

Lecture 4: Recursive Algorithms: Fast Multiplication; Fast Matrix Multiplication

1 Multiplication

- Grade School Multiplication
 - Perform n^2 multiplications to multiply two n-digit numbers
→ Long multiplication has running time $\Omega(n^2)$.
- Russian Peasant Multiplication
Mult(x, y)
If x = 1 **then** output y
If x is odd **then** output y + Mult($\lfloor \frac{x}{2} \rfloor$, 2y)
If x is even **then** output Mult($\frac{x}{2}$, 2y)

Binary Representation	x	y
1 0	46	324
2 1	23	648
4 1	11	1296
8 1	5	2592
16 0	2	5184
32 1	1	10368
46		14904

- This method does work. Base case ($x = 1$) is verified. The step when x is even also works. The only tricky step is that x is odd when you divide x by 2, you actually divide x by 2 and minus a half, so that you need to add one y back.
- How long does it take?

	Binary Representation	x	y
	1 0	46	324 ← # Bits(y)
	2 1	23	648
	4 1	11	1296
	8 1	5	2592
	16 0	2	5184
	32 1	1	10366 ← # Bits(y) + # Bits(x)
			14904

- There are n iterations, so we add up to n numbers with at most $2n$ digits each.
- Runtime = $O(n^2)$

2 Divide and Conquer Multiplication

○ Let:

$$\mathbf{x} = \underbrace{x_n x_{n-1} \cdots x_{\frac{n}{2}+1}}_{\mathbf{x}_L} \underbrace{x_{\frac{n}{2}} \cdots x_2 x_1}_{\mathbf{x}_R} \quad \text{e.g. } \mathbf{x} = 4132$$

$$\mathbf{y} = \underbrace{y_n y_{n-1} \cdots y_{\frac{n}{2}+1}}_{\mathbf{y}_L} \underbrace{y_{\frac{n}{2}} \cdots y_2 y_1}_{\mathbf{y}_R} \quad \mathbf{y} = 6703$$

○ Then:

$$\mathbf{x} = 10^{\frac{n}{2}} \cdot \mathbf{x}_L + \mathbf{x}_R \quad \mathbf{x} = 41 \cdot 10^2 + 32$$

$$\mathbf{y} = 10^{\frac{n}{2}} \cdot \mathbf{y}_L + \mathbf{y}_R \quad \mathbf{y} = 67 \cdot 10^2 + 03$$

○ Thus:

$$\begin{aligned} \mathbf{x} \cdot \mathbf{y} &= (10^{\frac{n}{2}} \cdot \mathbf{x}_L + \mathbf{x}_R) \cdot (10^{\frac{n}{2}} \cdot \mathbf{y}_L + \mathbf{y}_R) \\ &= 10^n \cdot \mathbf{x}_L \mathbf{y}_R + 10^{\frac{n}{2}} \cdot (\mathbf{x}_L \mathbf{y}_R + \mathbf{x}_R \mathbf{y}_L) + \mathbf{x}_R \mathbf{y}_R \end{aligned}$$

\Rightarrow Multiplying involves four products with $\frac{n}{2}$ -digit numbers!

- How long does this algorithm take?
 - The recursive formula for the running time is:

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Note: by padding some zeroes in front of the number, we can assume n is a power of 2.
 - Thus we have $a = 4$, $b = 2$, and $d = 1$.
 - This is Case 3 of the Master Theorem.
 - Runtime = $O(n^{\log_2 4}) = O(n^2)$

3 Gauss's Complex Number Multiplication

- Gauss considered the product of complex numbers:

$$(a + b \cdot i) \cdot (c + d \cdot i) = ac - bd + (bc + ad) \cdot i$$

- This seems to require taking **four products**, but he observed that:

$$(bc + ad) = (a + b) \cdot (c + d) - ac - bd$$

- So we can calculate ac and bd then we only need to perform only one more product to find $(bc + ad)$, namely $(a + b) \cdot (c + d)$.
 - Multiplying two complex numbers involves only **three products**! We got extra addition here, but addition is much cheaper than multiplication when we think about the running time.

4 Application of Gauss's Complex Number Multiplication

- We can simply replace i by $10^{\frac{n}{2}}$

$$(a + b \cdot i) \cdot (c + d \cdot i) = ac - bd + (bc + ad) \cdot i$$

$$(x_R + x_L \cdot 10^{\frac{n}{2}}) \cdot (y_R + y_L \cdot 10^{\frac{n}{2}}) = x_R y_R + x_L y_L \cdot 10^n + (x_L y_R + x_R y_L) \cdot 10^{\frac{n}{2}}$$
 ○ But now we have:

$$(x_L y_R + x_R y_L) = (x_R + x_L) \cdot (y_R + y_L) - x_R y_R - x_L y_L$$

$$(x_L y_R + x_R y_L) = x_R y_R + x_L y_L - (x_R - x_L) \cdot (y_R - y_L)$$

- Multiplying involves only **three products** with $\frac{n}{2}$ digit numbers!
- The recursive formula for the running time is:

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$
- Thus we have $a = 3$, $b = 2$, and $d = 1$.

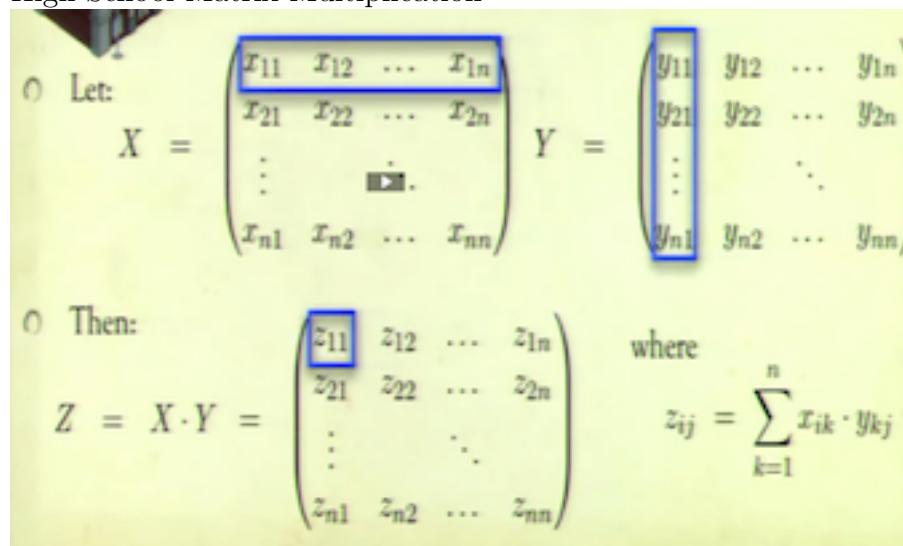
- This is Case 3 of the Master Theorem.
- Runtime = $O(n^{\log_2 3}) = O(n^{1.59})$
- So we can multiply two n-bit numbers using less than n^2 operations!

5 Fast Fourier Transforms

- We can multiply two n-bit numbers in time $O(n \cdot \log n)$ using a **Fast Fourier Transform**.
- More generally, FFTs can be used to multiply two **polynomial functions**.
 - This has a vast number of applications, for example in *image compression* and *signal processing*.
 - Indeed, it has been described by Strang as "the most important numerical algorithm of our lifetime."
- We might study FFTs later in the course.

6 Matrix Multiplication

- High School Matrix Multiplication



Let:

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & \ddots & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} \quad Y = \begin{pmatrix} y_{11} & y_{12} & \dots & y_{1n} \\ y_{21} & y_{22} & \dots & y_{2n} \\ \vdots & & \ddots & \\ y_{n1} & y_{n2} & \dots & y_{nn} \end{pmatrix}$$

Then:

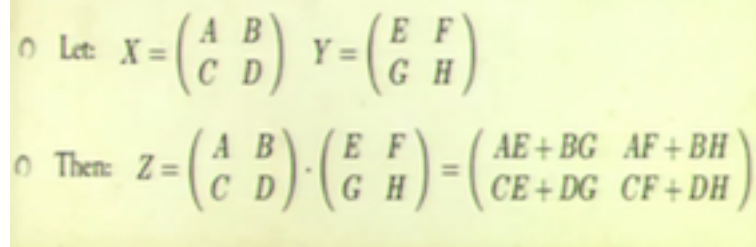
$$Z = X \cdot Y = \begin{pmatrix} z_{11} & z_{12} & \dots & z_{1n} \\ z_{21} & z_{22} & \dots & z_{2n} \\ \vdots & & \ddots & \\ z_{n1} & z_{n2} & \dots & z_{nn} \end{pmatrix} \quad \text{where} \quad z_{ij} = \sum_{k=1}^n x_{ik} \cdot y_{kj}$$

Then we perform n multiplications to calculate each entry of Z.

- Runtime = $\Omega(n^3)$

7 Divide and Conquer Matrix Multiplication

- We can also multiply matrices using **divide and conquer**.



o Let: $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$

o Then: $Z = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$

- Multiplying involves **eight products** with $\frac{n}{2} \times \frac{n}{2}$ matrices!
- The recursive formula for the running time is:
$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$
- Thus we have a = 8, b = 2, and d = 2.
- This is Case 3 of the Master Theorem.
- Runtime = $O(n^{\log_2 8}) = O(n^3)$
- No improvement!

8 An Algebraic Trick for Matrix Multiplication

○ Let:

$$Z = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

$$= \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

$$= \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

where

$$\begin{aligned} S_1 &= (B - D) \cdot (G + H) \\ S_2 &= (A + D) \cdot (E + H) \\ S_3 &= (A - C) \cdot (E + F) \\ S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) \\ S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

- Multiplying involves **seven products** with $\frac{n}{2} \times \frac{n}{2}$ matrices!
Note: Even though we now have 18 additions, the runtime is still dominated by the number of multiplications. Having additions would not hurt.
- The recursive formula for the running time is:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$
- Thus we have $a = 7$, $b = 2$, and $d = 2$.
- This is Case 3 of the Master Theorem.
- Runtime = $O(n^{\log_2 7}) = O(n^{2.81})$
- This divide and conquer matrix multiplication algorithm was designed by Strassen (1969). Since then faster algorithms (in theory) have been developed:
 - **Theorem:** There is a matrix multiplication algorithm that runs in time $O(n^{2.37})$
 - **Open Problem:** Is matrix multiplication $O(n^{2+\epsilon})$ time for any constant $\epsilon > 0$? Note: There is no way you can beat this because just to read the matrix x you have to do x^2 amount of work to read entries in x simply for y . Getting running time like this seems crazy but possible.

9 Fast Exponentiation

- Suppose we want to compute x^n
 - The slow way $x \cdot x \cdot \dots \cdot x$ requires $n-1$ multiplications.
 - A faster way is to use the fact that $x^n = x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor}$
- The latter method gives the following **recursive algorithm**.

Fast Exponentiation

FastExp(x, n)

If $n = 1$ **then** output x

Else

If n is even **then** output **FastExp**($x, \lfloor \frac{n}{2} \rfloor$)²

If n is odd **then** output $x \cdot \mathbf{FastExp}(x, \lfloor \frac{n}{2} \rfloor)^2$

- The number of multiplications used in Fast Exponentiation satisfies:

$$T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + 2 \Rightarrow T(n) = T(\frac{n}{2}) + O(1)$$

- Thus we have $a = 1$, $b = 2$, and $d = 0$.
- This is Case 2 of the Master Theorem.
- Runtime = $O(n^0 \cdot \log n) = O(\log n)$