// Our first attempt at computing Fibonacci numbers

public static int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

Fibonacci numbers

\[
\begin{align*}
F_0 &= 0, & F_1 &= 1, & F_n &= F_{n-1} + F_{n-2} \\
& & & \text{for all } n \geq 2.
\end{align*}
\]

Run time \( T(n) = T(n-1) + T(n-2) + 1 \)

\( T(n) \) is exponential in \( n \) so too slow.

[Diagram of Fibonacci sequence]
We compute $F_{n-1}$ $F_{n+1}$ times!

**Memoization:**

// Our second attempt: memoization
public static int fibTable = // initially set to all -1s

public static int fib(int n) {
    if (fibTable[n] == -1) {
        int fibNumber;
        if (n == 0) {
            fibNumber = 0;
        } else if (n == 1) {
            fibNumber = 1;
        } else {
            answer = fib(n - 1) + fib(n - 2);
            fibTable[n] = answer;
        }
    }
    return fibTable[n];
}

$fib(n)$ called twice
$\Rightarrow O(n)$ running time

We're computing all $fib(n)$
in order from 0 to n,

why not just use a
// Our third attempt: dynamic programming
public static int fib(int n) {
    int[] fibNumbers = new int[n + 1];
    // Fill in the base cases first.
    fibNumbers[0] = 0;
    fibNumbers[1] = 1;

    // Now fill in the rest in order.
    for (int i = 2; i <= n; i++) {
        fibNumbers[i] = fibNumbers[i - 1] + fibNumbers[i - 2];
    }

    // What were we here for again?
    return fibNumbers[n];
}

$\Theta(n)$ time.
$\Theta(n)$ space.
// Our final attempt: saving space
public static int fib(int n) {
    if (n == 0) {
        return 0;
    }

    int last = 1;
    int nextToLast = 0;
    int current = 1;
    for (int i = 2; i <= n; i++) {
        answer = last + nextToLast;
        nextToLast = last;
        last = answer;
    }
    return answer;
}

\( \Theta(n) \) time,
\( \Theta(1) \) space.
Ordering matrix multiplication

Have matrices

\[
\begin{array}{cccc}
A & B & C & D \\
50 \times 10 & 10 \times 40 & 40 \times 30 & 30 \times 5 \\
\end{array}
\]

Want to know product

\[
ABC \times D
\]

Matrix multiplication is associative. We can use any nesting of products of 2 matrices & get same result.
Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$: 

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:

Different parenthetizations of $ABCD$ give different total #s of scalar mults.

$(ACB)(CD)$: 10,500

$(ABC)(D)$: 87,500

$(CAB)(D)$:
Given $A_1, A_2, \ldots, A_n$, want to compute min scalar mults to evaluate $A_1 A_2 \cdots A_n$.

Really just given an array $c$, $c[i] = \# \text{columns in matrix } A_i$

$\# \text{rows in matrix } A_i$

$\# \text{rows in matrix } A_{i+1}$

$c[0] = \# \text{rows in } A_1$
Need a recursive solution. There is some final multiplication:

\[(A_1, A_2, \ldots, A_k)(A_{k+1}, \ldots, A_n)\]

If we know the best \( k \), we should minimize the scalar multiplications for \( A_1 \ldots A_k \) and \( A_{k+1} \ldots A_n \) individually.

So for all \( k \), do two recursive calls.

Recursive subproblems at
any level involve $A_i A_{i+1} \ldots A_j$
for some $i \leq j$.

$M(i, j)$: min # of scalar
mulTs for $A_i A_{i+1} \ldots A_j$.

$M(i, i)$: "product" $A_i$ requires $\Omega$ mulTs

If $j > i$?

If we knew the best
choice for $k$, $(A_\hat{i} \ldots A_k)(A_{k+1} \ldots A_j)$
$A_\hat{i} \ldots A_k$ uses $M(i, k)$ mulTs
$A_{k+1} \ldots A_j$ uses $M(k+1, j)$ mulTs
size of \( A_1 \ldots A_k \) is \( c[\omega - 1] \times c[k] \)

of \( A_k \ldots A_j \) is \( c[k] \times c[j] \)

last product in \( (A_1 \ldots A_k)(A_{k+1} \ldots A_j) \) uses \( c[\omega - 1] \cdot c[k] \cdot c[j] \) scalar multiplies

\[
M(\omega, j) = \min_{\omega \leq k < j} \left\{ M(\omega, k) + M(k + 1, j) + c[\omega - 1] \cdot c[j] \cdot c[k] \right\}
\]

if \( j > \omega \)
Min scalar mults for $A_1, A_2, \ldots, A_n$ is $M(1, n)$.

To fill a dynamic programming table, $i$ needs to decrease and $j$ needs to increase.

```java
public static void minScalarMults(int[] c) {
    int n = c.length - 1;
    long[][] m = new long[n + 1][n + 1];
    for (int i = n; i >= 1; i--) {
        m[i][i] = 0; // base case
        for (int j = i + 1; j <= n; j++) {
            m[i][j] = Long.MAX_VALUE;
            for (int k = i; k < j; k++) {
                int thisCost = m[i][k] + m[k + 1][j] + c[i - 1] * c[k] * c[j];
                if (thisCost < m[i][j]) { // need to update min
                    m[i][j] = thisCost;
                }
            }
        }
    }
    return m[1][n];
}
```

Runs in $\Theta(n^3)$. 
Dynamic Programming:

1) Find a recursive solution to your problem. Ideally use a recurrence with few subproblems.

2) Fill in a table (an array) with solutions to your recurrence in some good order.