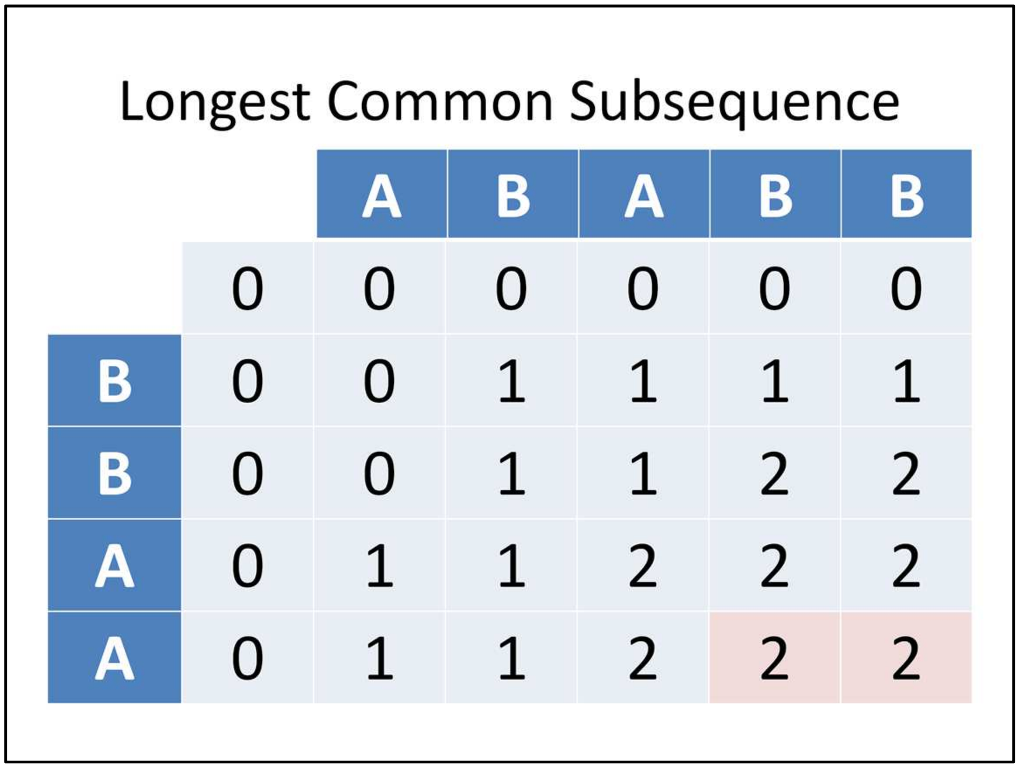
On the other hand, if the characters do match, then in addition to those two entries, we can also look at the entry diagonally up and left of us and add 1 to it, and see how it fares against the other two.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

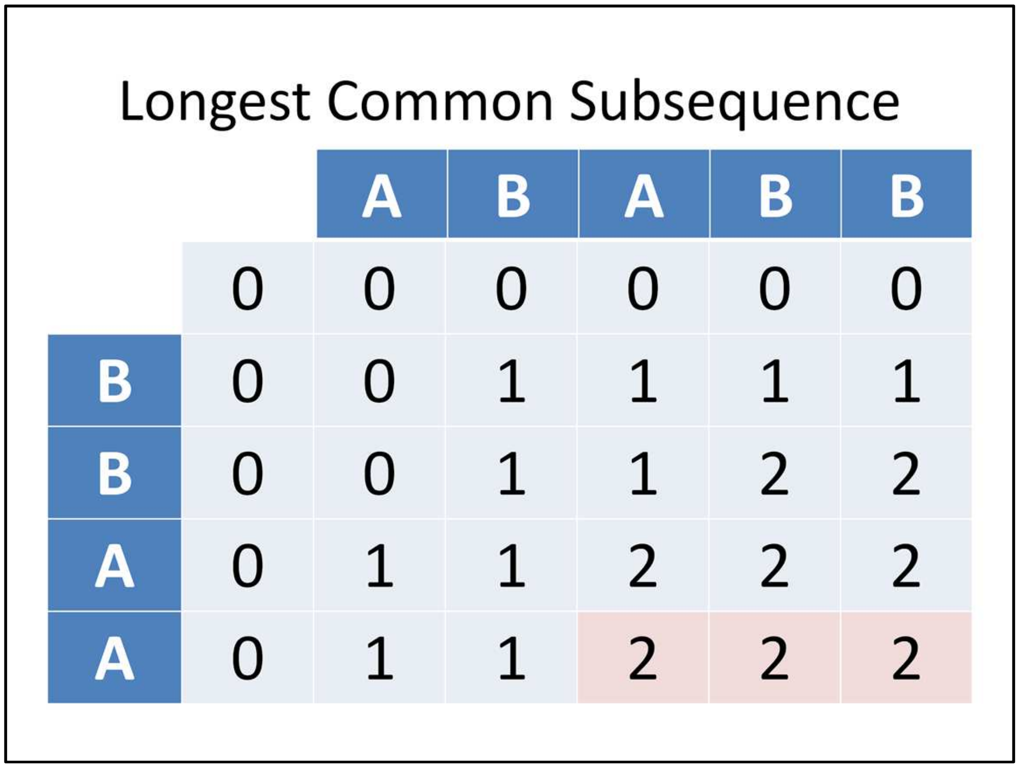
Now the problem as stated for this week only asks you to give the length of the LCS, so all you need to do is return the number in the bottom right corner. However, it's also worth understanding how to retrieve not only the LCS itself, but also the path through this table that generates it. Yes, you can do this with a second table that stores a bunch of pointers, but I'm going to show you how to walk through the path just by looking at the entries of the table. You start from the bottom right, and then you move to one of the entries that could have produced the value at your current location. For example, this 2 in the bottom right matches the 2 just to the left of it, so that was one way to produce this value.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

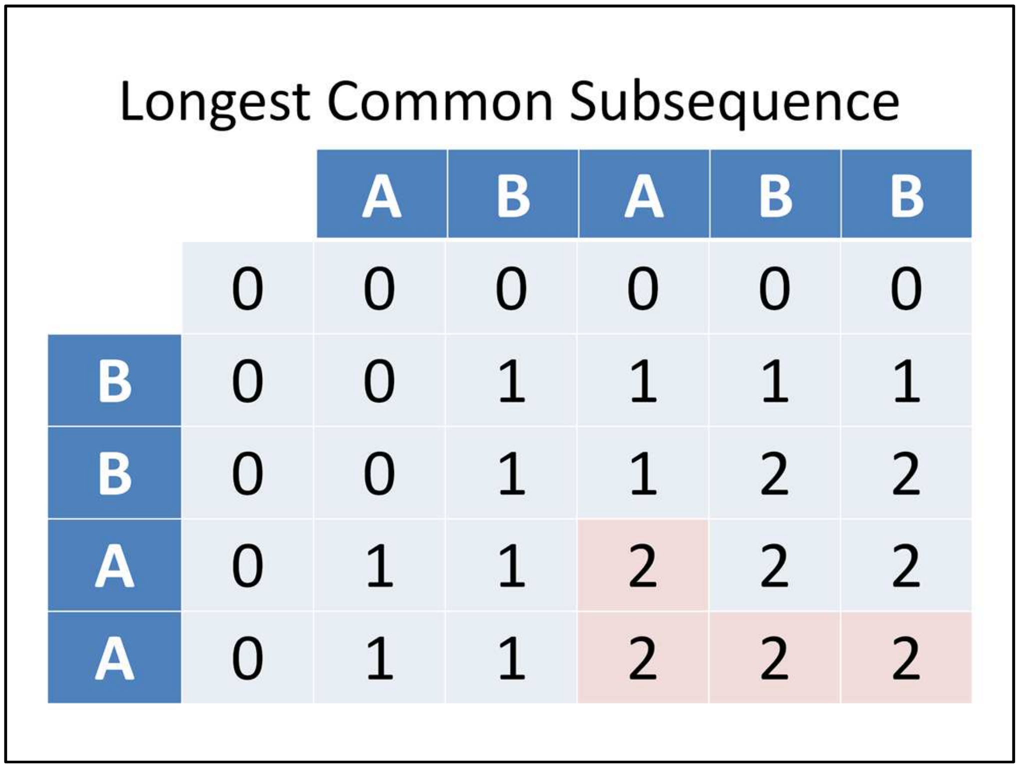
Therefore we can move one to the left. Notice that we also could have gone one up; both of these are valid choices because they just correspond to different ways of generating a longest common subsequence. For now, if we can move both left or up, let's prefer moving left.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

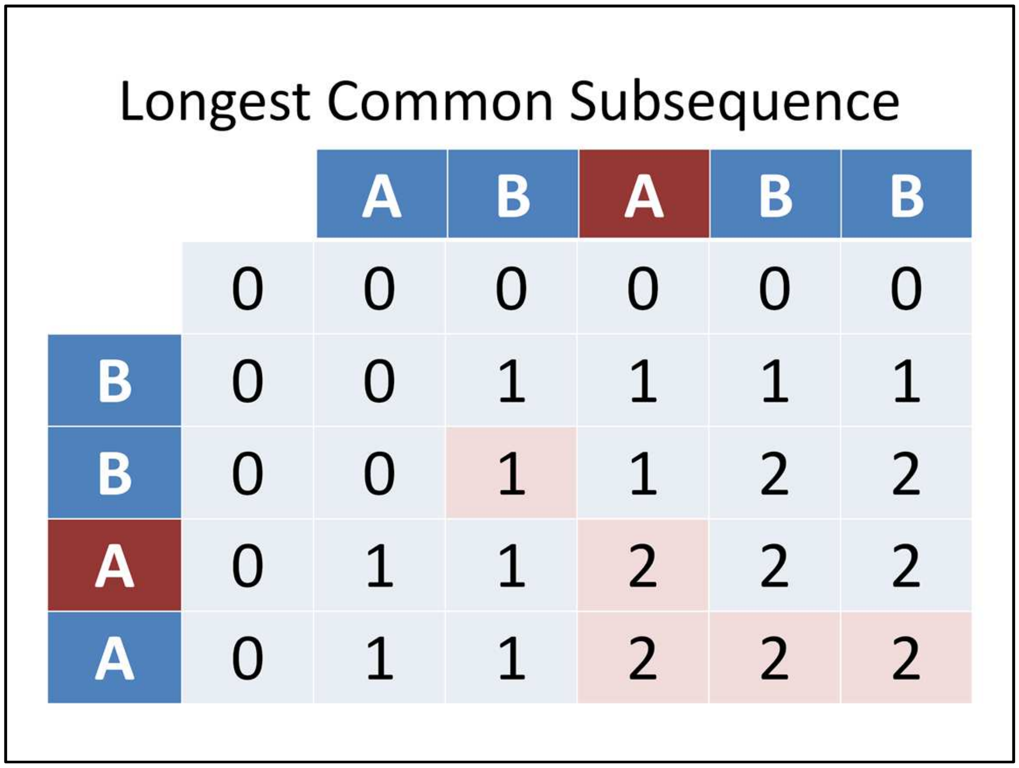
Eventually we'll run out of the ability to move left, but in this case we can still move up.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

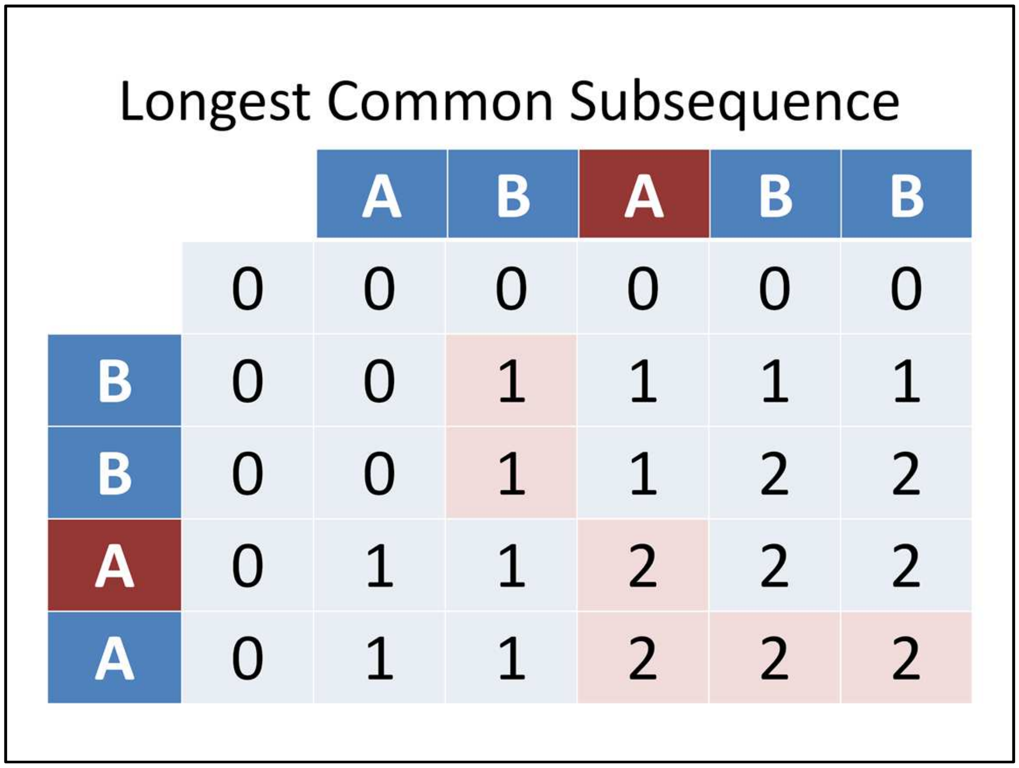
So let's do that. Now notice that we can't move left or up. But this number had to have been produced by some value, and if it wasn't made by the number to the left, or by the number above it, then it had to have been made by the diagonal.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

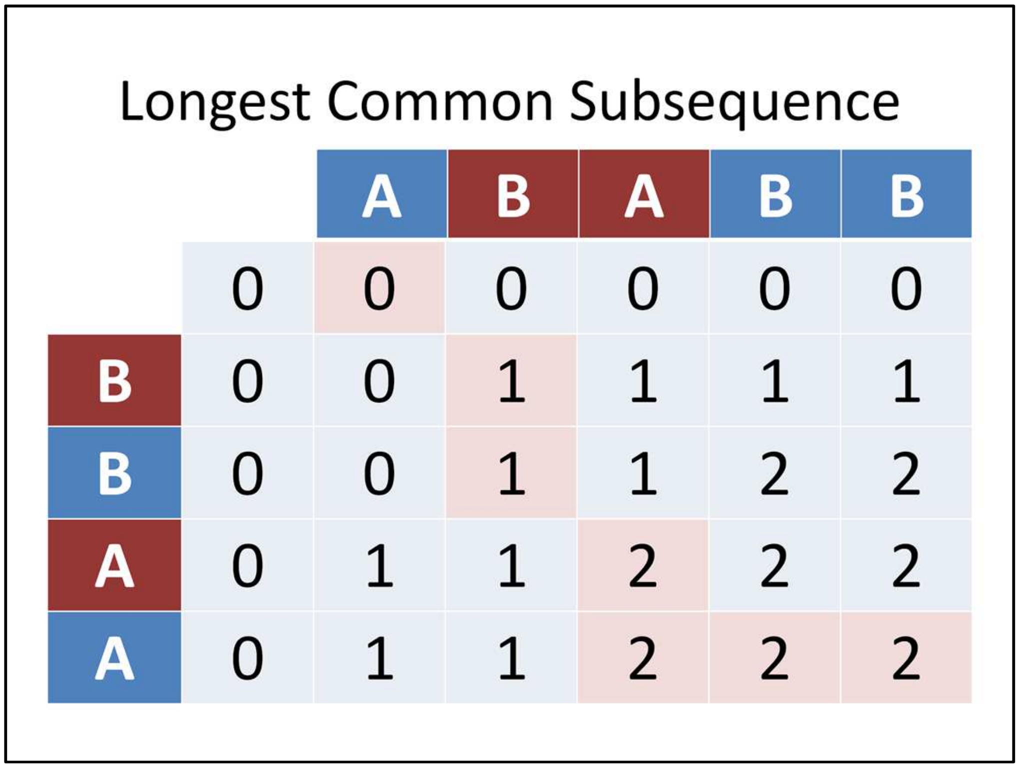
So let's move diagonally up and left. Remember that each diagonal movement corresponds to matching a pair of characters. Also notice that we didn't actually have to check to see whether the characters matched; we used process of elimination instead, saying that since we didn't fill in this table entry from the left or from above, we MUST have filled it in from the diagonal, which means the characters MUST match.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

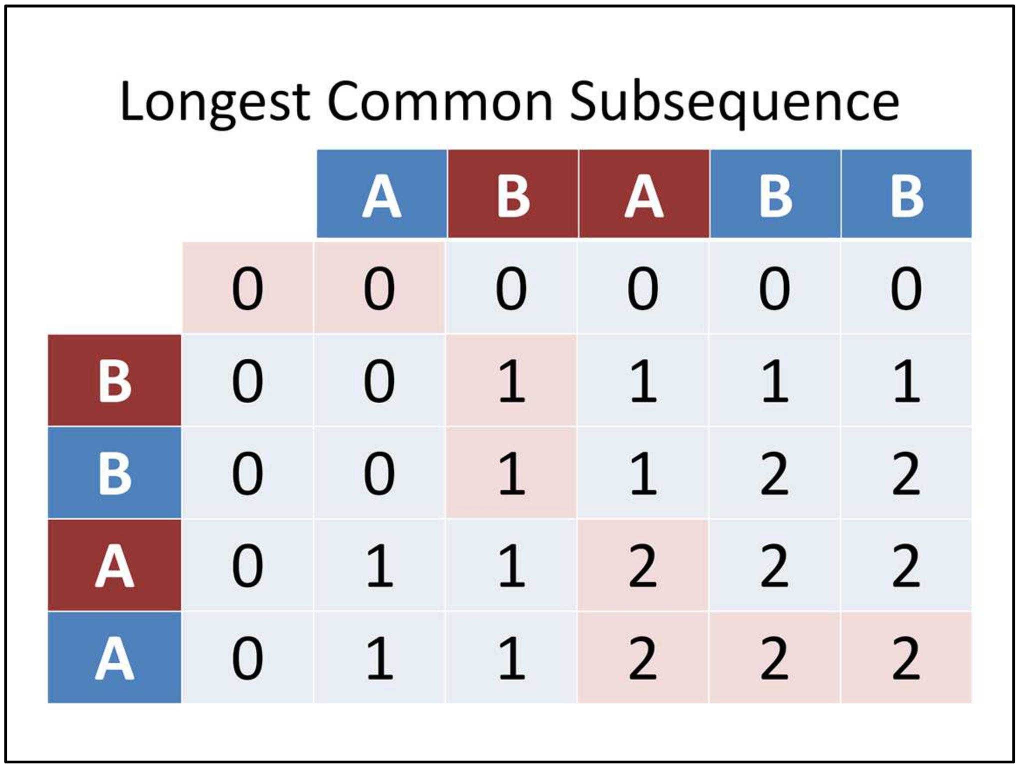
Anyhow, let's continue the process. We can't move left, but we can move up one. Then we can move neither left nor up,



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

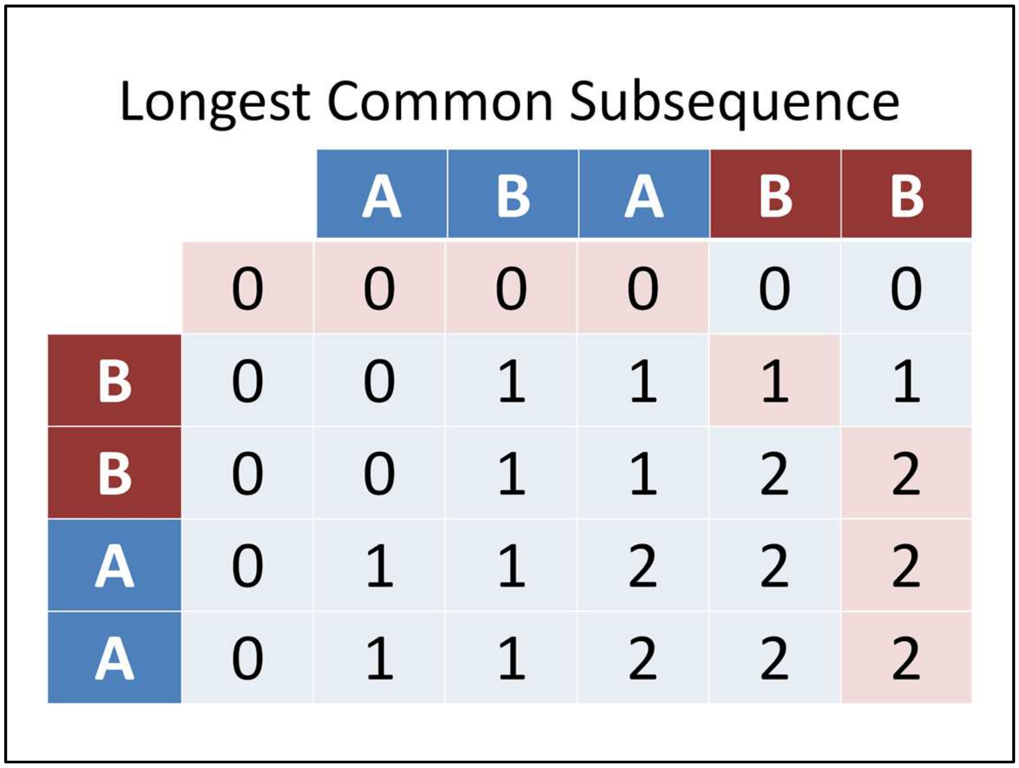
so we take another diagonal, matching another character.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

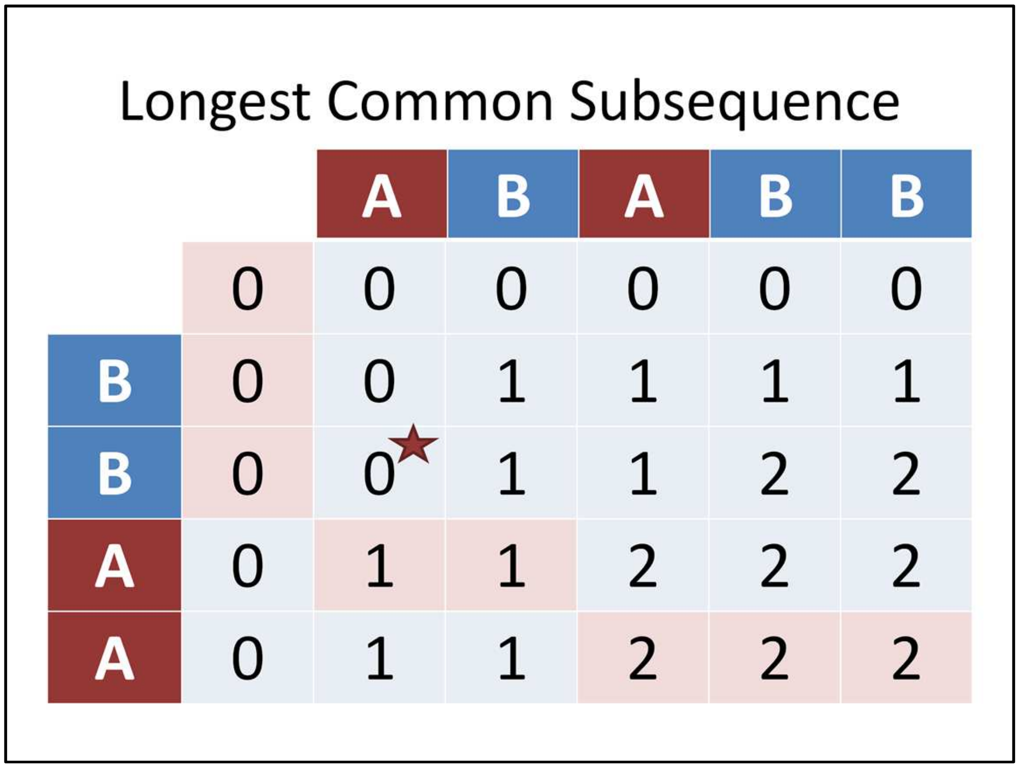
Then we move left and up until we make it to the top left corner. Once we're done, the diagonals line up with the longest common subsequence, which in this case is BA.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

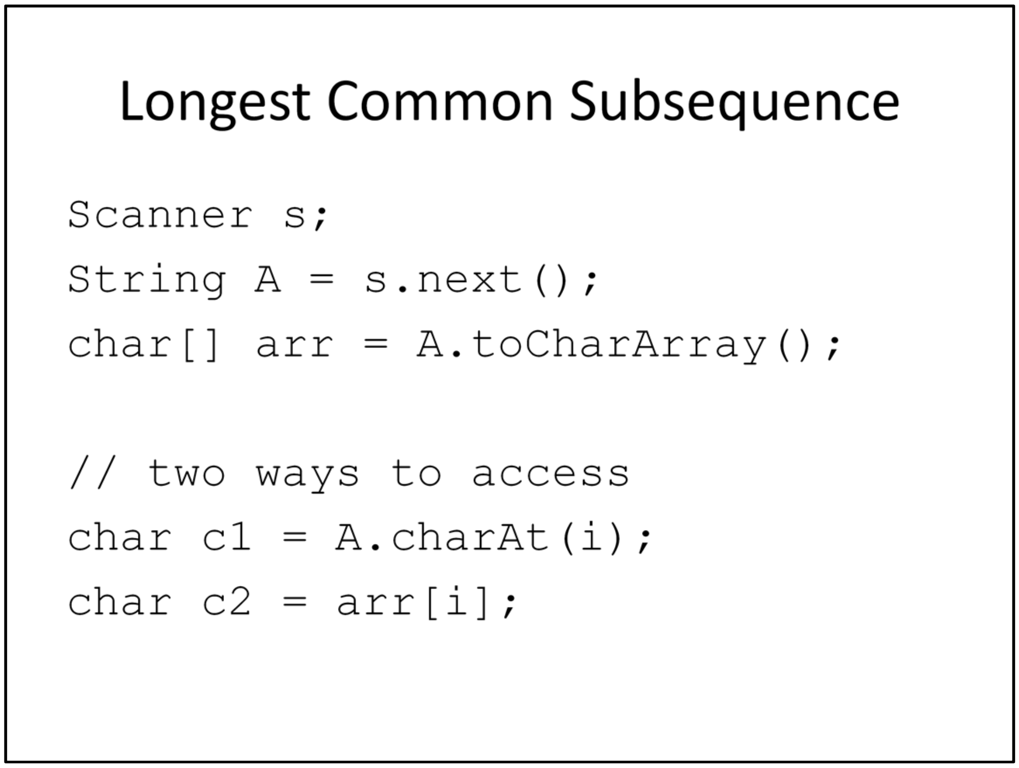
Notice that there are other paths that we could have taken, like the one here, where we prefer moving up first, then diagonally, and then left only if there are no other options. Notice that this gives us a different common subsequence, but it's still the same length, so it's still a correct answer.



Longest Common Subsequence

		A	B	A	B	B
	0	0	0	0	0	0
B	0	0	1	1	1	1
B	0	0★	1	1	2	2
A	0	1	1	2	2	2
A	0	1	1	2	2	2

Alternatively, we could prefer taking diagonals first, then lefts, and then ups, which would give us a different common subsequence. Notice that even though we prefer diagonals over lefts, we didn't take the starred entry. Why is that? (The characters don't match.) Notice that preferring left, then up, then diagonal does something special for us in that it allows us to skip checking the characters before taking the diagonal. This ONLY works if we prefer left then up, or up then left, before the diagonal. If the diagonal is NOT our last resort, then we HAVE to check the characters first. I'd encourage you folks to try tracing out the path after you're done getting the length working, so that you can actually see the subsequences produced. I haven't provided data to check the subsequences, though, since there can be many different LCSes.

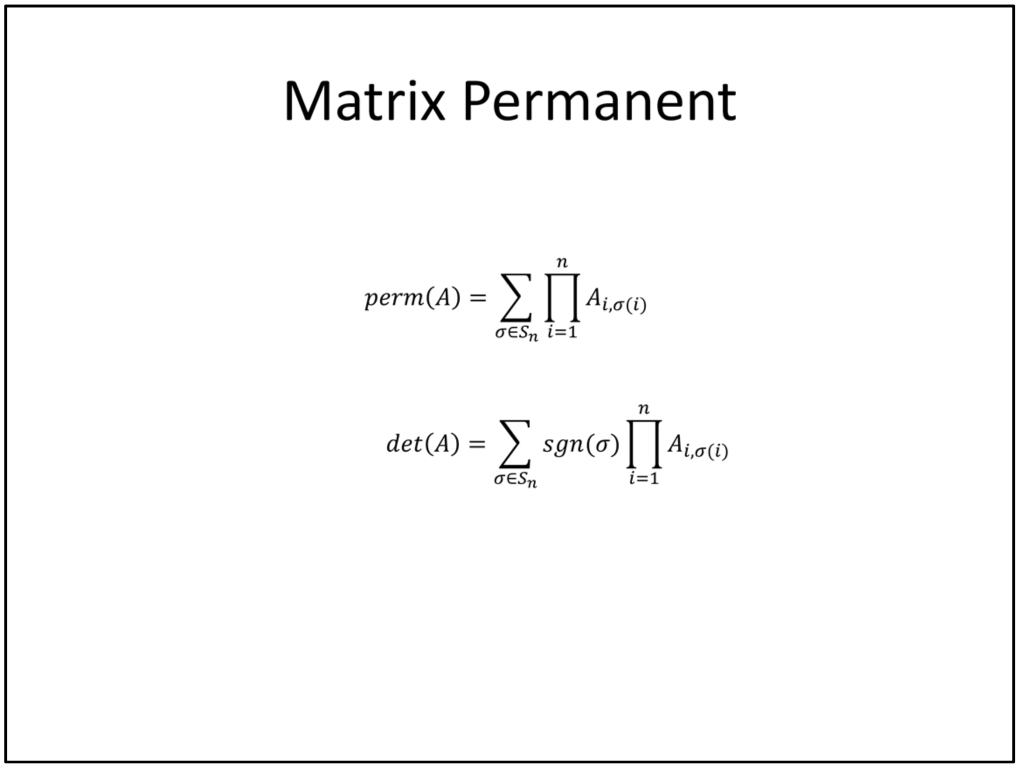


Longest Common Subsequence

```
Scanner s;
String A = s.next();
char[] arr = A.toCharArray();

// two ways to access
char c1 = A.charAt(i);
char c2 = arr[i];
```

A quick aside: This is the first assignment where you'll need to read in a String as input. Doing so is actually pretty simple; just use Scanner.next() to get the next word (the Scanner is whitespace-delimited), and then you can access the String either as an Object or by dumping it to a character array.

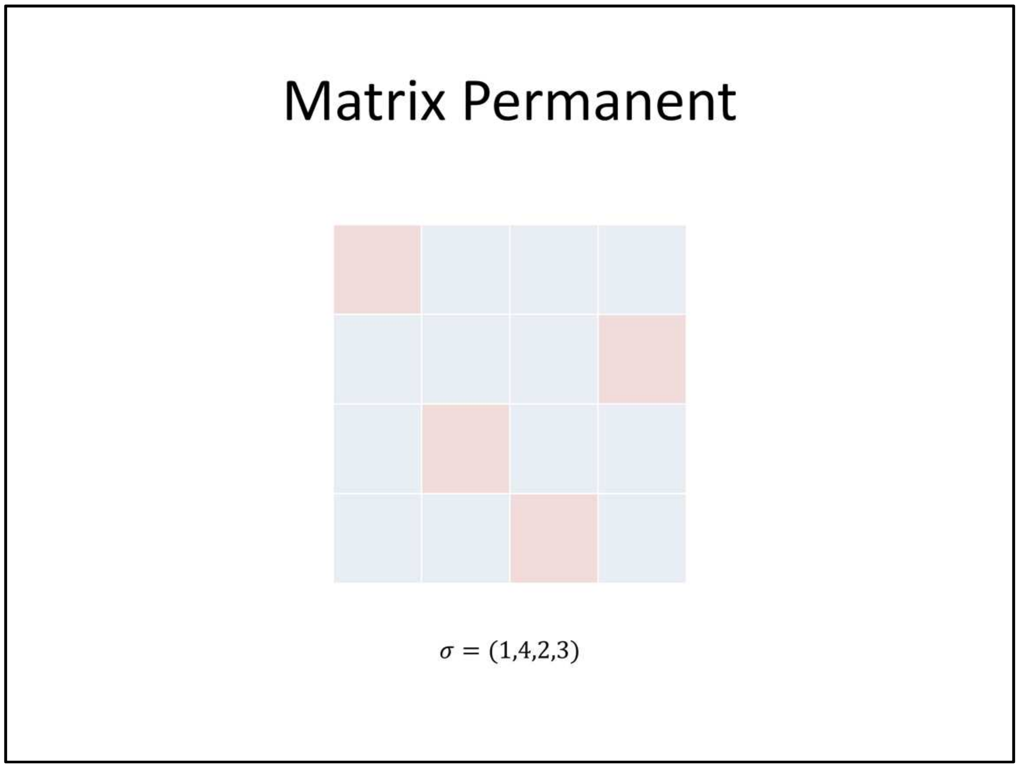


Matrix Permanent

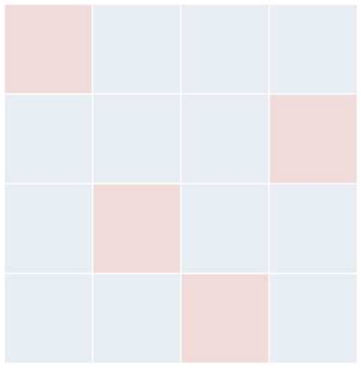
$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

$$det(A) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$$

For the required problem for this week, we're going to go over how to compute the permanent of a matrix. The formal definition of the permanent of a matrix A is given here. This definition may look familiar to some of you, because it's very similar to the definition of a determinant, which is much more widely known. Basically, the determinant of a matrix is the signed sum of the product of permutations of entries in the matrix, while the permanent is the unsigned sum. It's interesting to note that we know how to compute a determinant in polynomial time using row reduction, but computing the permanent is known to be NP-hard. It turns out that those alternating signs have a huge impact on the tractability of the problem.

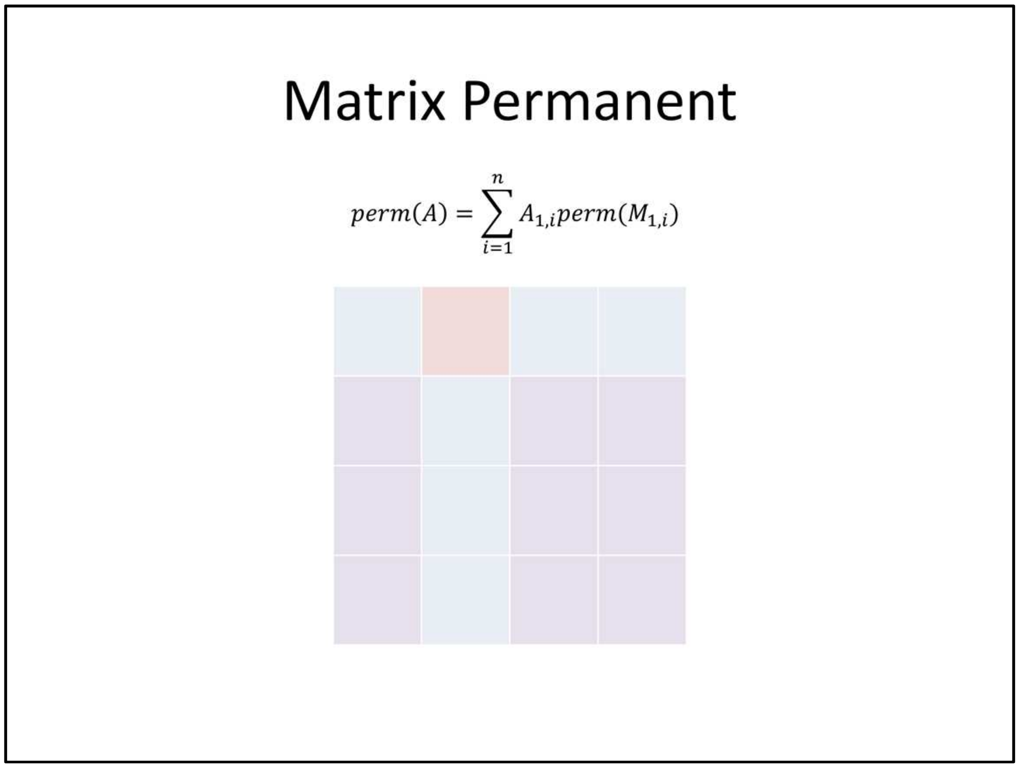


Matrix Permanent



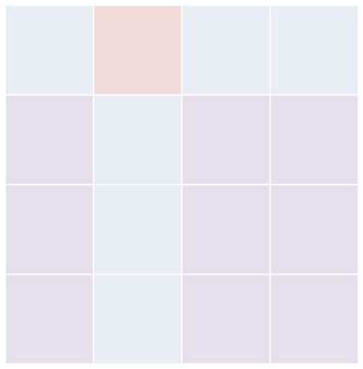
$\sigma = (1,4,2,3)$

To make sure we understand exactly what the permanent is, let's look at an example. Remember that we said that we're summing over all permutations of entries. What does that mean? Well, let's look at one permutation, say 1, 4, 2, 3. What this means is we go through the rows one-by-one, and we take the first entry, then the fourth, then the second, and finally the third, and we multiply all those entries together. This gives us one product per permutation. Then we sum over these n factorial permutations to get our answer. If we were taking the determinant, we would multiply each product with the sign of the permutation before summing, but in the permanent, we sum them all directly.

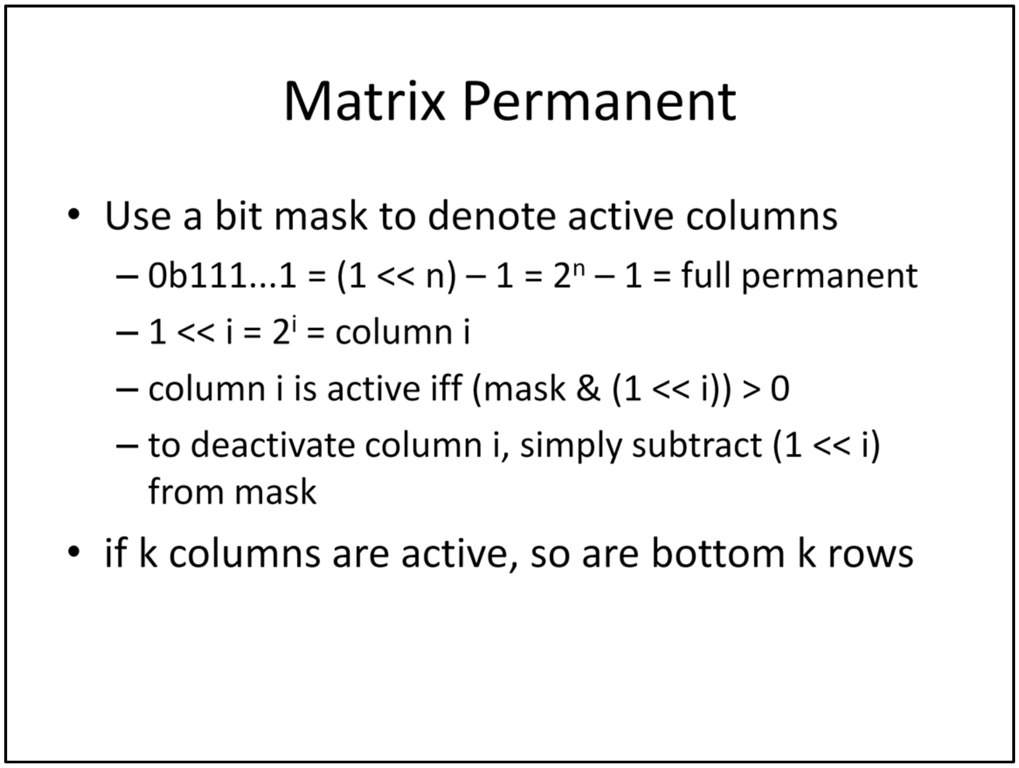


Matrix Permanent

$$perm(A) = \sum_{i=1}^{n} A_{1,i} perm(M_{1,i})$$



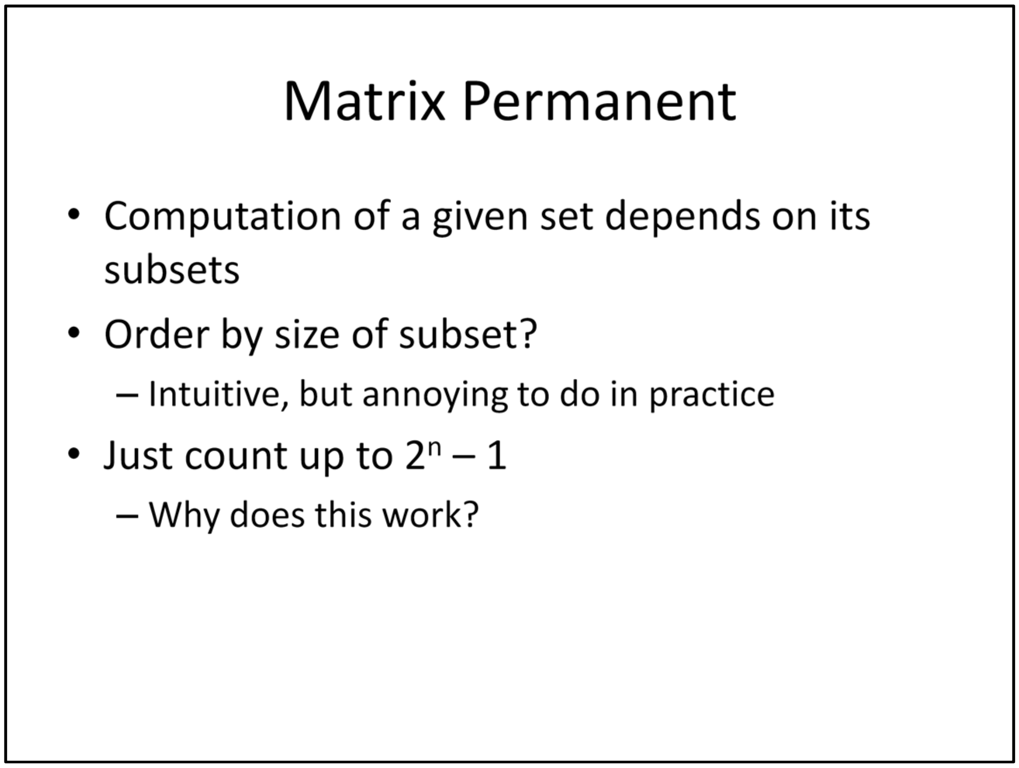
So we could solve this problem in factorial time by enumerating all the permutations. But can we do better? It turns out we can, using just exponential time and space. Yes, I actually said "just" before exponential. As you saw in homework 1, there's actually a pretty big gap between factorial and exponential growth, so this kind of reduction can still be worth it. First, let's write our permanent in terms of permanents of smaller matrices. What we can do is take the expression we had before and group it into n parts, the first being all the permutations that start with 1, the next being all the ones that start with 2, and so on. If we do this, then we see that each part contributes an amount equal to its corresponding entry in the first row, times the permanent of the submatrix you get if you delete the first row and the corresponding column. For example, the part that corresponds to permutations that start with 2 here equals the red entry times the permanent of the purple submatrix. Now, if we were just to use this recurrence directly, we'd still have to do factorial work. However, we can notice that submatrices can show up multiple times. For example, the submatrix that's left over for all permutations that start with (1, 2) is the same as the submatrix that's left over for all the ones that start with (2, 1). In general, to figure out what submatrix we want if we've fixed the first k entries of our permutation, we don't actually care about the order in which those entries were fixed. All we care about is which entries were fixed, because they correspond to the columns we have to delete. Remember that because of the way we're doing this expansion, we're always deleting the first k rows. This means that we actually only have 2^n subproblems, one for each possible submatrix that we create in this manner.



Matrix Permanent

- Use a bit mask to denote active columns
  - 0b111...1 = (1 << n) – 1 = 2^n – 1 = full permanent
  - 1 << i = 2^i = column i
  - column i is active iff (mask & (1 << i)) > 0
  - to deactivate column i, simply subtract (1 << i) from mask
- if k columns are active, so are bottom k rows

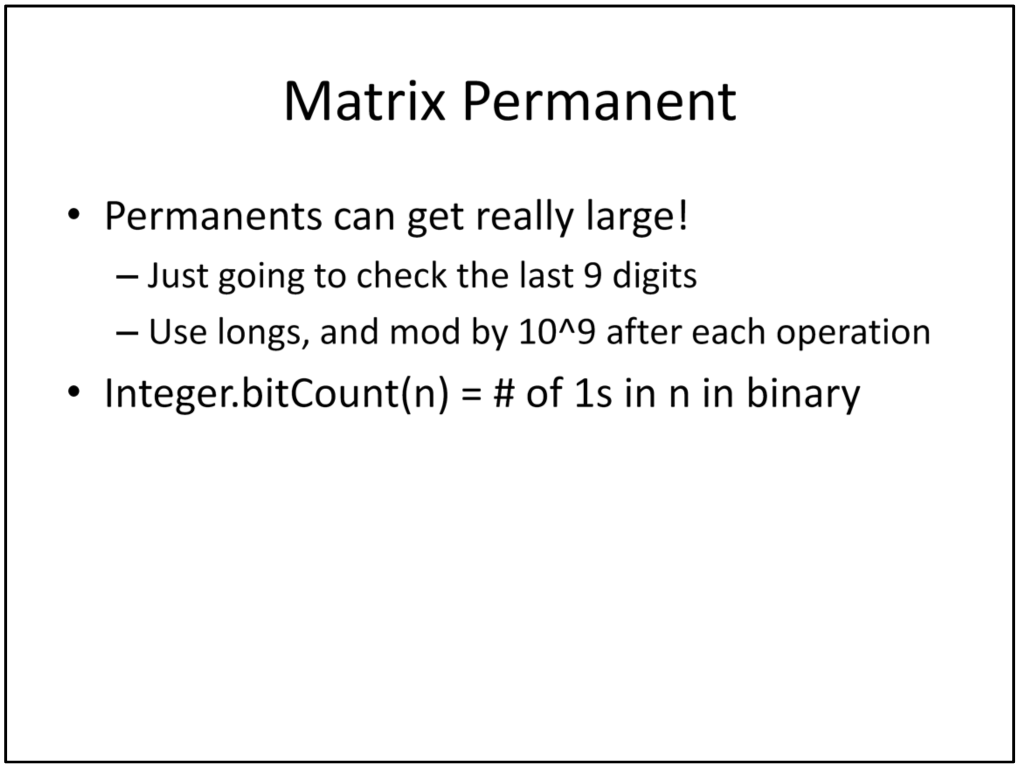
So, how do we organize these subproblems? Well, we can use a bit mask where the 1s indicate the columns that we still have, and the 0s are columns that we've deleted. This means that our full permanent corresponds to the bit mask of all 1s, and to find subproblems, we just subtract those 1s out. These numbers are then the numbers that we use to index into our table. You can see that they range from 1 to 2^n – 1, which corresponds to all nonempty subsets of n elements. To have a convenient base case, we can set the entry at index 0 to 1, which is saying that the permanent of a 0-by-0 matrix is 1.



Matrix Permanent

- Computation of a given set depends on its subsets
- Order by size of subset?
  - Intuitive, but annoying to do in practice
- Just count up to $2^n – 1$
  - Why does this work?

Now how do we fill this table? Well, remember that the important thing is that whenever we try to fill in an entry, all of the entries it depends on have to already be filled. In our case, that means when we're looking at a particular set of active columns, we need to make sure that we've already handled all subsets of that set. One way that we can do this is to first handle all sets of size 1, then all sets of size 2, and so on and so forth. While this makes a lot of intuitive sense, it's actually pretty annoying to iterate over all sets of a given size. Instead, I'm going to propose the following: Just start from the bit mask 0, and count all the way up to 2^n – 1. Why does this work? Well, remember what we did to get a subset of our current set? We flipped a bit from 1 to 0, which corresponds to subtracting a positive number from our bitmask. That means that the subsets our current set depends on have a bit mask that is strictly less than that of our current set, so if we count upwards, we'll always compute the subsets we need first. This is a really handy trick for subset DPs in general.



Matrix Permanent

- Permanents can get really large!
  - Just going to check the last 9 digits
  - Use longs, and mod by 10^9 after each operation
- Integer.bitCount(n) = # of 1s in n in binary

A couple more notes. First, matrix permanents can get really big really fast, since we're multiplying together a lot of numbers. For this assignment, we're more interested in the technique used for computing the permanent than the actual result, so we're only going to keep the last 9 digits of the permanent around to make sure our setup is correct. What this means is you should be doing all your computation using longs, and after every arithmetic operation, you should drop all but the last 9 digits by taking the result modulo a billion. This works because I've guaranteed that there won't be any negative numbers in the matrices I give you. Second, because we're iterating over all the bit masks in increasing order, the size of the subset we're looking at isn't very well organized. In order to figure out the size, we need to know how many 1s are present in our bit mask. Thankfully, there's a built-in function that does just that.