Algebraic Algorithms

Self-plug: lecture notes (in Hebrew) from an earlier class available at:
https://benleevolk.bitbucket.io/pdf/algcomplecturenotes.pdf

Algebraic Model of Computation

- We'll consider algorithms that can perform arithmetic operations (addition, subtraction, multiplication, division) at unit cost
- This is true regardless of the input (integers, rationals, irrational numbers, complex numbers)
- We won't care (for the most part) how the input is represented and only count arithmetic operations

Matrix multiplication

- Inputs: two $n \times n$ matrices A, B
- Output: The matrix C which equals $A \cdot B$
- How many operations are needed?
- Classical algorithm: $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$
  - Each entry requires n multiplications and n additions
  - n^2 entries for a total of $O(n^3)$ operations
- Can we do better?
- (Note: matrix addition can be done in time $O(n^2)$)

Strassen's Algorithm

- Strassen showed in 1969 that we can
- Consider first 2×2 matrix multiplication

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

- Strassen found a way to compute this with 7 multiplications instead of 8 (but with more additions and subtractions)

Strassen's Algorithm

$$
\begin{aligned}
p_1 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \\
p_2 &= (a_{21} + a_{22}) \cdot b_{11} \\
p_3 &= a_{11} \cdot (b_{12} - b_{22}) \\
p_4 &= a_{22} \cdot (-b_{11} + b_{21}) \\
p_5 &= (a_{11} + a_{12}) \cdot b_{22} \\
p_6 &= (-a_{11} + a_{21}) \cdot (b_{11} + b_{12}) \\
p_7 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22})
\end{aligned}
$$

$$
\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} p_1 + p_4 - p_5 + p_7 & p_3 + p_5 \\ p_2 + p_4 & p_1 + p_3 - p_2 + p_6 \end{pmatrix}
$$

Strassen’s Algorithm

- Strassen’s algorithm performs 7 multiplications and 18 additions/subtraction
- Its real usefulness is the fact that it can be used **recursively**
- We’ll partition our matrices into blocks and multiply them recursively
- The intuition is that matrix additions and subtraction is “cheap” and the real savings come from fewer multiplications

Strassen's Algorithm

Theorem: Matrix multiplication of $n \times n$ matrices can be done in time $O\left(n^{\log_2 7}\right) = O(n^{2.81\ldots})$

Proof: suppose wlog n is a power of 2 (otherwise, pad the matrix with zero rows and columns).

For the recursion, divide the matrix into blocks

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where each block has size $\frac{n}{2} \times \frac{n}{2}$.

Strassen's Algorithm

Using the base case on the blocks, we can multiply A and B using 7 multiplication of $\frac{n}{2} \times \frac{n}{2}$ matrices and some addition operations which take time $O(n^2)$.

We get the recursion $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$ whose solution is $T(n) = O\left(n^{\log_2 7}\right)$

Can we do better?

Bounding ω

Let ω be the best β such that we can multiply matrices in time $O(n^{\beta})$.

It turns out the only the number of multiplication matter.

If we can multiply $m \times m$ matrices with k multiplications,

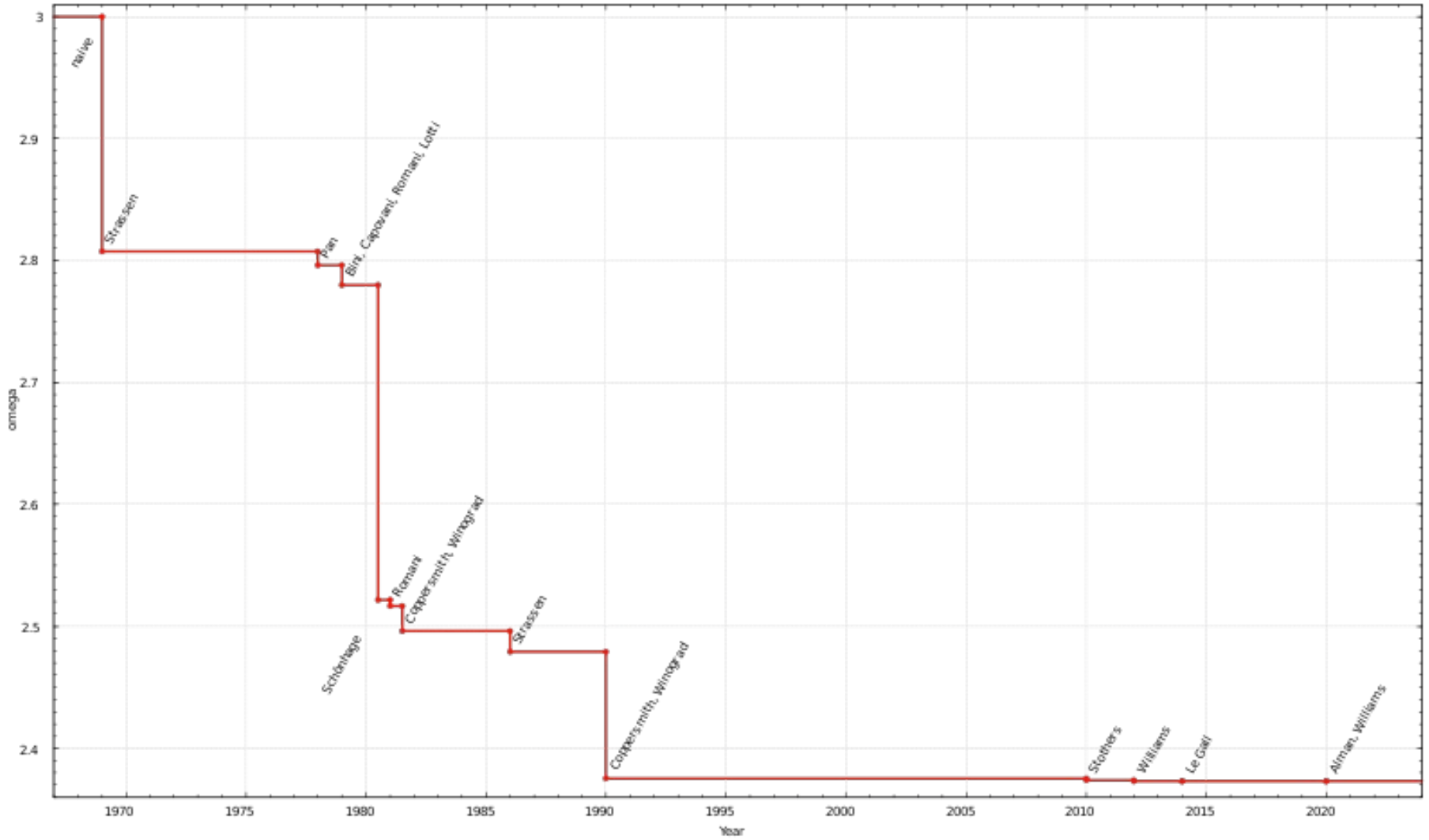
$$\omega \leq \log_m k$$

$$2 \leq \omega \leq 2.81 \ldots$$

Many people believe $\omega = 2$

The complexity of many other matrix operations (inversion, determinant) is also $O(n^{\omega})$

Known bounds on ω



Bounding ω

- Bounding ω can be done by finding efficient way to multiply small matrices
- We can try to search for such constructions using a computer
- Last year, there was a big brouhaha after a research team at google used deep neural nets to do this and found some improved algorithms for 4×4 and 5×5 matrix multiplication
- They did not lower ω
- Later, other researchers found better algorithms using other methods

All Pairs Shortest Path (APSP)

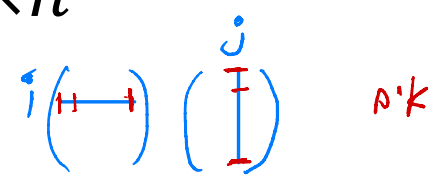
Input: a simple, connected, undirected graph $G = (V, E)$

Output: the distance d_{uv} between every pair $u, v \in V$

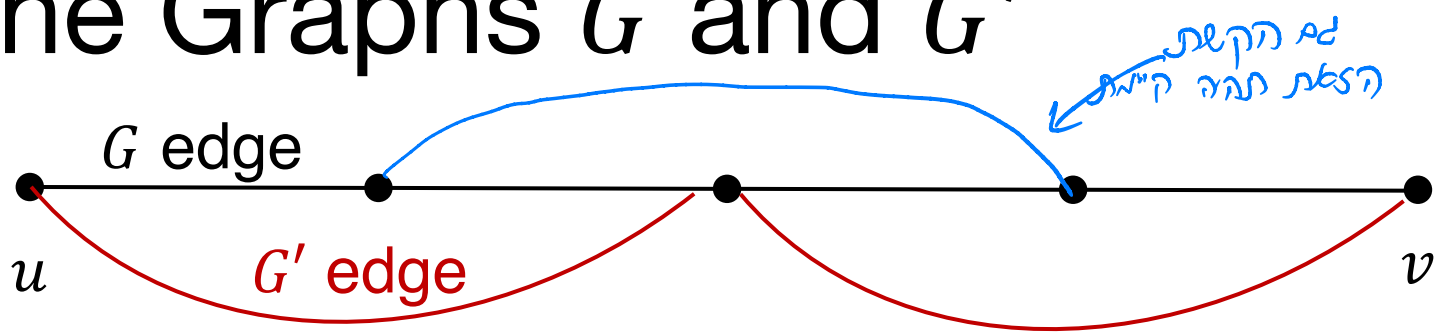
- Distance = length (in edges) of a shortest path
- You saw the Floyd-Warshall algorithm the runs in time $O(|V|^3)$
- Using matrix multiplication, this problem can be solved in time $O(|V|^{\omega} \log|V|)$

Adjacency Matrix

- Given $G = (V, E)$ with $|V| = n$ we define (as usual) an $n \times n$ matrix A such that $A_{i,j} = 1$ if $(i, j) \in E$, 0 otherwise
- What does A^2 correspond to?
- $(A^2)_{i,j} = \sum_{k=1}^{n} A_{i,k} A_{k,j}$
- $(A^2)_{i,j} > 0$ if and only if there's a path of length 2 from i to j
- Let B be a matrix such that:
  $B_{i,j} = 1$ if $(A^2)_{i,j} > 0$ or $A_{i,j} = 1$, 0 otherwise
- Let G' be the graph whose adjacency matrix is B

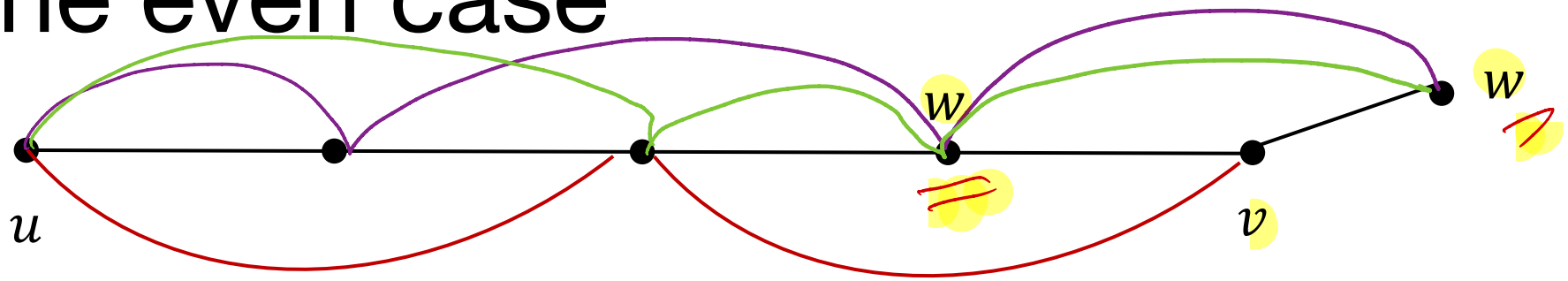


The Graphs G and G'



- Let d be the distance in G from u to v
- If d is even: the distance in G' is $\frac{d}{2}$
- If d is odd: the distance in G' is $\frac{d-1}{2}+1=\frac{d+1}{2}$
- Put differently: $d \in \{2d', 2d'-1\}$ where d' is the distance in G'
- We'll compute d' recursively, and then we just need to distinguish between the case $2d'$ and $2d'-1$

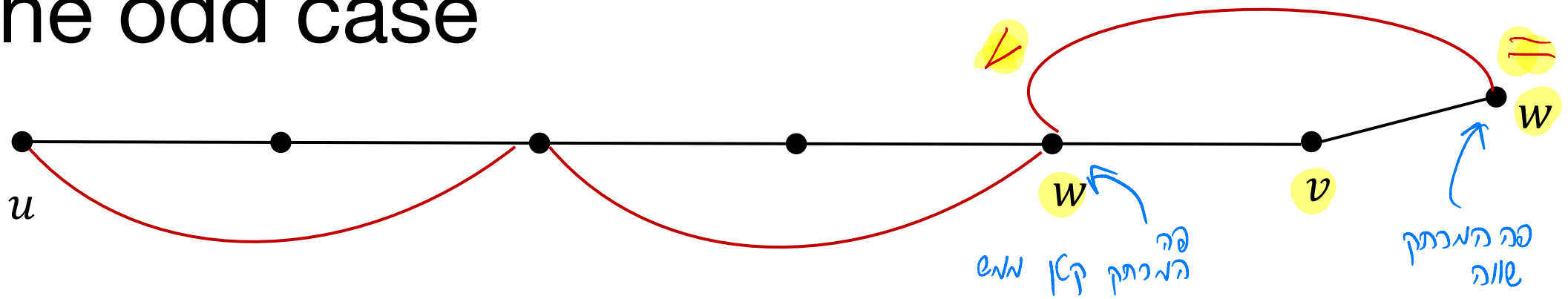
The even case



- If $d = 2d'$: for every neighbor w of v, $\mathrm{dist}_{G'}(u, w) \geq \mathrm{dist}_{G'}(u, v)$
- Let $\Gamma(v)$ be the set of neighbors of v **in G**. We get

$$\sum_{w \in \Gamma(v)} \mathrm{dist}_{G'}(u, w) \geq |\Gamma(v)| \cdot \mathrm{dist}_{G'}(u, v)$$

The odd case



- If $d = 2d' - 1$: for every neighbor w of v, $\mathrm{dist}_{G'}(u, w) \leq \mathrm{dist}_{G'}(u, v)$, and for at least one neighbor, the inequality is strict
- We get

$$\sum_{w \in \Gamma(v)} \mathrm{dist}_{G'}(u, w) < |\Gamma(v)| \cdot \mathrm{dist}_{G'}(u, v)$$

Final observation

- We need to compute $\sum_{w \in \Gamma(v)} \mathrm{dist}_{G'}(u, w)$
- By the induction hypothesis, we have a matrix T such that
$$T_{i,j} = \mathrm{dist}_{G'}(i, j)$$
- Then
$$(TA)_{u,v} = \sum_{w} T_{u,w} A_{w,v} = \sum_{w \in \Gamma(v)} \mathrm{dist}_{G'}(u, w)$$

Base case

- Q: When does the induction terminate?
- A: when all distances are at most 2 and G' is the complete graph
- In this case $B_{i,j} = 1$ for all i, j and

$$\mathrm{dist}_G(i,j) = 2B_{i,j} - A_{i,j} = \begin{cases} 2 & \text{if } A_{i,j} = 0 \\ 1 & \text{otherwise} \end{cases}$$

Final Algorithm

All Pairs Distance

We define a recursive algorithm APD.

Input: adj. matrix A of a simple, connected undirected graph

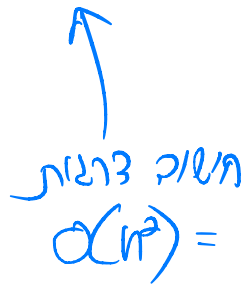
1. Compute A^2.
2. Let $B = (B_{i,j})$ be a matrix such that $B_{i,j} = 1$ if $(A^2)_{i,j} > 0$ or $A_{i,j} > 0$, and 0 otherwise (B is the adjacency matrix of G')
3. If $B_{i,j} = 1$ for all $i \neq j$, return $2B - A$
   (This is the base case of the induction)

Final Algorithm (cont.)

We define a recursive algorithm APD.

Input: adj. matrix A of a simple, connected undirected graph

4. Recursively compute $T \leftarrow \mathrm{APD}(B)$
5. Compute $X = TA$
   (This allows us to distinguish the even case from the odd case: we have that $X_{u,v} = \sum_{w \in \Gamma(v)} \mathrm{dist}_{G'}(u, w)$)
6. For every $u \neq v$: if $X_{u,v} \geq |\Gamma(v)| \cdot T_{u,v}$, set $D_{u,v} = 2 \cdot T_{u,v}$, and otherwise set $D_{u,v} = 2 \cdot T_{u,v} - 1$
7. Return D



Running time:

- In the beginning, the distance between every u, v is at most n
- At each step, we cut the distance roughly by half
- Hence the number of recursive calls is $O(\log n)$
- In each step, we have:
  - 2 steps of matrix multiplications (for computing A^2 and TA)
  - A bunch of other stuff that takes time $O(n^2)$
- Thus, the total running time is $O(n^\omega \log n)$

“Direct Sum”

- Consider the easier problem of matrix-vector multiplication:

  Input: an $n \times n$ matrix A and a column n-dim vector v

  Output: $A \cdot v$

- The naïve algorithm computes this in $O(n^2)$ operations
- This is also optimal
- Now suppose we need to perform this operation n times: Given $v_1, \ldots, v_n$, compute $Av_1, \ldots, Av_n$
- Using the naïve algorithm n times requires $O(n^3)$ operations

“Direct Sum”

- A better method: build a matrix B whose columns are $v_1, \ldots, v_n$
- Compute the multiplication $A \cdot B$ using a fast matrix multiplication algorithm
- The columns of $A \cdot B$ are $(Av_1, \ldots, Av_n)$
- This is an example of **economy of scale**: performing the same operations n times (**on different, unrelated inputs**) is cheaper than n times the cost of performing it once

Verifying Matrix Multiplication

- We don't know if we can multiply matrices in time $O(n^2)$
- However, we'll see that using **randomness**, we can **verify** matrix multiplication in time $O(n^2)$

Input: $n \times n$ matrices A, B, C

Output: "yes" if $C = AB$, "no" (with high probability) if $C \neq AB$

(if we could've multiplied matrices in $O(n^2)$ time, we can just compute AB and compare)

Verifying Matrix Multiplication

- Idea: matrix-vector multiplication takes $O(n^2)$ time
- If $C = AB$ then $Cv = (AB)v$ for every v
- If $C \neq AB$, $Cv = (AB)v$ means that v is in the kernel of the non-zero matrix $(C - AB)$.

Algorithm: pick a random $v \in \{0,1\}^n$ and compute (using matrix vector multiplication) Bv, $A(Bv)$, and Cv.

If $A(Bv) = Cv$ returns "yes", otherwise return "no".

3 אכפלות מטריצה בוקטור
כ- $O(n^2)$

Verifying Matrix Multiplication

Algorithm: pick a random $v \in \{0,1\}^n$ and compute (using matrix vector multiplication) Bv, $A(Bv)$, and Cv.

If $A(Bv) = Cv$ returns “yes”, otherwise return “no”.

Clearly, the algorithm performs $O(n^2)$ operations, and if $C = AB$ it always returns “yes”

Left to prove: If $C \neq AB$, algorithm returns “no” with probability at least $\frac{1}{2}$.

Verifying Matrix Multiplication

The matrix $C - AB$ is non-zero.

Lemma: If D is non-zero then $Dv = 0$ for at most half the points in $\{0,1\}^n$.

Proof: Let i be a non-zero row of D.

$(Dv)_i = \sum_{j=1}^{n} D_{ij} v_j$ is a non-zero polynomial in $v_1, \ldots, v_n$.

By the Schwartz-Zippel Lemma for $d = 1, S = \{0,1\}$,

$$\Pr_{v_1,\ldots,v_n \in \{0,1\}} \left[\sum_{j=1}^{n} D_{ij} v_j = 0 \right] \leq \frac{1}{2} = \frac{d}{|S|}$$

Fast Fourier Transform

- How to multiply two big numbers?

$$\begin{array}{r} 14528868975 \\ \times\ 94701820871 \\ \hline \end{array}$$

- Usually we think of “multiplication” as a single operation
- But when the numbers are huge, it makes sense to count the number of operations as a function of the number of digits
- Grade school algorithm: multiplies two n digit numbers using $O(n^2)$ operations

Polynomial Multiplication

- Can we do better?
- Easier problem: **polynomial multiplication**

Input: two univariate polynomials of degree $< n$, $f(x)$, $g(x)$

Output: the polynomial $f(x) \cdot g(x)$ of degree $< 2n$

- How many operations are needed?
- How are the polynomials “given”?

Coefficient Representation

- $f(x) = \sum_{i=0}^{n-1} a_i x^i$, $g(x) = \sum_{i=0}^{n-1} b_i x^i$
- f, g are both given as lists of n coefficients
- The output should also be a list of coefficients
- Computing $f + g$: $(f + g)(x) = \sum_{i=0}^{n-1} (a_i + b_i) x_i$
- n addition operations

חיבור

- $(f \cdot g)(x) = \sum_{k=0}^{2n-2} \left(\sum_{i=0}^{k} a_i b_{k-i} \right) x^k$
- $O(n^2)$ total operations

כפל

- Can we do better?

לשכה: בכפל פולינומים אין "carry"

Fast Fourier Transform

- The **Fast Fourier Transform** is a fast algorithm for polynomial multiplication
- Variants of it are also used for fast integer multiplication
- But it does much more – used in algorithm for JPEG or MP3 compression
- Appears often in lists of “top 10 most important algorithms”
- We’ll only see the application to polynomial multiplication

Evaluation Representation

- We've seen one natural way to represent polynomials: as a list of n coefficients
- However, there's another natural way to represent polynomials of degree $< n$:

Observation: Let $\alpha_0, \ldots, \alpha_{n-1}$ be **distinct** complex numbers. If f, g are polynomials of degree $< n$, and $f(\alpha_i) = g(\alpha_i)$ for every $0 \leq i \leq n-1$, then $f = g$.

Proof: The assumption implies that $f - g$ is a polynomial of degree $< n$ that has at least n roots, so it must be the zero polynomial.

Evaluation Representation

- The observation implies that we can represent f by its **evaluations** on any distinct n complex numbers
- Suppose $\deg f + \deg g < n$, and we're given f and g using n evaluations $\big(f(\alpha_0), \ldots, f(\alpha_{n-1})\big)$ and $\big(g(\alpha_0), \ldots, g(\alpha_{n-1})\big)$
- The representation of $f \cdot g$ using evaluations is
$$\big(f(\alpha_0) \cdot g(\alpha_0), \ldots, f(\alpha_{n-1}) \cdot g(\alpha_{n-1})\big)$$
- Proof: these are the n evaluations of $f \cdot g$ on $\alpha_0, \ldots, \alpha_{n-1}$ and this representation is unique
- Computing this takes only n multiplications

$O(n)$

Evaluation Representation

- The evaluation representation isn't always the best option
- For example: given $f(x) = \sum_{i=0}^{n-1} a_i \, x^i$ in coefficient representation and a point β, we can evaluate $f(\beta)$ quickly
- Given the coefficient representation, how can we find $f(\beta)$ for β which isn't one of the α_i's?
- Converting from the evaluation representation to the coefficient representation is called **interpolation**

Evaluation Representation

סיכום

A blueprint for a fast multiplication algorithm:

1. Go from coefficient representation to evaluation representation — $O(n \log n)$
2. Use n multiplications to compute $f \cdot g$ in the evaluation representation — $O(n)$
3. Interpolate back to the coefficient representation of $f \cdot g$ — $O(n \log n)$

We need steps 1 and 3 to take less than n^2 steps

This doesn't look promising – step 1 by itself seems to take n^2 steps

Smart Evaluation Points

- The key for performing steps 1 and 3 efficiently will be a careful choice of the evaluation point $\alpha_0, \ldots, \alpha_{n-1}$
- Evaluation is a **linear operation**
- The matrix representing it is called a **Vandermonde Matrix**

$$A = \begin{pmatrix} 1 & \alpha_0^1 & \cdots & \alpha_0^{n-1} \\ 1 & \alpha_1^1 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_{n-1}^1 & \cdots & \alpha_{n-1}^{n-1} \end{pmatrix} = \begin{pmatrix} - & a_i & - \end{pmatrix}$$

- The matrix representing the interpolation is A^{-1} - we want to pick $\alpha_0, \ldots, \alpha_{n-1}$ so that computing A^{-1} is easy

הערכה:

נתון פולינום:

$$f = \sum_{i=1}^{n} f_i x^i$$

ומעוניינים לבצע אינטרפולציה:

$$\begin{pmatrix} 1 & \alpha_0 & \cdots & \alpha_0^{n-1} \\ 1 & \alpha_1 & \cdots & \alpha_1^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \alpha_{n-1} & \cdots & \alpha_{n-1}^{n-1} \end{pmatrix}$$

A

$$\begin{pmatrix} 1 & \alpha_0 & \cdots & \alpha_0^{n-1} \\ 1 & \alpha_1 & \cdots & \alpha_1^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \alpha_{n-1} & \cdots & \alpha_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_0 \cdot 1 + f_1 \cdot \alpha_0 + f_2 \alpha_0^2 + \cdots + f_{n-1} \alpha_0^{n-1} \end{pmatrix} = \begin{pmatrix} f(\alpha_0) \\ \vdots \\ f(\alpha_{n-1}) \end{pmatrix}$$

$$Af = \overrightarrow{f(\alpha_i)}$$

מקדמים ⟸ שורשים

מסקנה:

שורשים ⟹ מקדמים

מה אומרת מסקנה?

שורשים ⟸ מקדמים.

כלומר, ניתן למצוא את A^{-1}.

כלומר: קיימת מטריצה שלפי α_i, נוכל למצוא את A^{-1} בקלות.

מסקנה: מציאת מטריצה הופכית לוקחת $O(n^3)$

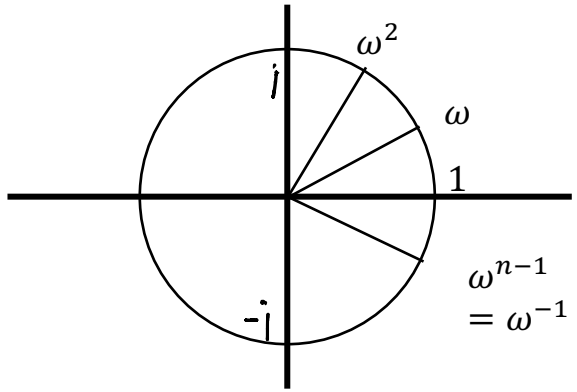
(או $O(n^\omega)$ ואומרת נכונה ב- $O(n \log n)$)

Roots of Unity

- A complex number $z \in \mathbb{C}$ is called a **root of unity of order n** if $z^n = 1$
- For every n there are exactly n distinct roots of unity of order n
- **Proof:** let $\omega = e^{\frac{2\pi i}{n}}$. Then $\omega^n = e^{2\pi i} = 1$ and for every $0 \leq j < n$,

$$\left(\omega^j\right)^n = \left(\omega^n\right)^j = 1$$

$$e^{\frac{2\pi i}{n}}.$$



שאלה: למה אין עוד שורש יחידה?

Roots of Unity

- No other roots of unity because $x^n - 1$ has at most n roots
- **Important**: no relation to the matrix multiplication exponent ω!
- E.g.: when $n = 2$, ± 1 are the two roots of unity of order 2
- 1 is always a root of unity, $-1 = \omega^{\frac{n}{2}}$ is iff n is even
- Spoiler alert: our n evaluation points will be the n roots of unity of order n: $\omega^0, \omega, \ldots, \omega^{n-1}$
- We'll first show that the inverse Vandermonde matrix for those points is easy to compute

תשובה:

DFT Matrix

Discrete Fourier Transform

- Let A_ω be the Vandermonde matrix corresponding to $1, \omega, \ldots, \omega^{n-1}$: $(A_\omega)_{i,j} = \omega^{i \cdot j}$

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \ddots & \\ 1 & \omega^{n-1} & & \omega^{(n-1)^2} \end{pmatrix}$$

$0 \le i, j \le n-1$

- A_ω is called the **Discrete Fourier Transform Matrix**
- **Claim**: $(A_\omega)^{-1} = \frac{1}{n} A_{\omega^{-1}}$ $\quad ((A_{\omega^{-1}})_{i,j} = \omega^{-i \cdot j})$

Inverse of DFT Matrix

- **Claim**: $(A_\omega)^{-1} = \frac{1}{n} A_{\omega^{-1}} \quad ((A_{\omega^{-1}})_{i,j} = \omega^{-i \cdot j})$
- **Proof**: Let $A = A_\omega$ and $B = A_{\omega^{-1}}$.
- For every $0 \leq i \leq n-1$,

$$(AB)_{ii} = \sum_{k=0}^{n-1} A_{ik} B_{ki} = \sum_{k=0}^{n-1} \omega^{ik} \omega^{-ik} = n$$

- For every $0 \leq i, j \leq n-1$ such that $i \neq j$:

$$(AB)_{ij} = \sum_{k=0}^{n-1} \omega^{ik} \omega^{-kj} = \sum_{k=0}^{n-1} \left(\omega^{(i-j)}\right)^k$$

Inverse of DFT Matrix

- Let $t = i - j$. $t \neq 0$ and $|t| < n$ so $\zeta = \omega^t \neq 1$
- Using the formula for a sum of geometric progression,

$$\sum_{k=0}^{n-1} \zeta^k = \frac{\zeta^n - 1}{\zeta - 1} = 0$$

- It follows that computing the inverse of A_ω is "almost identical" to computing A_ω
- Thus from now on we'll concentrate on finding an efficient algorithm for computing A_ω

Roots Of Unity

- We'll prove some simple properties of roots of unity that will later help us compute the linear transformation A_ω by induction
- **Claim**: suppose n is even and let $\omega = e^{\frac{2\pi i}{n}}$. The roots of unity of order $\frac{n}{2}$ are precisely $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$
- **Proof**: let $m = n/2$. Then $\zeta = \omega^2 = e^{\frac{2\pi i}{m}}$ and for every $j < m$, it holds that $\zeta^j = \omega^{2j}$

Roots Of Unity

- **Corollary**: the function $z \mapsto z^2$ is a 2-to-1 and onto function from the roots of unity of order n to the roots of unity of order $\frac{n}{2}$

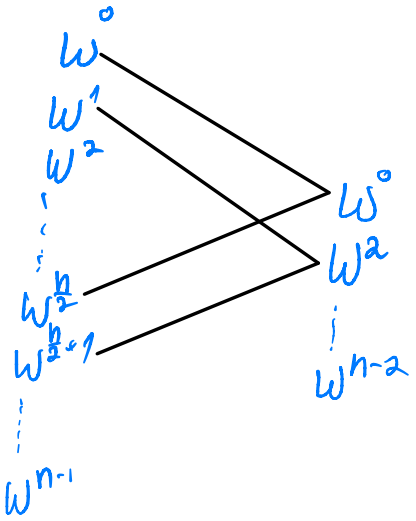
$$1 = (\omega^0)^2 = \left(\omega^{\frac{n}{2}}\right)^2 = (-1)^2$$

$$(\omega)^2 = \left(\omega^{\frac{n}{2}+1}\right)^2 = (\omega)^2 = \omega^{\frac{2n}{2}+2} = \omega^{\overset{1}{n}} \cdot \omega^2$$

$$(\omega^2)^2 = \left(\omega^{\frac{n}{2}+2}\right)^2 = (-\omega^2)^2$$

$$\vdots$$

$$\left(\omega^{\frac{n}{2}-1}\right)^2 = (\omega^{n-1})^2 = \left(-\omega^{\frac{n}{2}-1}\right)^2$$



An Algorithm for Computing DFT

- We're now ready to present the algorithm for computing A_ω
- The algorithm we present is called the **Fast Fourier Transform**
- Recall: computing A_ω is equivalent to evaluating a polynomial $f = \sum_{i=0}^{n-1} c_i x^i$ in $1, \omega, \ldots, \omega^{n-1}$:

$$A_\omega \begin{pmatrix} c_0 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} f(1) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}$$

- Also recall: the naive algorithm does this in time $O(n^2)$
- We'll do it in time $O(n \log n)$

An Algorithm for Computing DFT

- Suppose wlog n is a power of 2. Let $f = \sum_{i=0}^{n-1} c_i x^i$. Let

$$f_0 = c_0 + c_2 x + c_4 x^2 + \cdots + c_{n-2} x^{\frac{n}{2}-1} = \sum_{i=0}^{\frac{n}{2}-1} c_{2i} x^i$$

$$f_1 = c_1 + c_3 x + c_5 x^2 + \cdots + c_{n-1} x^{\frac{n}{2}-1} = \sum_{i=0}^{\frac{n}{2}-1} c_{2i+1} x^i$$

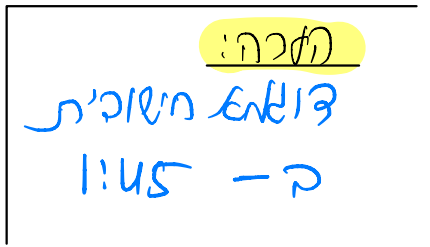
An Algorithm for Computing DFT

- Observe: f_0, f_1 are both polynomials of degree $< \frac{n}{2}$, and
$$f(x) = f_0(x^2) + x f_1(x^2)$$
- $f(\omega) = f_0(\omega^2) + \omega f_1(\omega^2)$
- $f\left(\omega^{\frac{n}{2}+1}\right) = f_0(\omega^{n+2}) + \omega^{\frac{n}{2}+1} f_1(\omega^{n+2}) = f_0(\omega^2) - \omega f_0(\omega^2)$
- This is true for all roots of unity, and their squares are exactly the roots of unity of order $\frac{n}{2}$
- So we can recursively evaluate f_0, f_1 and all roots of unity of order $\frac{n}{2}$ and use the results to evaluate f

An Algorithm for Computing DFT

Algorithm:

1. Recursively compute $f_0(\omega^2), f_0(\omega^4) \ldots, f_0(\omega^{n-2})$ and similarly $f_1(\omega^2), f_1(\omega^4) \ldots, f_1(\omega^{n-2})$
2. For $0 \leq i \leq \frac{n}{2} - 1$, compute
    1. $f(\omega^i) = f_0(\omega^{2i}) + \omega^i f_1(\omega^{2i})$
    2. $f\left(\omega^{\frac{n}{2}+i}\right) = f_0(\omega^{2i}) - \omega^i f_1(\omega^{2i})$



An Algorithm for Computing DFT

- Let $T(n)$ denote the running time of the algorithm
- The recursion implies that $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- So $T(n) = O(n \log n)$.

Back to Polynomial Multiplication

Algorithm for multiplying polynomials f, g such that $\deg f + \deg g < n$:

1. Compute $\hat{f} = \left(f(1), f(\omega), \ldots, f(\omega^{n-1})\right)$ and similarly $\hat{g}$ ← DFT
2. Compute $\widehat{f \cdot g} = \left(\hat{f}_0 \cdot \hat{g}_0, \ldots, \hat{f}_{n-1} \cdot \hat{g}_{n-1}\right)$
   (this is the evaluation vector of $f \cdot g$ on $1, \omega, \ldots, \omega^{n-1}$)
3. Compute $f \cdot g$ using the inverse Fourier transform on $\widehat{f \cdot g}$ ← Inverse DFT
   (this is done as in step 1 using $A_{\omega^{-1}}$)

Total time complexity: $O(n \log n)$

Other Polynomial Tricks

Using polynomial multiplication, we can also design fast algorithms for:

1. **Polynomial division with remainder**: given f, g such that $\deg g < \deg f$, find the **unique** q, r such that $f = q \cdot g + r$ and $\deg r < \deg g$, in time $O(n \log n)$
2. **Multipoint evaluation**: given f of degree n and $\alpha_1, \ldots, \alpha_n$, output $f(\alpha_1), \ldots, f(\alpha_n)$, in time $O(n \log^2 n)$
   - Another example of an "economy of scale" phenomenon
3. **Polynomial interpolation**: given $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ find the **unique** f of degree $< n$ such that $f(\alpha_i) = \beta_i$ for all i.

Computing Determinants

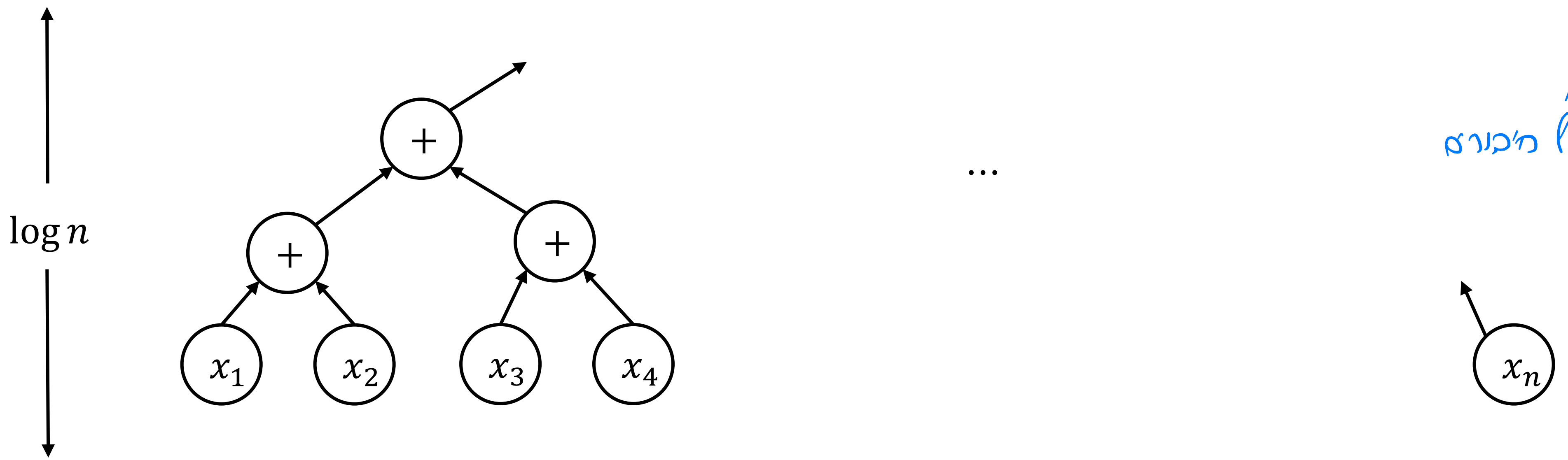
- Let A be an $n \times n$ matrix
- $\det A = \sum_{\sigma \in S_n} (-1)^{\operatorname{sgn} \sigma} \prod_{i=1}^{n} A_{i,\sigma(i)}$
- The naïve algorithm for computing $\det A$ requires $n!$ operations
- Nevertheless, there are efficient ways to compute the determinant
- Gaussian elimination: transform A to row echelon form using operations that preserve the determinant. Requires at most n^3 operations
- If A is in row echelon form, its determinant is the product of elements on the diagonal

Parallel computation

- Gaussian elimination (דירוג מטריצות) is sequential – we need to do n^2 row operations one after the other
- We'll be interested in **parallel** algorithms for the determinant
- We don't formally define the model
  - But informally, we have a polynomial number of processors that can operate in parallel
  - But if their input depends on previous steps, the previous steps must be completed

Example for Parallel Computation

- Consider matrix multiplication again: $C = A \cdot B$
- $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$
- Every entry of C can be calculated in parallel using the input
- Sum of n elements can be computed in parallel in $O(\log n)$:



הערה:

עדיין יש n מחוברים,

שאפשר לחבר ע"י log n חיבורים

Repeated Squaring

- **Corollary**: A^n can be computed in parallel time $O(\log^2 n)$
- By computing $A^2, (A^2)^2 = A^{2^2}, (A^4)^2 = A^{2^3}, \ldots, A^{2^{\log n}}$
- This algorithm is called **repeated squaring**
- Similarly, we can compute (simultaneously) all powers of A up to n: $A, A^2, A^3, \ldots, A^n$
- We'll show that the **determinant** can also be computed in parallel time $O(\log^2 n)$ using a polynomial number of processors

Eigenvalues

- The determinant has many different characterizations
- It also equals the product of eigenvalues $\lambda_1 \cdot \lambda_2 \cdots \lambda_n$
- More generally, let $p(x) = \det(xI - A)$ be the **characteristic polynomial** of A.
- $p(x) = (x - \lambda_1)(x - \lambda_2) \ldots (x - \lambda_n)$
- The coefficient of x^n is 1
- The coefficient of x^{n-1} is $-(\sum_{i=1}^{n} \lambda_i) = -\mathrm{tr}(A)$
- The constant coefficient is $(-1)^n \det A$

Characteristic Polynomial

- $p(x) = (x - \lambda_1)(x - \lambda_2) \ldots (x - \lambda_n)$
- The coefficient of x^{n-k} is $(-1)^k \sum_{S \subseteq \{1,\ldots,n\}, |S|=k} \prod_{i \in S} \lambda_i$
- We'll show an algorithm that computes **all the coefficients of the characteristic polynomial** in parallel time $O(\log^2 n)$
- $e_k(x_1, \ldots, x_n) = \sum_{S \subseteq \{1,\ldots,n\}, |S|=k} \prod_{i \in S} x_i$ is called the ***k*-th elementary symmetric polynomial**

Characteristic Polynomial

- $p(x) = (x - \lambda_1)(x - \lambda_2) \ldots (x - \lambda_n)$
- The coefficient of x^{n-k} is $(-1)^k \sum_{S \subseteq \{1,..,n\}, |S|=k} \prod_{i \in S} \lambda_i$
- It's very easy to compute the coefficient of x^n (it is 1)
- The coefficient of x^{n-1} is the trace of A which is also easy to compute: $\sum_{i=1}^{n} A_{ii}$
- We will compute $e_k(\lambda_1, \ldots, \lambda_n)$ for every $1 \leq k \leq n$
- We will start by computing other symmetric functions of $\lambda_1, \ldots, \lambda_n$

Eigenvalues of Powers

- Recall that we can compute $A, A^2, \ldots, A^n$ in parallel time $O(\log^2 n)$
- Q: What are the eigenvalues of A^2?
- A: $\lambda_1^2, \ldots, \lambda_n^2$
  - Proof: let v_i be eigenvector of λ_i. $(A^2)v_i = A(Av_i) = A(\lambda_i v_i) = \lambda_i^2 v_i$
- The eigenvalues of A^k are λ_i^k
- After computing A^k its easy to compute its trace
  - Which equals $\sum_{i=1}^{n} \lambda_i^k$

Power sum polynomials

- Let $p_k(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^k$
- We can compute $p_k(\lambda_1, \ldots, \lambda_n)$ for $0 \leq k \leq n$ in parallel time $O(\log^2 n)$:
  - Compute $A, A^2, \ldots, A^n$, compute the traces of all matrices
- The polynomials p_k are called **the power sum polynomials**
- They are also symmetric functions in $x_1, \ldots, x_n$
- It turns out that they are related to the polynomials e_k using **Newton's identities**

Newton's identities

- $e_1 = p_1$
- $2e_2 = e_1 p_1 - p_2$
- $3e_3 = e_2 p_1 - e_1 p_2 + p_3$
- $4e_4 = e_3 p_1 - e_2 p_2 + e_1 p_3 - p_4$
- …
- $k e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} p_i$

Proof of Newton's identities

- Suppose $n = k$ and consider $\prod_{i=1}^{k}(t - x_i)$
- As before, this equals $\sum_{i=0}^{k}(-1)^{k-i}e_{k-i}(x_1, \ldots, x_k)t^i$
- Plug in $t = x_j$ to get $0 = \sum_{i=0}^{k}(-1)^{k-i}e_{k-i}(x_1, \ldots, x_n)x_j^i$
- Sum equations for $j = 1, \ldots k$ to get

$$0 = k(-1)^k e_k + \sum_{i=1}^{k}(-1)^{k-i}e_{k-i} \cdot p_i$$

- Proofs for $k < n$ and $k > n$: similar.

Back to Determinant

- Recall: we have $p_1, \ldots, p_n$ and want $e_1, \ldots, e_k$
- $ke_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} p_i = \sum_{i=1}^{k-1} (-1)^{i-1} e_{k-i} p_i + (-1)^{k-1} p_k$
- By multiplying by $(-1)^k$ and dividing by k, we get

$$\frac{-p_k}{k} = (-1)^k e_k + \sum_{i=1}^{k-1} (-1)^{k-i} e_{k-i} \cdot \frac{p_i}{k}$$

- We put it in matrix form…

Newton Identities in Matrix Form

$$\underbrace{\begin{pmatrix} -p_1 \\ -\frac{p_1}{2} \\ \vdots \\ -\frac{p_i}{i} \\ \vdots \\ -\frac{p_n}{n} \end{pmatrix}}_{p} = \underbrace{\begin{pmatrix} 1 & 0 & & & 0 \\ \frac{p_1}{2} & 1 & 0 & \cdots & 0 \\ \frac{p_2}{3} & \frac{p_1}{3} & 1 & \vdots & 0 \\ \vdots & \ddots & & & \\ \frac{p_{n-1}}{n} & & \frac{p_2}{n} & \frac{p_1}{n} & 1 \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} -e_1 \\ e_2 \\ \vdots \\ (-1)^i e_i \\ \vdots \\ (-1)^n e_n \end{pmatrix}}_{e}$$

Recap

- We have the vector p and the matrix M and we want to compute e
- $p = Me$ which implies that $e = M^{-1}p$
- If we could invert M we would get e by multiplying with p
- By inversion seems as hard as computing the determinant…
- Is M even invertible?
- Yes, it is **lower triangular**. This will also help in inverting it

Inverting Lower Triangular Matrices

Lemma: if A is lower triangular it can be inverted in parallel time $O(\log^2 n)$

Proof:

- Break A as $\frac{n}{2} \times \frac{n}{2}$ blocks $\begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$
  - $A^{-1} = \begin{pmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{pmatrix}$ (multiply to verify)
- Recursively compute (in parallel) B^{-1}, D^{-1}
- Compute A^{-1} using two extra matrix multiplications

Inverting Lower Triangular Matrices

Lemma: if A is lower triangular it can be inverted in parallel time $O(\log^2 n)$

Let $T(n)$ be the parallel time this takes and $M(n) = O(\log n)$ the parallel time of matrix multiplication. We get

$$T(n) = T\left(\frac{n}{2}\right) + 2M\left(\frac{n}{2}\right)$$

So $T(n) = O(\log^2 n)$

Wrapping up

- Using a parallel algorithm for inverting lower triangular matrices, we can invert M
- We have already computed p so we compute $M^{-1}p$ in parallel time $O(\log n)$
- The total parallel time is $O(\log^2 n)$

Matrix Inversion

- Since we did all this work we it's also easy now to show how to compute the **inverse** of A in parallel time $O(\log^2 n)$
- The **Cayley-Hamilton Theorem** states that A satisfies its characteristic polynomial
- We have computed all the coefficients of the characteristic polynomial

$$A^n - e_1 A^{n-1} + e_2 A^{n-2} + \cdots + (-1)^{n-1} e_{n-1} A + (-1)^n e_n I = 0$$

- Multiply by A^{-1}, we get

$$A^{-1} = \frac{e_{n-1} I - e_{n-2} A + \cdots + (-1)^{n-1} A^{n-1}}{e_n}$$

Matrix Inversion

$$A^{-1} = \frac{e_{n-1}I - e_{n-2}A + \cdots + (-1)^n A^{n-1}}{e_n}$$

We have computed $e_1, \ldots, e_n$ and $A, A^2, \ldots A^{n-1}$

In extra parallel time $O(\log n)$ we can compute the expression above and invert A