Divide-and-Conquer

Application: Maxima Sets

- We can visualize the various trade-offs for optimizing two-dimensional data, such as points representing hotels according to their pool size and restaurant quality, by plotting each as a two-dimensional point, \((x, y)\), where \(x\) is the pool size and \(y\) is the restaurant quality score.
- We say that such a point is a maximum point in a set if there is no other point, \((x', y')\), in that set such that \(x \leq x'\) and \(y \leq y'\).
- The maximum points are the best potential choices based on these two dimensions and finding all of them is the maxima set problem.

We can efficiently find all the maxima points by divide-and-conquer. Here the set is \(\{A,H,I,G,D\}\.\)
Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - **Divide**: divide the input data $S$ into two or more disjoint subsets $S_1$, $S_2$, ...
  - **Conquer**: solve the subproblems recursively
  - **Combine**: combine the solutions for $S_1$, $S_2$, ..., into a solution for $S$

- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations

Solving the Maxima Set Problem

- A point is a **maximum point** in $S$ if there is no other point, $(x', y')$, in $S$ such that $x \leq x'$ and $y \leq y'$.
- To find a **maxima set** for a set, $S$, of $n$ points in the plane, we may divide $S$ into two equal parts.
- We compare two points in $S$ using a lexicographic ordering of the points in $S$, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties.
Divide-and-Conquer Solution

- **Base case:** If n ≤ 1, the maxima set is just S itself.
- **Divide:** let p = (x_p, y_p) be the median point in S according to the lexicographic order. Then x = x_p is a line dividing S into two halves.
- **Conquer:** we recursively solve the maxima-set problem for the set of points on the left of this line and also for the points on the right.
- **Combine:**
  - The maxima set of points on the right are also maxima points for S.
  - But some of the maxima points for the left set might be dominated by a point from the right, namely the point, q, that is leftmost.
  - So then we do a scan of the left set of maxima, removing any points that are dominated by q.
  - The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S.

Example for the Combine Step
There is the issue of how to efficiently find the point, \( p \), that is the median point in a lexicographical ordering of the points in \( S \) according to their \((x, y)\)-coordinates.

There are two immediate possibilities:

- One choice is to use a linear-time median-finding algorithm, such as that given in Section 9.2. \( O(n) \) for each recursive call.
- Another choice is to sort the points in \( S \) lexicographically by their \((x, y)\)-coordinates as a preprocessing step, prior to calling the MaximaSet algorithm on \( S \). \( O(n \log(n)) \) for preprocessing and \( O(1) \) for each recursive call, to find the middle of the list.

### Pseudo-code

```plaintext
Algorithm MaximaSet(S):
    Input: A set, \( S \), of \( n \) points in the plane
    Output: The set, \( M \), of maxima points in \( S \)
    if \( n \leq 1 \) then
        return \( S \)
    Let \( p \) be the median point in \( S \), by lexicographic \((x, y)\)-coordinates
    Let \( L \) be the set of points lexicographically less than \( p \) in \( S \)
    Let \( G \) be the set of points lexicographically greater than or equal to \( p \) in \( S \)
    \( M_1 \leftarrow \) MaximaSet(\( L \))
    \( M_2 \leftarrow \) MaximaSet(\( G \))
    Let \( q \) be the lexicographically smallest point in \( M_2 \)
    for each point, \( r \), in \( M_1 \) do
        if \( x(r) < x(q) \) and \( y(r) \leq y(q) \) then
            Remove \( r \) from \( M_1 \)
    return \( M_1 \cup M_2 \)
```
Analysis

- In either case, the rest of the non-recursive steps can be performed in \( O(n) \) time, so this implies that, ignoring floor and ceiling functions, the running time for the divide-and-conquer maxima-set algorithm can be specified as follows (where \( b \) is a constant):

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

- Thus, according to the merge sort example, this algorithm runs in \( O(n \log n) \) time.

Iterative Substitution

- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

\[
T(n) = 2(2T(n/2^i) + b(n/2)) + bn
\]

\[
= 2^2T(n/2^i) + 2bn
\]

\[
= 2^3T(n/2^i) + 3bn
\]

\[
= 2^4T(n/2^i) + 4bn
\]

... 

\[
= 2^iT(n/2^i) + ibn
\]

- Note that base, \( T(n) = b \), case occurs when \( 2^i = n \). That is, \( i = \log n \).

- So,

\[
T(n) = bn + bn \log n
\]

- Thus, \( T(n) \) is \( O(n \log n) \).
The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

<table>
<thead>
<tr>
<th>depth</th>
<th>T's size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
<td>bn</td>
</tr>
<tr>
<td>1</td>
<td>n/2</td>
<td>bn</td>
</tr>
<tr>
<td>i</td>
<td>n/2^i</td>
<td>bn</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Total time = \( bn + bn \log n \)
(last level plus all previous levels)

Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

- Guess: \( T(n) \leq cn \log n \).

\[
T(n) = 2T(n/2) + bn \\
\leq 2(c(n/2)\log(n/2)) + bn \\
= cn(\log n - \log 2) + bn \\
= cn \log n - bn + bn \\
= cn \log n - (c-b)n 
\]

- We can conclude that ne be less than \( T(n) \leq cn \log n \) if \( c \geq b \).
Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:
  \[ T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
  \end{cases} \]

- Guess: \( T(n) \leq cn \log n \).
  \[
  T(n) = 2T(n/2) + bn \log n \\
  \leq 2(c(n/2) \log(n/2)) + bn \log n \\
  = cn(\log n - \log 2) + bn \log n \\
  = cn \log n - cn + bn \log n 
  \]

- Wrong: we cannot make this last line be less than \( cn \log n \).

Guess-and-Test Method, (cont.)

- Recall the recurrence equation:
  \[
  T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
  \end{cases} \]

- Guess #2: \( T(n) \leq cn \log^2 n \).
  \[
  T(n) = 2T(n/2) + bn \log n \\
  \leq 2(c(n/2) \log^2(n/2)) + bn \log n \\
  = cn(\log n - \log 2)^2 + bn \log n \\
  = cn \log^2 n - 2cn \log n + cn + bn \log n \\
  \leq cn \log^2 n - cn \log n + cn + bn \log n \\
  \leq cn \log^2 n \text{ if } c \geq b. 
  \]

- So, \( T(n) \) is \( O(n \log^2 n) \).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.
Master Method

- Many divide-and-conquer recurrence equations have the form:
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. if \( f(n) \) is \( O(n^\log_b a) \), then \( T(n) \) is \( \Theta(n^\log_b a) \)
  2. if \( f(n) \) is \( \Theta(n^\log_b a \log^k n) \), then \( T(n) \) is \( \Theta(n^\log_b a \log^{k+1} n) \)
  3. if \( f(n) \) is \( \Omega(n^\log_b a + \varepsilon) \), then \( T(n) \) is \( \Theta(f(n)) \),
     provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Master Method, Example 1

- The form:
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. if \( f(n) \) is \( O(n^\log_b a - \varepsilon) \), then \( T(n) \) is \( \Theta(n^\log_b a) \)
  2. if \( f(n) \) is \( \Theta(n^\log_b a \log^k n) \), then \( T(n) \) is \( \Theta(n^\log_b a \log^{k+1} n) \)
  3. if \( f(n) \) is \( \Omega(n^\log_b a + \varepsilon) \), then \( T(n) \) is \( \Theta(f(n)) \),
     provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 4T(n/2) + n \]

  Solution: \( \log_b a = 2 \), so case 1 says \( T(n) \) is \( O(n^2) \).
Master Method, Example 2

- The form: \( T(n) = \begin{cases} \ c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases} \)

- The Master Theorem:
  1. if \( f(n) = O(n^{\log_{b}a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_{b}a}) \)
  2. if \( f(n) = \Theta(n^{\log_{b}a \log^{k}n}) \), then \( T(n) = \Theta(n^{\log_{b}a \log^{k+1}n}) \)
  3. if \( f(n) = \Omega(n^{\log_{b}a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = 2T(n/2) + n \log n \]

Solution: \( \log_{b}a = 1 \), so case 2 says \( T(n) = O(n \log^{2}n) \).

Master Method, Example 3

- The form: \( T(n) = \begin{cases} \ c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases} \)

- The Master Theorem:
  1. if \( f(n) = O(n^{\log_{b}a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_{b}a}) \)
  2. if \( f(n) = \Theta(n^{\log_{b}a \log^{k}n}) \), then \( T(n) = \Theta(n^{\log_{b}a \log^{k+1}n}) \)
  3. if \( f(n) = \Omega(n^{\log_{b}a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- Example:
  \[ T(n) = T(n/3) + n \log n \]

Solution: \( \log_{b}a = 0 \), so case 3 says \( T(n) = O(n \log n) \).
Master Method, Example 4

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = O(n^{log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{log_b a}) \)
  2. If \( f(n) = \Theta(n^{log_b a} \log^k n) \), then \( T(n) = \Theta(n^{log_b a} \log^{k+1} n) \)
  3. If \( f(n) = \Omega(n^{log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \)

- Example:
  \[ T(n) = 8T(n/2) + n^2 \]
  Solution: \( \log_2 a = 3 \), so case 1 says \( T(n) \) is \( O(n^3) \).

Master Method, Example 5

- The form: 
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- The Master Theorem:
  1. If \( f(n) = O(n^{log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{log_b a}) \)
  2. If \( f(n) = \Theta(n^{log_b a} \log^k n) \), then \( T(n) = \Theta(n^{log_b a} \log^{k+1} n) \)
  3. If \( f(n) = \Omega(n^{log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \)

- Example:
  \[ T(n) = 9T(n/3) + n^3 \]
  Solution: \( \log_3 a = 2 \), so case 3 says \( T(n) \) is \( O(n^3) \).
Master Method, Example 6

- **The form:**
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- **The Master Theorem:**
  1. if \( f(n) = O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. if \( f(n) = \Theta(n^{\log_b a \log^k n}) \), then \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \)
  3. if \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \),
     provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- **Example:**
  \[ T(n) = T(n/2) + 1 \] (binary search)

  Solution: \( \log_b a = 0 \), so case 2 says \( T(n) = O(\log n) \).

Master Method, Example 7

- **The form:**
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- **The Master Theorem:**
  1. if \( f(n) = O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
  2. if \( f(n) = \Theta(n^{\log_b a \log^k n}) \), then \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \)
  3. if \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \),
     provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

- **Example:**
  \[ T(n) = 2T(n/2) + \log n \] (heap construction)

  Solution: \( \log_b a = 1 \), so case 1 says \( T(n) = O(n) \).
### Integer Addition

- **Addition.** Given two \(n\)-bit integers \(a\) and \(b\), compute \(a + b\).
- **Grade-school.** \(\Theta(n)\) bit operations.

![Addition Example](image)

Remark. Grade-school addition algorithm is optimal.

### Integer Multiplication

- **Multiplication.** Given two \(n\)-bit integers \(a\) and \(b\), compute \(a \times b\).
- **Grade-school.** \(\Theta(n^2)\) bit operations.

![Multiplication Example](image)

Q. Is grade-school multiplication algorithm optimal?
Divide-and-Conquer Multiplication: Warmup

- To multiply two $n$-bit integers $a$ and $b$:
  - Multiply four $\frac{n}{2}$-bit integers, recursively.

\[
a = 2^{\frac{n}{2}} \cdot a_1 + a_0 \\
b = 2^{\frac{n}{2}} \cdot b_1 + b_0 \\
ab = \left(2^{\frac{n}{2}} \cdot a_1 + a_0\right)\left(2^{\frac{n}{2}} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{\frac{n}{2}} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0
\]

- Ex. $a = 10001101$ $b = 11100001$

Divide-and-Conquer Multiplication:

- $T(n) = 4T(n/2) + \Theta(n)$ => $T(n) = \Theta(n^2)$

- $T(n) = \sum_{k=0}^{\log n} 2^k = n \left(\frac{2^{\log n + 1} - 1}{2 - 1}\right) = 2n^2 - n$

- $T(n) = \begin{cases} 
0 & \text{if } n = 0 \\
4T(n/2) + n & \text{otherwise}
\end{cases}$
Karatsuba Multiplication

- To multiply two $n$-bit integers $a$ and $b$:
  - Add two $\frac{1}{2}n$ bit integers.
  - Multiply three $\frac{1}{2}n$-bit integers, recursively.
  - Add, subtract, and shift to obtain result.

\[
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n/2} \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 \\
    &= 2^{n/2} \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0
\end{align*}
\]

\[T(n) \leq \underbrace{T(\lfloor n/2 \rfloor)}_{\text{recursive calls}} + \underbrace{T(\lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{T(1)}_{\text{add, subtract, shift}} + \underbrace{\Theta(a)}_{\text{add, subtract, shift}} \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})\]
Dot Product

Dot product. Given two length $n$ vectors $a$ and $b$, compute $c = a \cdot b$.

Grade-school. $\Theta(n)$ arithmetic operations.

\[ a \cdot b = \sum_{i=1}^{n} a_i b_i \]

Remark. Grade-school dot product algorithm is optimal.

Matrix Multiplication

Matrix multiplication. Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \]

Q. Is grade-school matrix multiplication algorithm optimal?
**Block Matrix Multiplication**

**Matrix Multiplication: Warmup**

To multiply two $n$-by-$n$ matrices $A$ and $B$:
- **Divide**: partition $A$ and $B$ into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- **Conquer**: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- **Combine**: add appropriate products using 4 matrix additions.

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

$$T(n) = 8T(n/2) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^3)$$
Fast Matrix Multiplication

**Key idea.** multiply 2-by-2 blocks with only 7 multiplications.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \times \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

| \(C_{11}\) | \(= P_3 + P_4 - P_2 + P_6\) |
| \(C_{12}\) | \(= P_1 + P_2\) |
| \(C_{21}\) | \(= P_3 + P_4\) |
| \(C_{22}\) | \(= P_3 + P_1 - P_5 - P_7\) |

- 7 multiplications.
- 18 = 8 + 10 additions and subtractions.

---

**Fast Matrix Multiplication**

To multiply two \(n\)-by-\(n\) matrices \(A\) and \(B\): [Strassen 1969]

- Divide: partition \(A\) and \(B\) into \(\frac{1}{2}n\)-by-\(\frac{1}{2}n\) blocks.
- Compute: 14 \(\frac{1}{2}n\)-by-\(\frac{1}{2}n\) matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of \(\frac{1}{2}n\)-by-\(\frac{1}{2}n\) matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

**Analysis.**

- Assume \(n\) is a power of 2.
- \(T(n)\) = # arithmetic operations.

\[
T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{\approx 2.81})
\]
Fast Matrix Multiplication

To multiply two $n$-by-$n$ matrices $A$ and $B$: [Strassen 1969]

\[
T(n) = 7T(n/2) + \Theta(n^2) = T(n) = \Theta(n\log^7 n) = O(n^{2.81})
\]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Multiplications</th>
<th>Additions</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional alg.</td>
<td>$n^3$</td>
<td>$n^3 - n^2$</td>
<td>$\Theta(n^3)$</td>
</tr>
<tr>
<td>Recursive version</td>
<td>$n^3$</td>
<td>$n^3 - n^2$</td>
<td>$\Theta(n^3)$</td>
</tr>
<tr>
<td>Strassen’s alg.</td>
<td>$n^{\log_2 7}$</td>
<td>$6n^{\log_2 7} - 6n^2$</td>
<td>$\Theta(n^{\log_2 7})$</td>
</tr>
</tbody>
</table>

Table 6.2  The number of arithmetic operations done by the three algorithms.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Multiplications</th>
<th>Additions</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1,000,000</td>
<td>990,000</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>411,822</td>
<td>2,470,334</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1,000,000,000</td>
<td>999,000,000</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>264,280,285</td>
<td>1,579,681,709</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>10^{12}</td>
<td>9.99 \times 10^{12}</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>0.169 \times 10^{12}</td>
<td>10^{12}</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3 Comparison between Strassen’s algorithm and the traditional algorithm.

Fast Matrix Multiplication: Practice

Implementation issues.
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception. “Strassen is only a theoretical curiosity.”
- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2,500$.
- Range of instances where it’s useful is a subject of controversy.

Remark. Can "Strassenize" $Ax = b$, determinant, eigenvalues, ....
Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969] \( \Theta(n^{\log_2 7}) = O(n^{2.807}) \)

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971] \( \Theta(n^{\log_2 6}) = O(n^{2.59}) \)

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible. \( \Theta(n^{19/21}) = O(n^{2.27}) \)

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]
- Two 20-by-20 matrices with 4,460 scalar multiplications. \( O(n^{2.315}) \)
- Two 48-by-48 matrices with 47,217 scalar multiplications. \( O(n^{2.701}) \)
- A year later.

Best known. \( O(n^{2.376}) \) [Coppersmith-Winograd, 1987]

Conjecture. \( O(n^{2+\epsilon}) \) for any \( \epsilon > 0 \).

Caveat. Theoretical improvements to Strassen are progressively less practical.
Exercise

Given a $m \times m$ matrix $M$ and a positive integer $n$, how to compute $M^n$ efficiently?