Example: Fibonacci sequence.

$$\boxed{a_0 = a_1 = 1.}$$

$$a_n = a_{n-1} + a_{n-2}.$$

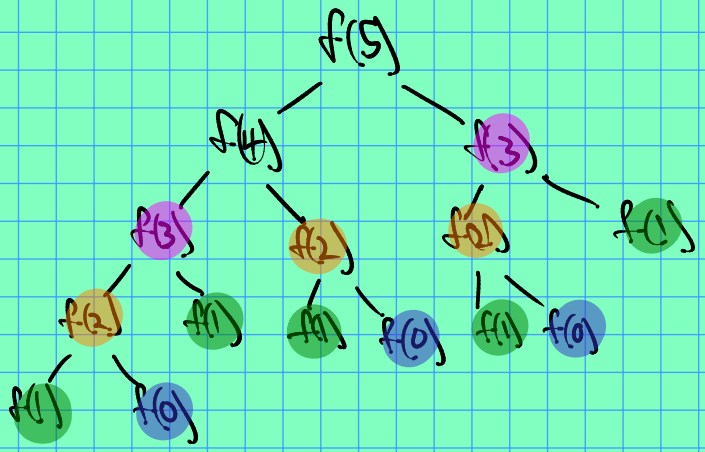
```
int f(int n) {
    if (n<2) return 1;
    return f(n-1) + f(n-2);
}
```

This works. But running it for $n \geq 40$ will get pretty slow...

Recursion Trees

Idea: use a graph. Each node represents a function call. Place an edge between two nodes if one node's execution called the other function.

Trace f(5):



To compute f(n) looks like $\approx 2^n$ Yikes.

Question: at any given moment in time, what is the largest # of "pending" calls to f?
Answer: $\approx n$.

Recursive version looked nice (short, logic mirrored defn. of the sequence), but performance was terrible. Any way to get the best of both worlds?

Memoization

Intuitively, issue w/ recursive fibonacci was that it was forgetful & kept on re-computing f on the same values.

Solution: give it memory!

Could use a map or a vector to store computed values. E.g. vector<int> M;

M[i] ≡ f(i).

Let's try it:

```
int fm(int n, vector<int>& M)
{   // invariant: if n < M.size,
    //then M[n] == f(n)
    if(n < M.size()) return M[n];
    int ans;
    if(n<2) ans = 1;
    else ans = fm(n-1, M) + fm(n-2, M);
    // update memory / table of answers:
    // need to set M[n] = ans, but
    // be careful to make sure there is space,
    // & note that push_back might add
    // elements in the wrong answer.
    ...
    return ans;
}
```