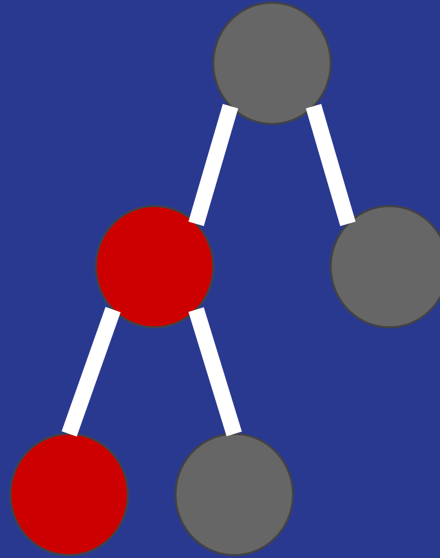


Multiplication [edit]

Polynomials can also be multiplied. To expand the product of two polynomials into a sum of terms, the distr
results in each term of one polynomial being multiplied by every term of the other.[7] For example, if

$$P = 2x + 3y + 5$$
$$Q = 2x + 5y + xy + 1$$

then

$$\begin{array}{rccccccc} PQ = & (2x \cdot 2x) & + & (2x \cdot 5y) & + & (2x \cdot xy) & + & (2x \cdot 1) \\ + & (3y \cdot 2x) & + & (3y \cdot 5y) & + & (3y \cdot xy) & + & (3y \cdot 1) \\ + & (5 \cdot 2x) & + & (5 \cdot 5y) & + & (5 \cdot xy) & + & (5 \cdot 1) \end{array}$$

Carrying out the multiplication in each term produces

$$\begin{array}{rccccccc} PQ = & 4x^2 & + & 10xy & + & 2x^2y & + & 2x \\ + & 6xy & + & 15y^2 & + & 3xy^2 & + & 3y \\ + & 10x & + & 25y & + & 5xy & + & 5. \end{array}$$

Combining similar terms yields

$$\begin{array}{rccccccccc} PQ = & 4x^2 & + & (10xy + 6xy + 5xy) & + & 2x^2y & + & (2x + 10x) \\ + & 15y^2 & + & 3xy^2 & + & (3y + 25y) & + & 5 \end{array}$$

which can be simplified to

$$PQ = 4x^2 + 21xy + 2x^2y + 12x + 15y^2 + 3xy^2 + 28y + 5.$$

As in the example, the product of polynomials is always a polynomial.[9][4]

CS 9

Week 9 Problems

Andrew Benson
Ian Tullis

Announcements

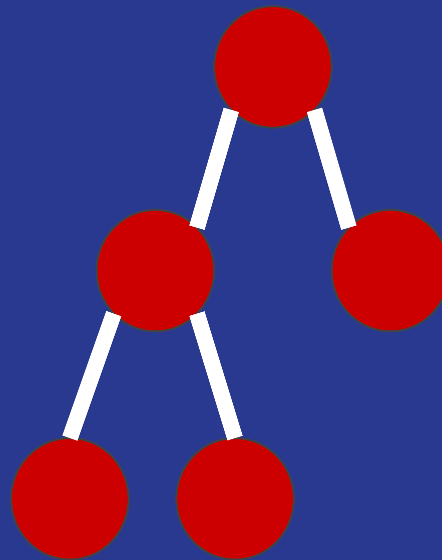
- This is the last set of problems! Next week, there is only a meeting on Tuesday.
    - Final thoughts, and some stuff about ML interviews…

- You can get 1 bonus point for doing the course feedback survey via Axess (when it opens) and then letting us know on Gradescope that you did it.
    - We have no way (or intention) of matching your Gradescope feedback to your Axess feedback. But you can always space them apart by some random amount of time…

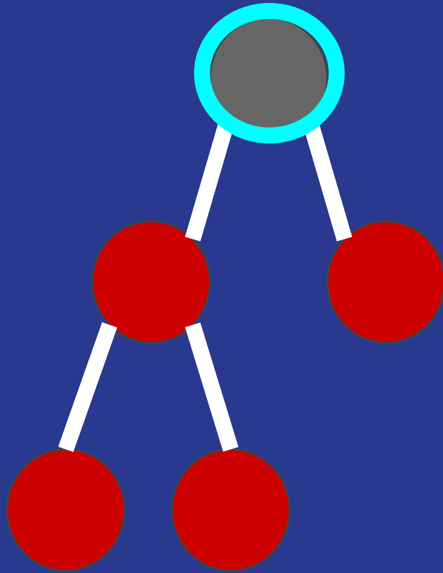
⭐ Problem 9-1: Paint It Black

You have a binary tree in which each of the N nodes ($3 \le N \le 9999$) has either 0 or 2 children. Initially, every node is colored red.

You start at the root, and automatically turn that node black. Thereafter, you can only move from your current node to an adjacent node. Each time you visit a node, you automatically toggle its color from red to black (or vice versa). You are done when every node is black (it doesn't matter where you finish). Write code to do this.

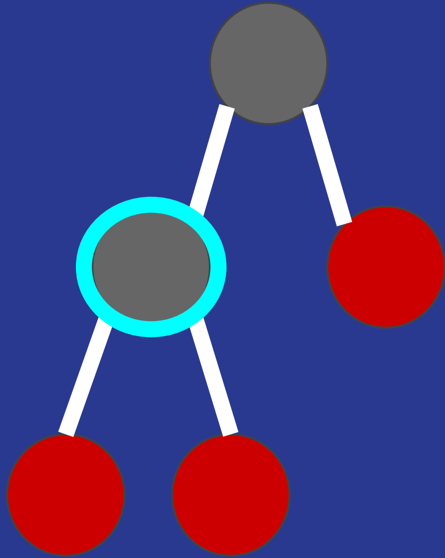


⭐ Problem 9-1: Example

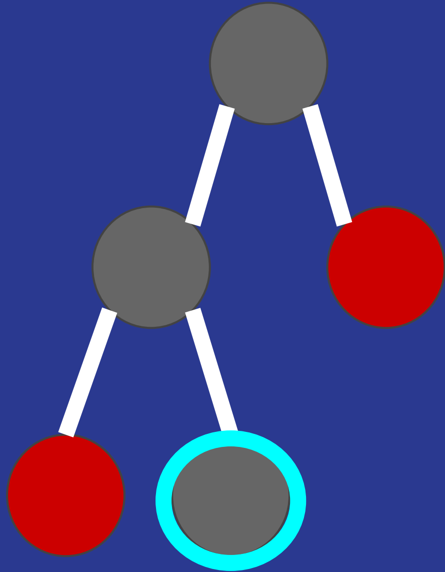


Start at root, automatically turn it black

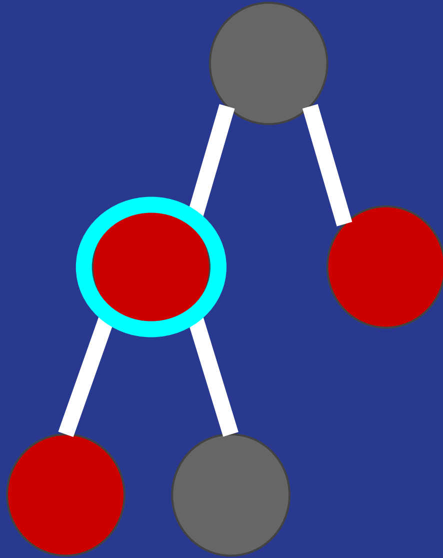
⭐ Problem 9-1: Example



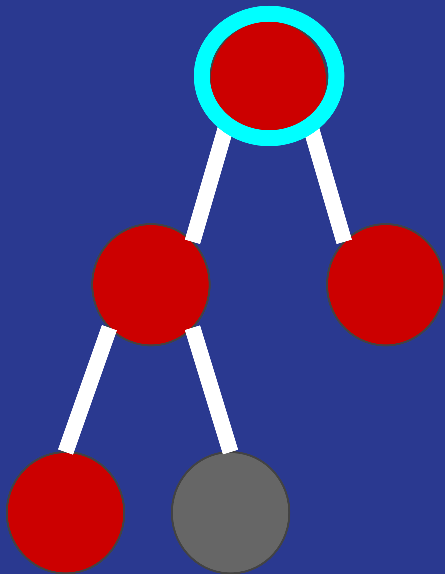
⭐ Problem 9-1: Example



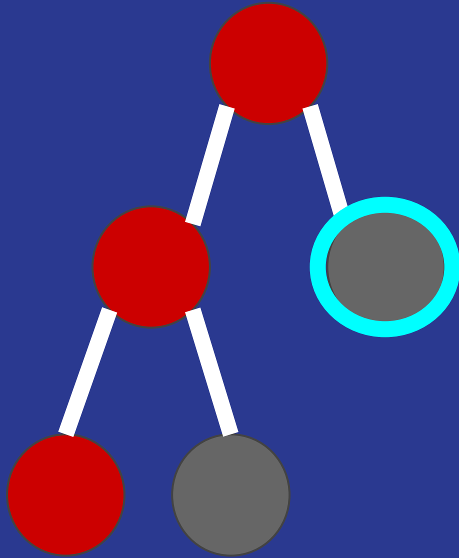
⭐ Problem 9-1: Example



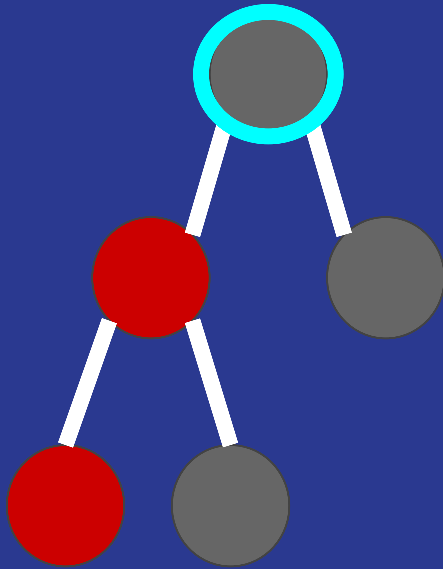
⭐ Problem 9-1: Example



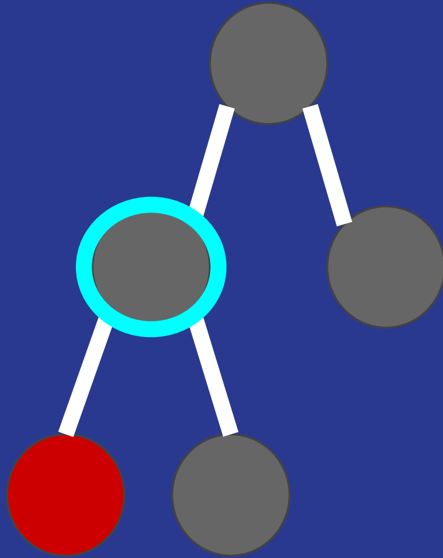
⭐ Problem 9-1: Example



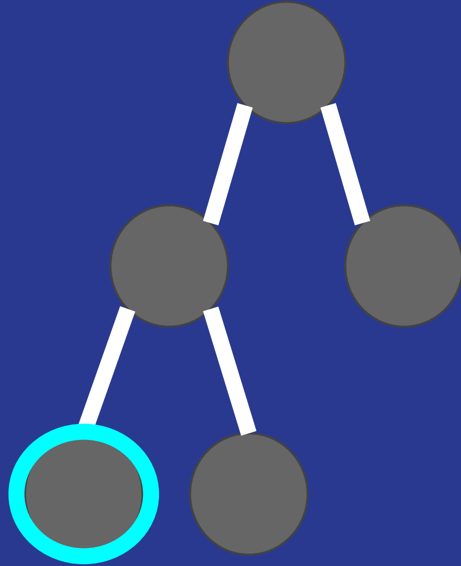
⭐ Problem 9-1: Example



⭐ Problem 9-1: Example



⭐ Problem 9-1: Example



All done!

(Note: this was just one way of doing it…)

⭐⭐ Problem 9-2: Polynomials

In this problem, a polynomial is given as a sum of terms, each of the form

$$(a_0 + a_1 x + a_2 x^2 + \ldots)^b$$

We will represent each such term as a list `[a0, a1, a2, ..., b]`, where each element is an integer, the last element is understood to be the exponent (1 or greater), and the next-to-last element is not 0.

So, for example, the expression $\mathbf{(1 - x)^2 + (-3 + x^3)^1}$ would be given as:

```
[[1, -1, 2], [-3, 0, 0, 1, 1]]
```

⭐⭐ Problem 9-2: Polynomials

Your code will be presented with two polynomials in this form, and must determine whether they are the same. Example:

```
[[1, -1, 2], [-3, 0, 0, 1, 1]]
[[-1, 1, 3], [-1, 2, 2], [-2, 1]]
```

These turn out to be almost, but not quite, the same! If the second polynomial had another `[0, -1, 1]` term, they would be the same. (Might be worth checking this by hand to get a feel for the setup…)

First try to write code that would work on small examples, then think about how to handle large ones. (This is intentionally vague…)

⭐⭐⭐ Problem 9-3: MissingNo.

Your solution is only allowed to use a constant amount of memory and a constant number of passes over the data. Storing any number takes constant memory regardless of its size.

You are presented with a number *N* and a (not necessarily sorted) list of *N* numbers, in a streaming fashion (you cannot edit the list). Each number from 1 to *N* occurs exactly once in the list, except:

- (warm-up) One number is missing. (The list is length *N*-1)
- (harder) Two numbers are missing. (length *N*-2)
- ⭐⭐⭐⭐(even harder) Three numbers are missing. (length *N*-3)

Determine the missing number(s).

Solutions to 9-2 (discussed in-class)

Overall Idea

- What does it mean for two polynomials to be the same?
- Every polynomial can be **uniquely** expressed in the form

$$a_0 + a_1x + a_2x^2 + \dots$$

up to some power. (Note the lack of an overall exponent here.) We'll call this a *standard form*.

- So to see if two expressions are really the same polynomial, we can put them both in this standard form and then compare.

Road map

For each polynomial,

- Convert each term to a standard form, then add the terms together to get an overall standard form.
    - To do this, we need to be able to **add standard forms**, and also **deal with exponents**.
        - To add two standard forms, we can add the lists of coefficients entry-wise.
        - To deal with exponents, we can **multiply** a term by itself (and then a running product) over and over.
            - To multiply two standard forms, we do something like the grade school multiplication algorithm.

Adding standard forms

```
def add(coeffs1, coeffs2):
    # Make it so coeffs2 is no shorter than coeffs1.
    if len(coeffs1) > len(coeffs2):
        coeffs1, coeffs2 = coeffs2, coeffs1
    result = coeffs2[:]
    for i in range(len(coeffs1)):
        result[i] += coeffs1[i]
    # Subtle point: there could be one or more trailing zeroes.
    # Remove all (unless the expression is all zeroes, then leave one)
    i = len(result) - 1
    while i >= 0 and result[i] == 0:
        i -= 1
    return result[0:i+1]
```

Multiplying standard forms

```
def multiply(coeffs1, coeffs2):
    degree1 = len(coeffs1) - 1
    degree2 = len(coeffs2) - 1
    result = [0] * (degree1 + degree2 + 1)
    for i in range(len(coeffs1)):
        for j in range(len(coeffs2)):
            new_degree = i + j
            new_coeff = coeffs1[i] * coeffs2[j]
            result[new_degree] += new_coeff
    # No need to worry about the last term being nonzero.
    return result
```

(We could improve this by using binomial coefficients to calculate the new coefficients directly.)

Handling exponents

```
def handle_exponent(coeffs, exp):
    result = coeffs
    for i in range(exp - 1):
        result = multiply(result, coeffs)
    return result
```

(There are also better ways to do this. One good trick to know is *repeated squaring*. For example, to compute the 13th power of some x, first square it to get x^2, then square that to get x^4, then square that to get x^8. Then take x times x^4 times x^8, since $13 = 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$. This takes many fewer operations than just finding x times x times x etc.)

Putting it together

```
def handle_poly(p):
    result = [0]
    for term in p:
        coeffs, exp = term[:-1], term[-1]
        simplified_term = handle_exponent(coeffs, exp)
        result = add(result, simplified_term)
    return result

def solve(p1, p2):
    simplified_p1 = handle_poly(p1)
    simplified_p2 = handle_poly(p2)
    if len(simplified_p1) != len(simplified_p2):
        return False
    for i in range(len(simplified_p1)):
        if simplified_p1[i] != simplified_p2[i]:
            return False
    return True
```

Hey, that was just boring math

- But the implementation details were nontrivial!
- And sometimes, interview problems are math masquerading as computer science.
  - (Arguably, *computer science* is math masquerading as computer science)
- One issue: this **blows up** for even modestly large exponents. Is there some better way to do this?

A different take

- What does it mean for two polynomials to be the same?
- For every input, they must produce the same output as each other.
- So if we find an input that makes our two polynomials produce different results, they are not the same…
- …and evaluating a polynomial at an input is way easier than doing the grindy additions/multiplications to find standard forms.

But wait...

- What if we just happen to pick an input where the two polynomials happen to produce the same result, even though they are different?
- No problem! Just try a bunch of different inputs. If we ever get different outputs, return DIFFERENT. Otherwise, eventually return SAME.
- But how do we know how many different inputs to try?

But wait...

- Let $p(x)$ and $q(x)$ be our two polynomials we're comparing. Let's think about the difference between the two, $p(x) - q(x)$.

- Now, $p(x) - q(x)$ is itself a polynomial, and its degree (highest-order exponent) is no greater than the degree of either $p(x)$ or $q(x)$.

- Also, a polynomial of degree d can have at most d distinct roots (inputs that make the polynomial output 0). (FWIW, this is called the Fundamental Theorem of Algebra)

- Therefore, if we check more than d random inputs, we should be fine.

Another complication

- These values are still going to get pretty enormous…

- We can carry out all computations modulo some large prime. Why a prime? The usual answer is that this is necessary for division to be possible, but we don't care about that here…
  - We do want the limit on the number of roots to still hold even in this modular arithmetic world, and choosing a prime guarantees that (for hard mathy reasons).

- (There are efficient ways to do an exponentiation and take that mod some value, but we omit that here for simplicity.)

Now it all fits on one slide!

```
PRIME = 10**9 + 7
def evaluate_poly(poly, v):
    total = 0
    for term in poly:
        subtotal = 0
        coeffs, exp = term[:-1], term[-1]
        for i in range(len(coeffs)):
            subtotal += (coeffs[i] * (v**i % PRIME)) % PRIME
        total = (total + (subtotal ** exp) % PRIME) % PRIME
    return total

import random
def solve(p1, p2):
    max_degree = 0
    for p in (p1, p2):
        for term in p:
            max_degree = max(max_degree, term[-2] * term[-1])
    # We could have randomized this, but instead we just try all
    # distinct possible values less than max_degree.
    for i in range(max_degree + 1):
        if evaluate_poly(p1, i) != evaluate_poly(p2, i):
            return False
    return True
```

Takeaways

- Randomized algorithms are powerful (and sometimes their randomness can be removed for extra assurance).
- Make sure you think about how to handle very large numbers – the "take everything modulo a prime" trick is very useful in these problems and even in practice!
- Math-related problems do show up in these interviews, so if math is a weak point for you, make sure to get practice in.

Don't let Ian forget to say the password!!!!!!

Solutions to 9-3

One missing number

- The "just put everything in a hash table" strategy (e.g. from the Legs problem in Week 1, which feels like a million years ago now) does not work here, since it uses more than constant memory.

- Notice that if the numbers were arbitrary rather than 1 through N, the problem would be impossible – we wouldn't have any way to know what was missing. So we have to somehow use this special condition.

- We know the sum of **all** numbers from 1 to N... it's $(N)(N+1)/2$. So we can find the sum of *our* list (using constant space) and subtract it from $(N)(N+1)/2$, and there's our missing number!

Two missing numbers

- In this case, we can use the previous method to find the sum, $a + b$, of the two missing numbers a and b. So it seems like we're close, but there are probably lots of pairs of numbers with that sum… we'd still need non-constant space to check them all. We need another piece of information!

- This is where we can take a huge leap… what if we also store the sum of squares (*SS*) of the numbers?

- There is a formula for the sum of squares of all numbers from 1 to *N*… it's $(N)(N+1)(2N+1)/6$. So if we subtract SS from that, we get the sum of squares of the missing numbers – $a^2 + b^2$.

- We have $a + b$ from before. Now we use the fact that $(a + b)^2 = a^2 + 2ab + b^2$, and solve for $2ab$, and then plug that back into our $a + b$ expression. Voila!

Two missing numbers - other solutions

There are a couple other ways to solve this problem.

- One is to find $a + b$ as before, then use the fact that $(a + b)/2$ is the average of the missing numbers. One of them (without loss of generality, say it's a) is less than that average, and one is greater. So if we compute the sum of all numbers less than or equal to that average, we will be missing only a. Note that this requires a second pass over the data.

- By the way, how do we deal with the potentially enormous sums (or, worse yet, sums of squares)? One option is to compute everything modulo a large prime, as in the slick solution to the Polynomials problem.

Two missing numbers - other solutions

There is also a clever method involving XOR, with no overflow issues!

- We take the XOR of the all of the values, and XOR that with the XOR of all values from 1 to *N*. What's left is the XOR of *a* and *b*; call that *c*.
- If there is a 1 bit in *c*, then one of *a* and *b* has a 1 in that position and the other has a 0.
- Also, there must be at least one 1 bit in *c*, since for *c* to be all 0s, *a* and *b* would have to be the same, which we know is not true.
- So we pick some 1 bit in *c*. If we XOR all the values that have that bit set, then XOR that with the XOR of all values between 1 and *N* that have that bit set, we get one of our missing numbers. Then we XOR that with *c* to get the other.

(It's also easy to solve the one-missing version of the problem using XOR.)

Three missing numbers?

This isn't a standard part of the question, but you can imagine extending one of the previous methods.

- We could use the "average" method – now, either two of the missing numbers will be below the average and one will be above it, or one will be the average and there will be one above that and one below that. There are some messy details/cases to handle, and this uses multiple passes.

- Or we can maintain a sum of cubes as well, and there is a formula for that. Then we can maybe? solve a system of equations involving $a + b + c$, $a^2 + b^2 + c^2$, and $a^3 + b^3 + c^3$. I started on it and found abc and then ran out of steam. But that seems to maybe turn into number theory rather than algebra at that point. If you do all the math, let me know!

- Not sure about extending the XOR method. Seems ugly if doable.