Objectives

- Divide and conquer
  - Integer multiplication
  - Matrix multiplication
- Introduction to Dynamic Programming
  - Fibonacci

INTEGER AND MATRIX MULTIPLICATION

Integer Arithmetic

- Add. Given 2 *n*-digit integers a and b, compute a + b.
  - Algorithm?
  - Runtime?

```
  1 1 1 1 1 1 0 1
    1 1 0 1 0 1 0 1
+   0 1 1 1 1 1 0 1
  1 0 1 0 1 0 0 1 0
```


Integer Arithmetic

- Add. Given 2 *n*-digit integers a and b, compute a + b.
  - Algorithm?
  - Runtime?

```
  1 1 1 1 1 1 0 1
    1 1 0 1 0 1 0 1
+   0 1 1 1 1 1 0 1
  1 0 1 0 1 0 0 1 0
```

O(n) operations

Integer Arithmetic

- Multiply. Given 2 *n*-digit integers a and b, compute a × b.

Algorithm?

Runtime?

```
  1 1 0 1 0 1 0 1
* 0 1 1 1 1 1 0 1
```


Integer Arithmetic

- Multiply. Given 2 *n*-digit integers a and b, compute a × b.
  - Brute force solution: $\Theta(n^2)$ bit operations

Goal: Faster algorithm

```
  1 1 0 1 0 1 0 1
* 0 1 1 1 1 1 0 1
  1 1 0 1 0 1 0 1
  0 0 0 0 0 0 0 0
    1 1 0 1 0 1 0 1
      1 1 0 1 0 1 0 1
        1 1 0 1 0 1 0 1
          1 1 0 1 0 1 0 1
            1 1 0 1 0 1 0 1
  0 0 0 0 0 0 0 0
0 1 1 0 1 0 0 0 0 0 0 0 0 1 0
```

Divide-and-Conquer Multiplication: Warmup

- To multiply 2 n-digit integers:
  - Multiply 4 ½n-digit integers
  - Add 2 ½n-digit integers and shift to obtain result

Higher order bits Lower order bits x=10001101 x_1=1000 x_0=1101

Shift

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \end{aligned}$$

A B C D

What is the recurrence relation?
- How many subproblems?
- What is merge cost?
- What is its runtime?

Divide-and-Conquer Multiplication: Warmup

- To multiply 2 n-digit integers:
  - Multiply 4 ½n-digit integers
  - Add 2 ½n-digit integers and shift to obtain result

Higher order bits Lower order bits

Shift

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \end{aligned}$$

A B C D

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$

assumes n is a power of 2

Not an improvement over brute force

Karatsuba Multiplication

- To multiply two n-digit integers:
  - Add 2 ½n digit integers
  - Multiply 3 ½n-digit integers
  - Add, subtract, and shift ½n-digit integers to obtain result

Anatolii Alexeevich Karatsuba

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \\ &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0\right) + x_0 y_0 \end{aligned}$$

A B A C C

What is the recurrence relation? Runtime?

Karatsuba Multiplication

- Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \\ &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0\right) + x_0 y_0 \end{aligned}$$

A B A C C

$$T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}$$

$$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

MATRIX MULTIPLICATION

Matrix Multiplication

- Given 2 n-by-n matrices A and B, compute C = AB

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

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

- Ex: $c_{12} = a_{11} b_{12} + a_{12} b_{22} + a_{13} b_{32} + \ldots + a_{1n} b_{n2}$

Row 1 of a Column 2 of b

Solve using brute force …

Matrix Multiplication

- Given 2 n-by-n matrices A and B, compute C = AB

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

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

  - Ex: $c_{12} = a_{11} b_{12} + a_{12} b_{22} + a_{13} b_{32} + \ldots + a_{1n} b_{n2}$

- Brute force. $\Theta(n^3)$ arithmetic operations
- Fundamental question: Can we improve upon brute force?

Matrix Multiplication: Warmup

- Divide: partition A and B into ½n-by-½n blocks
- Conquer: multiply 8 ½n-by-½n recursively
- Combine: add appropriate products using 4 matrix additions

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

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

Recurrence relation? Runtime?

Matrix Multiplication: Warmup

- Divide: partition A and B into ½n-by-½n blocks
- Conquer: multiply 8 ½n-by-½n recursively
- Combine: add appropriate products using 4 matrix additions

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

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

Trade expensive multiplication for less expensive addition/subtraction

- Multiply 2-by-2 block matrices with only 7 multiplications and 15 additions

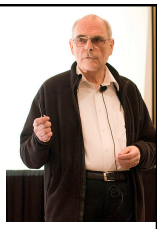
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

Fast Matrix Multiplication [Strassen, 1969]

- Divide: partition A and B into ½n-by-½n blocks
- Compute: 14 ½n-by-½n matrices via 10 matrix additions
- Conquer: multiply 7 ½n-by-½n matrices recursively
- Combine: 7 products into 4 terms using 8 matrix additions
- Analysis.
  - Assume n is a power of 2.
  - T(n) = # arithmetic operations.



Volker Strassen

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Practice

- Implementation issues: problems putting theory into practice
  - Sparsity
  - Caching effects
  - Numerical stability
    - Theoretically correct but possible problems with round off errors, etc
  - Odd matrix dimensions
- Crossover to classical algorithm around n = 128

Fast Matrix Multiplication in Practice

- Common misperception: "Strassen is only a theoretical curiosity."
  - Advanced Computation Group at Apple Computer reports **8x** speedup on G4 Velocity Engine when n ~ 2,500
  - Range of instances where it's useful is a subject of controversy
- Can "Strassenize" Ax=b, determinant, eigenvalues, and other matrix ops

Fast Matrix Multiplication in Theory

- Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?
- A. Yes! [Strassen, 1969] $\Theta(n^{\log_2 7}) = O(n^{2.81})$
- Q. Multiply two 2-by-2 matrices with only 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 only 21 scalar multiplications?
- A. Also impossible $\Theta(n^{\log_3 21}) = O(n^{2.77})$
- Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?
- A. Yes! [Pan, 1980] $\Theta(n^{\log_{70} 143640}) = O(n^{2.80})$

- Decimal wars.
  - December, 1979: $O(n^{2.521813})$
  - January, 1980: $O(n^{2.521801})$

Fast Matrix Multiplication in Theory

- Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]
  - But *really* large constant
- Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.
- Caveat. Theoretical improvements to Strassen are progressively less practical.

Algorithmic Paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion
- Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem
- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems

Dynamic Programming History

- Richard Bellman pioneered systematic study of dynamic programming in 1950s
- Etymology
  - Dynamic programming = planning over time
    - Not our typical use of "programming"
  - Then-Secretary of Defense was hostile to mathematical research
  - Bellman sought an impressive name to avoid confrontation
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

WARMUP: FIBONACCI SEQUENCE

How Would You Solve the Fibonacci Sequence?

- Input: the number of Fibonacci numbers, x
- Output: display the list of the first x Fibonacci numbers

Sequence:

- $F_0=F_1=1$
- $F_n=F_{n-1}+F_{n-2}$

Soln 1: Using a List

- Typical Solution:

```
fibs = []               # create an empty list
fibs.append(1)          # append the first two Fib numbers
fibs.append(1)
print fibs[0], fibs[1],
for x in xrange(2, N):
        newfib = fibs[x-1]+fibs[x-2]
        print newfib,
        fibs.append(newfib)

print fibs              # print out the list
```

Building up solution

Running time? Space cost?

Do we need a whole list?

Soln 2: Using Three Variables

- Only need the solutions to the last two problems (F[k-1], F[k-2])

```
lastNum = 1
twoAgo = 1
print twoAgo, lastNum,

for n in xrange(2, N):

    nthNum = twoAgo + lastNum
    print nthNum,

    twoAgo = lastNum
    lastNum = nthNum
```


Soln 3: Recursion

```
def fibonacci(n):
   return fibonacci(n-1) + fibonacci(n-2)
```

- What is the running time of this algorithm?

Dynamic Programming **Memoization** Process

- Create a table with the possible inputs
- If the value is in the table, return it, without recomputing it
- Otherwise, call function recursively
  - Add value to table for future reference

How can we apply this template to our Fibonnaci problem?

Memoization Example: Fibonacci

```
memoized_fibonacci(n):
    for j = 1 to n:
       results[i] = -1     # -1 means undefined

    return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
    if results[n] != -1:   # value is defined
       return results[n]
    if n == 1:
       val = 1
    elif n == 2:
       val = 1
    else:
       val = memoized_fib_recurs(results, n-2)
       val = val + memoized_fib_recurs(results, n-1)
    results[n] = val
    return val
```

Runtime?

O(n)

Memoization Example: Fibonacci

Alternative version…

```
memoized_fibonacci(n):
    for j = 1 to n:
       results[i] = -1 # -1 means undefined
    results[1] = 1
    results[2] = 1

    return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
    if results[n] != -1: # value is defined
       return results[n]

    val = memoized_fib_recurs(results, n-2)
    val = val + memoized_fib_recurs(results, n-1)
    results[n] = val
    return val
```


Looking Ahead

- Wiki due Tuesday
  - Chapter 5.3, 5.4, 5.5
- PS7 due Friday
- SSA EC soon
- Exam 2 handed out Friday