Dynamic Programming
Warmup

Find a minimum spanning tree for the following graph:
With Kruskal’s Algorithm

Find a minimum spanning tree for the following graph:

Choose smallest remaining edge that doesn’t make a cycle. Use disjoint sets to track CC.
With Primm’s Algorithm

Find a minimum spanning tree for the following graph:

```
<table>
<thead>
<tr>
<th>Vertex</th>
<th>Known</th>
<th>D</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>B</td>
<td>✓</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>✓</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>F</td>
<td>✓</td>
<td>5</td>
<td>C</td>
</tr>
</tbody>
</table>
```
Dynamic Programming

When the greedy approach fails.
Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13, ...

\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]
\[ \text{fib}(0) = \text{fib}(1) = 1 \]
public int fib(int i) {
    int result;
    if (i < 2) {
        result = 1;
    } else {
        result = fibonacci(i - 1) + fibonacci(i - 2);
    }
    return result;
}

What is the runtime of this algorithm?
Fibonacci Numbers
Fibonacci Numbers: By Hand

We did this by hand in much less than exponential time. How?
We looked up previous results in the table, re-using past computation.
Big Idea: Keep an array of sub-problem solutions, use it to avoid re-computing results!
Memoization

Memoization is storing the results of a **deterministic** function in a table to use later:

If \( f(n) \) is a deterministic function (from ints to ints):

```java
memo = Integer [N] // N is the biggest input we care about

memoized_f(n) {
    if (memo[n] == null) {
        memo[n] = f(n);
    }
    return memo[n];
}
```

\( \mathcal{O}(N) \)

\( \mathcal{O}(1) \)
Memoized Fibonacci

```java
public int fib(int i) {
    if (memo[i] <= 0) {
        if (i < 2) {
            memo[i] = 1;
        } else {
            memo[i] = fib(i - 1) + fib(i - 2);
        }
    }
    return memo[i]
}
```

memo = int[N]; // initialized to all 0s – use as sentinels since fib(n) > 0 for all n
Dynamic Programming

Breaking down a problem into smaller subproblems that are more easily solved.

Differs from divide and conquer in that subproblem solutions are re-used (not independent)
- Ex: Merge sort:

\[ \text{AFDEGCBQ} \]

Memoization is such a problem is sometimes called “top-down” dynamic programming.

If this is top-down, what is bottom up?
Top Down Evaluation

Which order do we call fib(n) in?
Which order are the table cells filled in?

In bottom-up dynamic programming (sometimes just called dynamic programming), we figure out ahead of time what order the table entries need to be filled, and solve the subproblems in that order from the start!
Fibonacci – Bottom Up

```java
public int fib(int n) {
    int[] fib = new int[n+1];
    fib[0] = fib[1] = 1;  // Base cases: pre-fill the table
    for (int i = 2; i < n; ++i) {  // Loop order is important: non-base cases in order of fill
        fib[i] = fib[i - 1] + fib[i - 2];  // Recursive case: looks just like a recurrence
    }
    return fib[n];
}
```

<table>
<thead>
<tr>
<th>Pros:</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs faster</td>
<td>More difficult to write</td>
</tr>
<tr>
<td>Won't build up a huge call stack</td>
<td></td>
</tr>
<tr>
<td>Easier to analyze runtime</td>
<td></td>
</tr>
</tbody>
</table>
An Optimization

We only ever need the previous two results, so we can throw out the rest of the array.

```java
public int fib(int n) {
    fib = new int[2];
    fib[0] = fib[1] = 1;
    for (int i = 2; i < n; ++i) {
        fib[i % 2] = fib[0] + fib[1];
    }
    return fib[n%2];
}
```

Now we can solve for arbitrarily high Fibonacci numbers using finite memory!
Another Example

Here’s a recurrence you could imagine seeing on the final. What if you want to numerically check your solution?

\[ C(N) = \frac{2}{N} \sum_{i=0}^{N-1} C(i) + N \]

\[ C(0) = 1 \]
Recursively

public static double eval(int n) {
    if (n == 0 ) {
        return 1.0;
    } else {
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            sum += eval(i);
        }
        return 2.0 * sum / n + n;
    }
}

What does the call tree look like for this?

With your neighbor: Try writing a bottom-up dynamic program for this computation.
With Dynamic Programming

\[ C(5), C(4), C(3), C(2), \ldots, C(0) \]

\[
C(n) = \frac{1}{n} \sum_{i=0}^{n-1} C(i) + n
\]

\[ C(0) = 1 \]

\[ C(n) = \frac{2}{n} \sum_{i=0}^{n-1} C(i) + n \]

\[ C = \text{new array}[n+1] \]

\[ C[0] = 1 \]

\[
\text{for}\ \ i = 1;\ \ i \leq n;\ \ i++ \\ \ \ 3
\]

\[
\text{sum} = 0
\]

\[
\text{for}\ \ j = 0 \text{ to } n-1
\]

\[
\text{sum} += C[j]
\]

\[ C[0] = \frac{2}{n} \text{sum} + n \]

\[ \text{return}\ \ C[n] \]
public static double eval( int n ) {
    double[] c = new double[n + 1]; // n + 1 is pretty common to allow a 0 case
    c[0] = 1.0;
    for (int i = 1; i <= n; ++i) { // Loop bounds in DP look different, not always 0 < x < last
        double sum = 0.0;
        for (int j = 0; j < i; j++) {
            sum += c[j];
        }
        c[i] = 2.0 * sum / i + i;
    }
    return c[n];
}
Where is Dynamic Programming Used

These examples were a bit contrived.

Dynamic programming is very useful for **optimization** problems and **counting** problems.  
- Brute force for these problems is often exponential or worse. DP can often achieve polynomial time.

Examples:

- How many ways can I tile a floor?
- How many ways can I make change? *
- What is the most efficient way to make change? *
- Find the best insertion order for a BST when lookup probabilities are known.
- All shortest paths is a graph. *
Coin Changing Problem (1)

THIS IS A VERY COMMON INTERVIEW QUESTION!

Problem: I have an unlimited set of coins of denominations \( w[0] \), \( w[1] \), \( w[2] \), ... I need to make change for \( W \) cents. How can I do this using the minimum number of coins?

Example: I have pennies \( w[0] = 1 \), nickels \( w[1] = 5 \), dimes \( w[2] = 10 \), and quarters \( w[3] = 25 \), and I need to make change for 37 cents.

I could use 37 pennies (37 coins), 3 dimes + 1 nickels + 2 pennies (6 coins), but the optimal solution is 1 quarter + 1 dime + 2 pennies (5 coins).

We want an algorithm to efficiently compute the best solution for any problem instance.