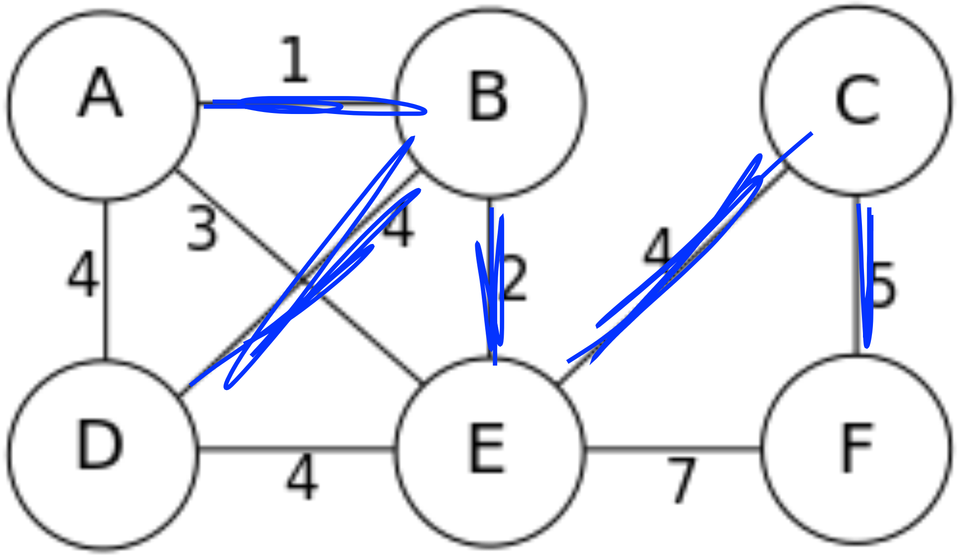
Dynamic Programming

Data Structures and Algorithms

Warmup

Find a minimum spanning tree for the following graph:

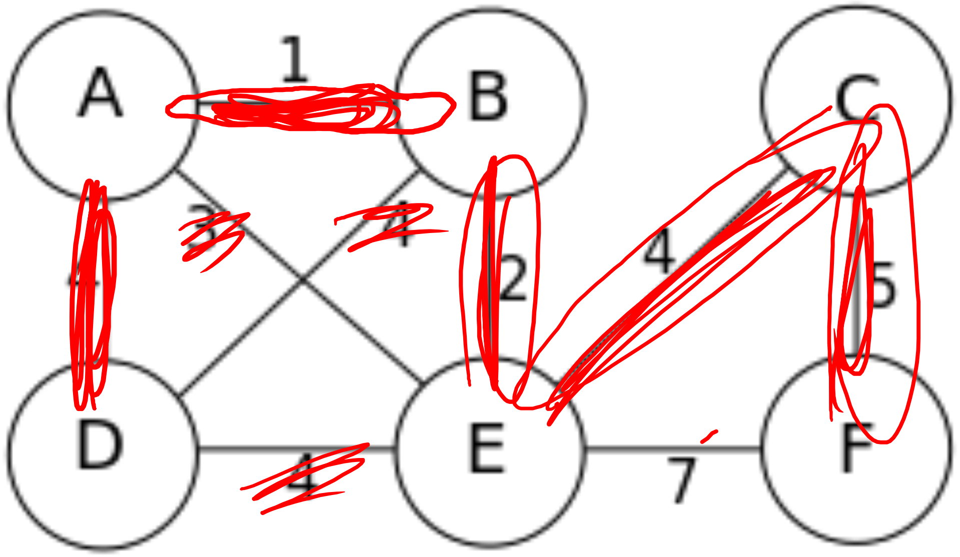


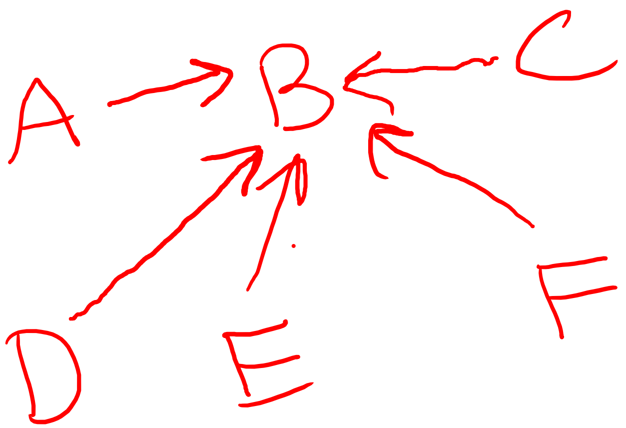
With Kruskal's Algorithm

Choose smallest remaining edge that doesn't make a cycle.

Use disjoint sets to track CC

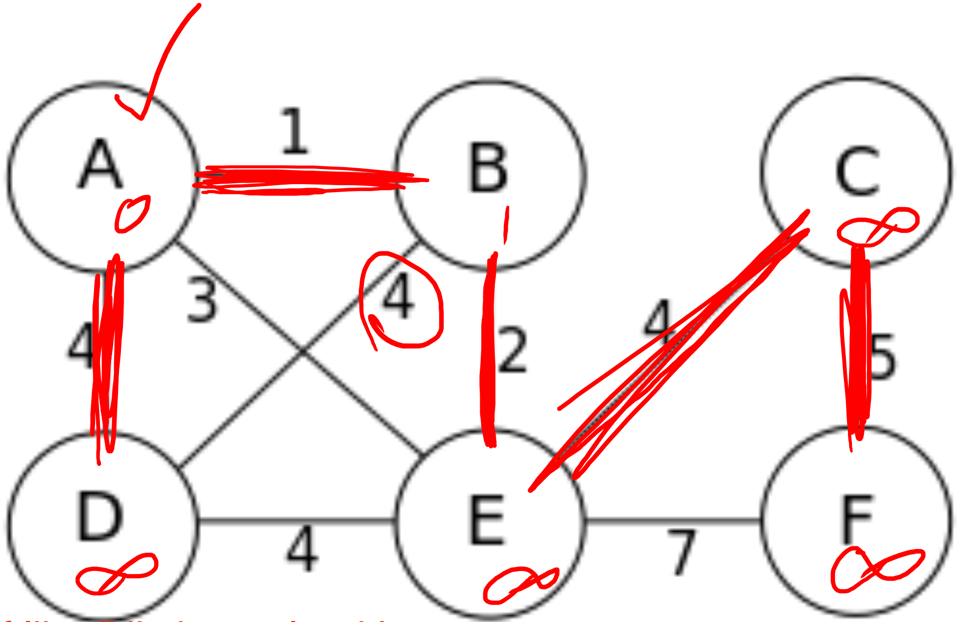
Find a minimum spanning tree for the following graph:





With Primm's Algorithm

Find a minimum spanning tree for the following graph:



Vertex	Known	D	P
A	✓	0	—
B	✓	1	A
C	✓	4	E
D	✓	4	A
E	✓	2	B
F	✓	5	C

n-1 edges

Kind of like Dijstkras algorithm

set all distances to infinity.
Choose an arbitrary "start vertex" as current; mark its distance as 0
for each adjacent vertex, if the cost of edge from current vertex to it < its distance, set distance to that edge weight and predecessor to current
let current = vertex with smallest distance; add edge predecessor —> new current to min span tree
repeat

Dynamic Programming

When the greedy approach fails.

Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13, …

fib(n) = fib(n-1) + fib(n-2)

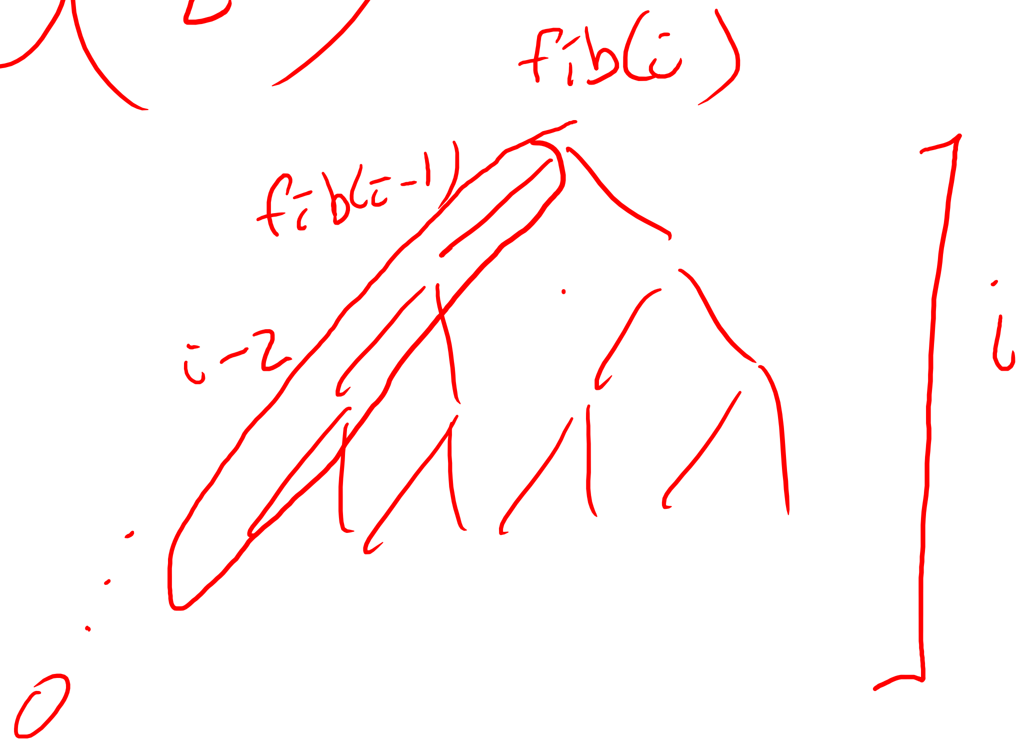
fib(0) = fib(1) = 1

Fibonacci Numbers

```
public int fib(int i) {
    int result;
    if (i < 2) {
        result = 1;
    } else {
        result = fibonacci(i - 1) + fibonacci(i - 2);
    }
    return result;
}
```

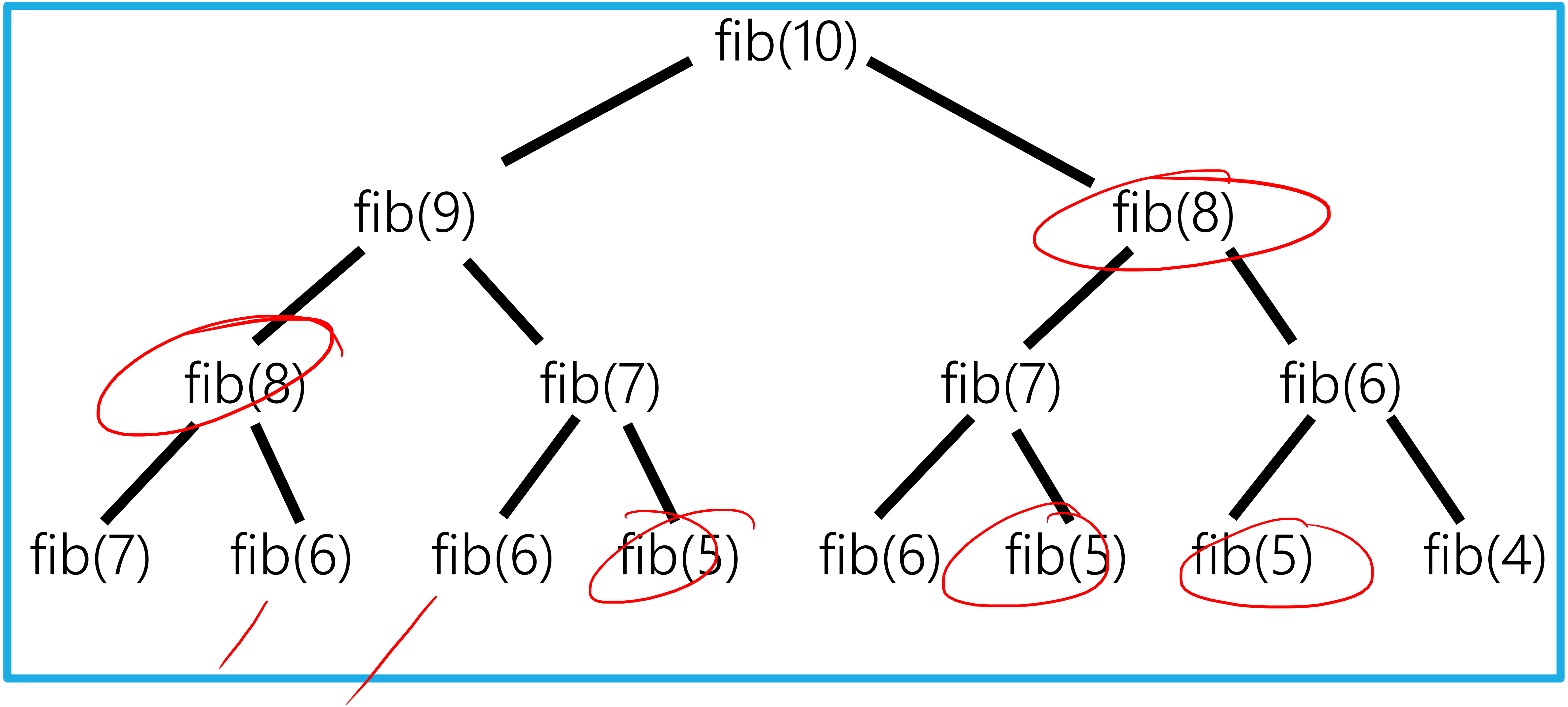
fib(0) = fib(1) = 1

$O(2^i)$



What is the runtime of this algorithm?

Fibonacci Numbers



Fibonacci Numbers: By Hand

n	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(n)	1	1	2	3	5	8	13	21	34	55	89	. .	.

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

merge sort isnt deterministic? 8/6 @ 28min

same arguments ALWAYS lead to same output

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):

O(g)

```
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];
}
```

If I have never computed function at this input before, compute it and put it in my table

O(g)

O(1) else if I have, just return the already computed value

trade off between space and time efficiency

Memoized Fibonacci

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

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:

problems like merge sort don't have unique subproblems —> list to be sorted changes each time

A F D E | G C B Q

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

If this is top-down, what is bottom up?

Top Down Evaluation

find what order table needs to be filled in to be most efficient, then just solve the subproblems in that order!

fib(12) (R → L)

Which order to we call fib(n) in? Top Down

Which order are the table cells filled in? Bottom up (L → R)

n	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(n)	1	1	2	3	5								

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!

fib 12
fib 10 + fib 11
fib 8 + fib 9   fib 9 + fib 10

first one to actually return a value, and thus set something in the array, is the lowest ones!
zero and 1

Fibonacci – Bottom Up

```
public int fib(int n) {
    fib = new int[n];        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];
}
```

Pros:	Cons
Runs faster Won't build up a huge call stack Easier to analyze runtime	More difficult to write

An Optimization

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

```
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];
}
```

throwing out information we dont need anymore!

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?

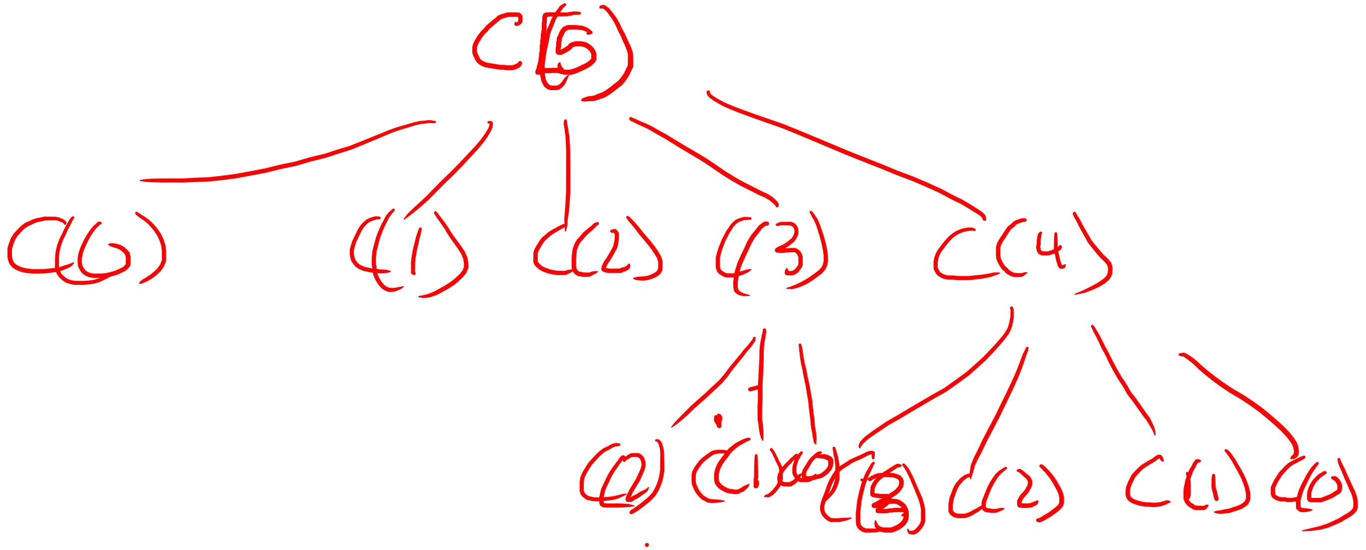
$$C(N) = \frac{2}{N}\sum_{i=0}^{N-1}\left(C(i)\right) + N$$

$$C(0) = 1$$

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?



wildily inefficient; we can see values of C(i) are calculated multiple times

This repetition of work hints at dynamic programming

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

With Dynamic Programming

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

$\boxed{C(0) = 1}$

$O(1)$

$C(5), C(4), C(5), C(3), \ldots, C(0)$

```
c = new ~~data~~[n+1]
c[0] = 1
for (i=1; i <= n; i++) {
    sum = 0
    for j=0 to N-1
        sum += c[j]
    c[i] = 2/N sum + N
}
return c[n]
```

More efficient algorithm:
have each index store C(n) + C(n-1) + C(n-2) + .. + C(0)

So then finding the next index i we would just have to do

C(i) + value at i - 1
= 2/i (value at i - 1) + i + value at i-1

This is also more space efficient because then our array only needs to be of size 2.

With Dynamic Programming

```
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!

are these sorted?

Problem: I have an unlimited set of coins of denomitations 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?

Can be tricky considering if the denominations do not line up nicely we cannot just look for biggest coins first. Example denominations 22, 17, 8, 1 : make change for 25 cents.

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.

Coin changing problem:

Text

assuming the denominations are in sorted order