Algorithmik II

Markus Lohrey

Universität Siegen

Wintersemester 2020/2021

Convolution of polynomials

Consider two polynomials (with coefficients from $\mathbb{C}$ or another field)

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n, \ g(x) = b_0 + b_1 x + b_2 x^2 + \cdots b_m x^m$$

represented by the coefficient tuples

$$f = (a_0, \ldots, a_n, a_{n+1}, \ldots, a_{N-1}), \ g = (b_0, \ldots, b_m, b_{m+1}, \ldots, b_{N-1}),$$

where $N = n + m + 1$, $a_{n+1} = \cdots = a_{N-1} = b_{b+1} = \cdots = b_{N-1} = 0$.

We want to compute the product polynomial

$$(fg)(x) = a_0 b_0 + (a_1 b_0 + a_0 b_1) x + \cdots + (a_0 b_{N-1} + \cdots + a_{N-1} b_0) x^{N-1}$$

represented by the coefficient tuple

$$fg = (a_0 b_0, a_1 b_0 + a_0 b_1, \ldots, a_0 b_{N-1} + \cdots + a_{N-1} b_0),$$

(also called the convolution of the tuples f and g).

Point representation of polynomials

Naive computation of fg: $\mathcal{O}(N^2)$ scalar operations in the coefficient field.

FFT (James Cooley and John Tukey, 1965): only $\mathcal{O}(N \log(N))$ operation.

Main idea: work with the point representation of polynomials.

A polynomial f of degree at most $N-1$ can be uniquely represented by its values

$$(f(\zeta_0), f(\zeta_1), \ldots, f(\zeta_{N-1})),$$

where $\zeta_0, \ldots, \zeta_{N-1}$ are N different values of the underlying field (e.g. $\mathbb{C}$).

We obviously have $(fg)(\zeta) = f(\zeta)g(\zeta)$.

Hence: the point representation of the convolution of f and g can be computed in time $\mathcal{O}(N)$ from the point representations of f and g.

Main principle of FFT

Main principle of the fast Fourier transformation (FFT):

coefficient rep. of f and g

Main principle of FFT

Main principle of the fast Fourier transformation (FFT):

coefficient rep. of f and g

evaluation

point rep. of f and g

Main principle of FFT

Main principle of the fast Fourier transformation (FFT):

coefficient rep. of f and g
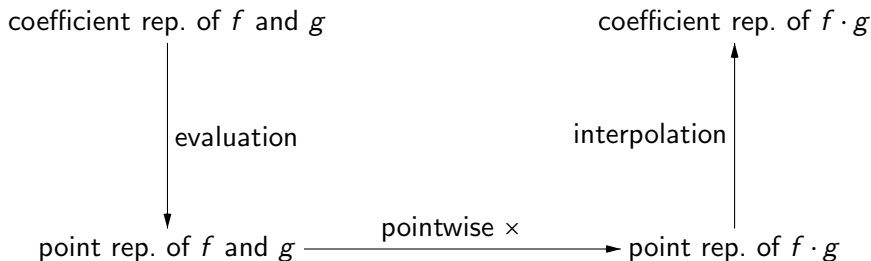
$\downarrow$ Auswertung

point rep. of f and g $\xrightarrow{\text{pointwise} \times}$ point rep. of $f \cdot g$

Main principle of FFT

Main principle of the fast Fourier transformation (FFT):

Roots of unity

The crucial point is the choice of the points $\zeta_0, \zeta_1, \ldots, \zeta_{N-1}$.

Assumption: The coefficients of the polynomials come from a field $\mathbb{F}$ such that:

- N has a multiplicative inverse in $\mathbb{F}$, i.e. the characteristics of $\mathbb{F}$ does not divide N.

- The polynomial $X^N - 1$ has N different roots (the N-th roots of unity), which can be written as ω^i $(0 \le i < N)$ for a root ω.

For the field $\mathbb{F} = \mathbb{C}$ both conditions are satisfied and the N-th roots of unity are ω^j $(0 \le j < N)$, where $\omega = e^{\frac{2\pi i}{N}}$.

The root ω is also called a primitive N-th root of unity.

Roots of unity in $\mathbb{C}$

Let $\omega = e^{\frac{2\pi i}{12}} = \cos\left(\frac{2\pi}{12}\right) + \sin\left(\frac{2\pi}{12}\right) \cdot i$ (a primitve 12-th root of unity in $\mathbb{C}$).
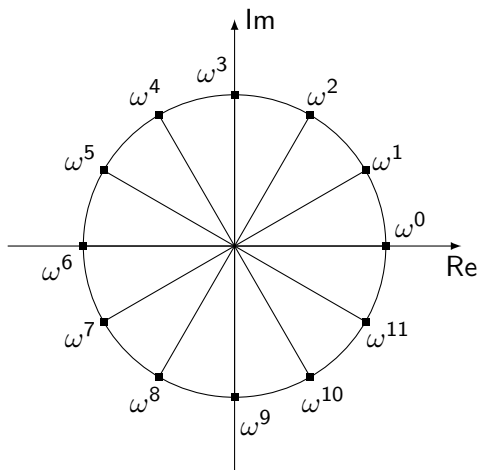
Fast Fourier transformation (FFT)

Lemma 1

If there are N different N-th roots of unity, then there is a primitive N-th root of unity.

Proof: Assume that the polynomial $X^N - 1$ has N different roots.

The N-th roots of unity form a finite abelian group G of cardinality N under multiplication.

We have to show that G is cyclic, i.e., G is generated by single element ω (which then is a primitive N-th root of unity).

Assume that G is not cyclic. Then, $x^d = 1$ for all $x \in G$, where d is a proper divisor of N.

Thus, $x^d - 1$ has N different roots, a contradiction. $\qquad\square$

Roots of unity

Some useful facts from algebra (recall that $i \equiv j \bmod N$ means that $i - j = k \cdot N$ for some $k \in \mathbb{Z}$):

Lemma 2

Let ω be a primitive N-th root of unity and let $i, j \in \mathbb{Z}$:

- $\omega^i = \omega^j$ *if and only if* $i \equiv j \bmod N$.
- ω^i *is a primitive N-th root of unity if and only if* $\gcd(i, N) = 1$
  ($\gcd(i, N)$ = greatest common divisor of i and N).
- *In particular,* $\omega^{-1} = \omega^{N-1}$ *is a primitive N-th root of unity.*

Proof:

- $\omega^i = \omega^j \iff \omega^{i-j} = 1 \iff \omega^{(i-j) \bmod N} = \omega^0 \iff (i - j) \bmod N = 0$
  $\iff i \equiv j \bmod N$

Roots of unity

- First assume that $\gcd(i, N) = 1$. Then there exist $a, b \in \mathbb{Z}$ with $a \cdot i + b \cdot N = 1$.

  Hence, $\omega = \omega^1 = \omega^{a \cdot i + b \cdot N} = (\omega^i)^a (\omega^N)^b = (\omega^i)^a$.

  Thus, ω^i is primitive root of unity.

- Assume that ω^i is a primitive root of unity.

  Hence, there exists $a \in \mathbb{Z}$ such that $\omega^1 = (\omega^i)^a = \omega^{a \cdot i}$.

  By the first statement of the lemma, we get $1 \equiv a \cdot i \bmod N$.

  Hence, there exists $b \in \mathbb{Z}$ with $1 = a \cdot i + b \cdot N$.

  This implies $\gcd(i, N) = 1$.

- The third statement of the lemma follows from the second, since $\gcd(-1, N) = 1$. $\qquad\square$

Fast Fourier transformation (FFT)

Fix a primitive N-th root of unity ω.

Choose the points $\zeta_i = \omega^i$ $(0 \le i \le N - 1)$ for the evaluation of f and g.

Evaluation of the polynomial $f = a_0 + a_1 x + \cdots a_{N-1} x^{N-1}$ at the points $\omega^0 = 1, \omega^1, \ldots, \omega^{N-1}$ is equivalent to a matrix-vector multiplication:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} f(1) \\ f(\omega) \\ f(\omega^2) \\ \vdots \\ f(\omega^{N-1}) \end{pmatrix}$$

The linear mapping realized by the matrix $F_N(\omega) = (\omega^{ij})_{0 \le i, j < N}$ is called the discrete Fourier transformation.

Inverse FFT

Lemma 3

$(F_N(\omega))^{-1} = \frac{1}{N} F_N(\omega^{-1})$, i.e. the inverse of the matrix $(\omega^{ij})_{0 \le i,j < N}$ is the matrix $\left(\frac{\omega^{-ij}}{N}\right)_{0 \le i,j < N}$ (recall: ω^{-1} is a primitive N-th root of unity).

Proof: Since we have

$$x^N - 1 = (x - 1) \cdot \sum_{k=0}^{N-1} x^k,$$

every ω^i for $1 \le i \le N - 1$ is a root of the polynomial $\sum_{k=0}^{N-1} x^k$.

Hence, we get for all $0 \le i \le N - 1$:

$$\sum_{k=0}^{N-1} \omega^{i \cdot k} = \begin{cases} 0 & \text{if } i > 0 \\ N & \text{if } i = 0 \end{cases}$$

Inverse FFT

We obtain for all $0 \le i, j \le N - 1$:

$$
\begin{aligned}
\sum_{k=0}^{N-1} \omega^{ik} \cdot \frac{\omega^{-kj}}{N} &= \sum_{k=0}^{N-1} \frac{1}{N} \cdot \omega^{k \cdot (i-j)} \\
&= \sum_{k=0}^{N-1} \frac{1}{N} \cdot \omega^{k \cdot (i-j) \bmod N} \\
&= \begin{cases} 0 & \text{if } i \ne j \\ 1 & \text{if } i = j \end{cases}
\end{aligned}
$$

(note that for $0 \le i, j \le N - 1$: $i = j$ if and only if $(i - j) \bmod N = 0$)

Hence $\left(\frac{\omega^{-ij}}{N} \right)_{0 \le i, j < N}$ is indeed the inverse of the matrix $\left(\omega^{ij} \right)_{0 \le i, j < N}$. $\qquad \square$

FFT using divide & conquer

It remains to compute the discrete Fourier transformation

$$f \mapsto F_N(\omega) \cdot f$$

(where $f = (a_0, a_1, \ldots, a_{N-1})^T$) in time $\mathcal{O}(N \log(N))$.

Then, we can compute the inverse discrete Fourier transformation
($=$ interpolation)

$$h \mapsto (F_N(\omega))^{-1} \cdot h = \frac{1}{N} F_N(\omega^{-1}) \cdot h$$

within the same time bound.

The "school method" for the multiplication of a matrix with a vector needs time $\mathcal{O}(N^2)$: no gain over the "school method" for polynomial multiplication.

We compute $F_N(\omega) \cdot f = (\omega^{ij})_{0 \le i,j < N} \cdot f$ using divide & conquer.

FFT using divide & conquer

Assume that N is even. For $f(x) = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$ let

$$f_0(x) = a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{N-2} x^{N-2}$$
$$\widehat{f_0}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{N-2} x^{\frac{N-2}{2}}$$
$$f_1(x) = a_1 + a_3 x^2 + a_5 x^4 + \cdots + a_{N-1} x^{N-2}$$
$$\widehat{f_1}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{N-1} x^{\frac{N-2}{2}}$$

We have $f(x) = f_0(x) + x \cdot f_1(x)$, $f_0(x) = \widehat{f_0}(x^2)$ and $f_1(x) = \widehat{f_1}(x^2)$.

The polynomials $\widehat{f_0}(x)$ and $\widehat{f_1}(x)$ have degree $\leq \frac{N-2}{2} = \frac{N}{2} - 1$.

Let $0 \leq i < N$. Since ω^2 is a primitive $\frac{N}{2}$-th root of unity, we get:

$$\begin{aligned}
(F_N(\omega) \cdot f_0)_i &= f_0(\omega^i) = \widehat{f_0}(\omega^{2i}) = \widehat{f_0}(\omega^{2i \bmod N}) = \widehat{f_0}(\omega^{2(i \bmod \frac{N}{2})}) \\
&= \widehat{f_0}((\omega^2)^{i \bmod \frac{N}{2}}) = (F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_0})_{i \bmod \frac{N}{2}},
\end{aligned}$$

FFT using divide & conquer

In particular, we have

$$(F_N(\omega) \cdot f_0)_i = (F_N(\omega) \cdot f_0)_{i+\frac{N}{2}} = (F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_0})_i$$

for $0 \le i \le \frac{N}{2} - 1$.

We obtain

$$F_N(\omega) \cdot f_0 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_0} \\ F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_0} \end{pmatrix}$$

and analogously

$$F_N(\omega) \cdot f_1 = \begin{pmatrix} F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_1} \\ F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_1} \end{pmatrix}.$$

FFT using divide & conquer

With $f(x) = f_0(x) + x \cdot f_1(x)$ we get for all $0 \le i \le N - 1$:

$$
\begin{aligned}
(F_N(\omega) \cdot f)_i &= f(\omega^i) \\
&= f_0(\omega^i) + \omega^i \cdot f_1(\omega^i) \\
&= (F_N(\omega) \cdot f_0)_i + \omega^i \cdot (F_N(\omega) \cdot f_1)_i
\end{aligned}
$$

We have reduced the computation of $F_N(\omega) \cdot f$ to

- the computation of $F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_0}$ and $F_{\frac{N}{2}}(\omega^2) \cdot \widehat{f_1}$
  (2 FFTs in dimension $N/2$)

- and $\Theta(N)$ many further arithmetic operations.

We obtain the recursion $T_{\text{fft}}(N) = 2 T_{\text{fft}}(N/2) + dN$ for a constant d.

Master theorem I ($a = b = 2, c = 1$): $T_{\text{fft}}(N) \in \theta(N \log N)$.
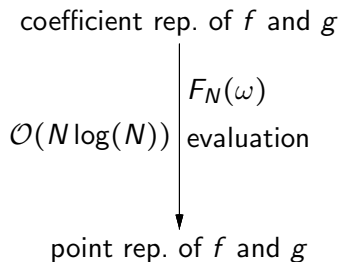
Fast Fourier transformation (FFT)

Main principle of FFT:

coefficient rep. of f and g
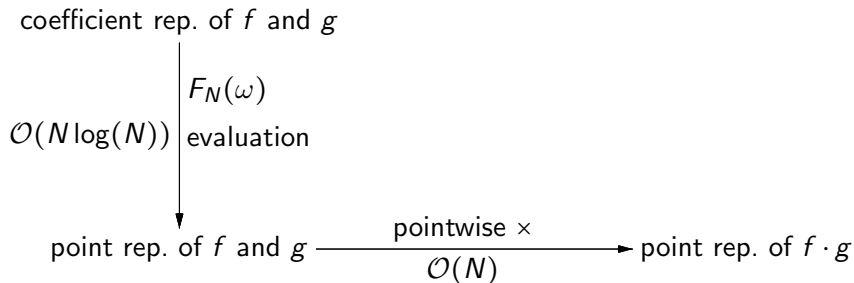
Fast Fourier transformation (FFT)

Main principle of FFT:

coefficient rep. of f and g

$\qquad \left| \begin{array}{l} F_N(\omega) \\ \text{evaluation} \end{array} \right.$

$\mathcal{O}(N \log(N))$

point rep. of f and g

Fast Fourier transformation (FFT)

Main principle of FFT:

coefficient rep. of f and g

$$F_N(\omega)$$
$$\mathcal{O}(N \log(N)) \Big| \text{ evaluation}$$

point rep. of f and g $\xrightarrow[\mathcal{O}(N)]{\text{pointwise } \times}$ point rep. of $f \cdot g$

Fast Fourier transformation (FFT)

Main principle of FFT:

coefficient rep. of f and g $\qquad\qquad\qquad$ coefficient rep. of $f \cdot g$

$\mathcal{O}(N \log(N))$ $\quad$ $\begin{array}{c} F_N(\omega) \\ \text{evaluation} \end{array}$ $\qquad\qquad$ interpolation $\begin{array}{c} \\ \frac{1}{N} F_N(\omega^{-1}) \end{array}$ $\mathcal{O}(N \log(N))$

point rep. of f and g $\xrightarrow[\mathcal{O}(N)]{\text{pointwise} \times}$ point rep. of $f \cdot g$
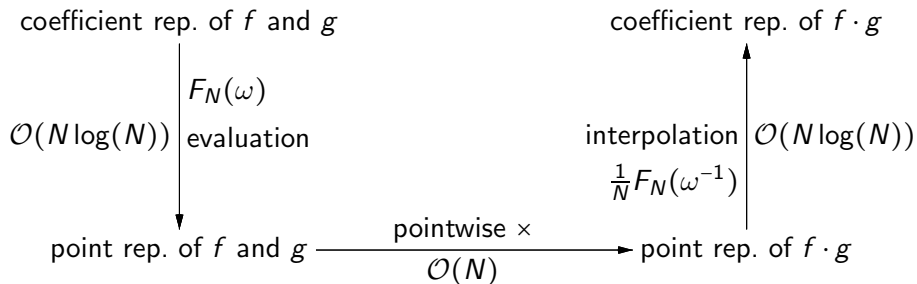
FFT over finite fields

Assume we want to use FFT in order to multiply polynomials with integer coefficients.

The main problem with computing in the field $\mathbb{C}$ are rounding errors to to appoximation of N-th roots of unity.

Recall that FFT works for every field $\mathbb{F}$ with the following properties:

- N has a multiplicative inverse in $\mathbb{F}$, i.e., the characteristic of $\mathbb{F}$ does not divide N.
- There is a primitive N-th root of unity.

Lemma 4

Let p be a prime number. The field $\mathbb{F}_p$ has an N-th primitive root of unity, if N is a divisor of $p - 1$.

FFT over finite fields

Solution of the rounding problem for $\mathbb{C}$:

- Assume that all coefficients of the product polynomial $f \cdot g$ belong to the interval $[-d, d]$.

- $N = 2^m$ is convenient for FFT.

- Search for a prime p of the form $p = 2^m \cdot k + 1$ for some $k \geq 1$ (Fourier prime) with $p > 2d$ (there are many such primes, see e.g. http://www.csd.uwo.ca/~moreno/CS874/Lectures/Newton2Hensel.html/node9.html).

- Since $\gcd(N, p) = 1$, N has a multiplicative inverse in $\mathbb{F}_p$. We can therefore do FFT in $\mathbb{F}_p$.

- This yields the coefficients of $f \cdot g$ modulo p, from which we get (because of $p > 2d$) the coefficients in $\mathbb{Z}$.

Further applications of FFT

FFT has many applications, e.g.

- fast multiplication of integers starting with Schönhage-Strassen (running time: $\mathcal{O}(n \log n \log \log n)$).

  Latest development (Harvey, van der Hoeven, March 2019): multiplication of n-bit integers in time $\mathcal{O}(n \log n)$, see
  https://web.maths.unsw.edu.au/~davidharvey/papers/nlogn/.

- pattern matching in strings

- filter algorithms in signal processing

- fast algorithms for the discrete cosine and sine transformation (used for instance in JPEG and MP3/MPEG).

FFT can be found on a list of the 10 most important algorithms of the 20th century: https://cs.gmu.edu/~henryh/483/top-10.html.

How can we test whether $A \cdot B = C$?

The best current algorithm for multiplying two $(n \times n)$-matrices needs approx. $\Theta(n^{2,372873\ldots})$ many arithmetic operations (Virginia Vassilevska Williams 2014).

Conjecture

For every $\epsilon > 0$ there exists an algorithm for multiplying two $(n \times n)$-matrices in time $\mathcal{O}(n^{2+\varepsilon})$.

Assume now that we have three $(n \times n)$-matrices A, B and C.

How many arithmetic operations are needed to test whether $A \cdot B = C$ holds?

Trivial answer: $\mathcal{O}(n^{2,372873\ldots})$

But there is a better method!

How can we test whether $A \cdot B = C$?

Theorem 5 (Korec, Wiedermann 2014)

Let A, B, C be $(n \times n)$-matrices with entries from $\mathbb{Z}$. Using $\mathcal{O}(n^2)$ many operations we can check whether $A \cdot B = C$ holds.

Proof: Let

$$
\begin{aligned}
A &= (a_{i,j})_{1 \le i,j \le n}, \\
B &= (b_{i,j})_{1 \le i,j \le n}, \\
C &= (c_{i,j})_{1 \le i,j \le n} \text{ and} \\
D &= (d_{i,j})_{1 \le i,j \le n} = A \cdot B - C.
\end{aligned}
$$

Thus, we have $A \cdot B = C$ if and only if D is the zero-matrix.

Let x be real-valued variable and consider the column-vector

$$
v(x) = (1, x, x^2, \ldots, x^{n-1})^T.
$$

How can we test whether $A \cdot B = C$?

Hence, $D \cdot v(x) = [A \cdot B - C] \cdot v(x)$ is a column-vector whose i-th entry is the polynomial

$$p_i(x) = d_{i,1} + d_{i,2}x + d_{i,3}x^2 + \cdots + d_{i,n}x^{n-1}.$$

We therefore have $A \cdot B = C$ if and only if $p_i(x)$ is the zero polynomial for all $1 \le i \le n$.

We use the following theorem:

Cauchy bound

Let $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{R}[x]$ be not the zero-polynomial and $a_n \neq 0$. For every α with $p(\alpha) = 0$ we have

$$|\alpha| < 1 + \frac{\max\{|a_i| \mid 0 \le i \le n-1\}}{|a_n|}.$$

How can we test whether $A \cdot B = C$?

Proof of the Cauchy bound:

Assume that we have already proved the Cauchy bound for the case $a_n = 1$.

Then we get the general statement as follows:

Let α be a root of $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ with $a_n \neq 0$.

Then α is also a root of the polynomial $x^n + \frac{a_{n-1}}{a_n} x^{n-1} + \cdots + \frac{a_1}{a_n} x + \frac{a_0}{a_n}$.

The Cauchy bound for the case $a_n = 1$ yields

$$|\alpha| < 1 + \max\left\{ \left| \frac{a_i}{a_n} \right| \mid 0 \leq i \leq n - 1 \right\} = 1 + \frac{\max\{|a_i| \mid 0 \leq i \leq n - 1\}}{|a_n|}.$$

How can we test whether $A \cdot B = C$?

It remains to prove the Cauchy bound for a polynomial

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0.$$

Let $h = \max\{|a_i| \mid 0 \le i \le n-1\}$.

Assume that $p(\alpha) = \alpha^n + a_{n-1}\alpha^{n-1} + \cdots + a_1\alpha + a_0 = 0$, i.e.,

$$\alpha^n = -a_{n-1}\alpha^{n-1} - \cdots - a_1\alpha - a_0. \tag{1}$$

We show that $|\alpha| < 1 + h$.

If $|\alpha| \le 1$, we have $|\alpha| < 1 + h$ (if $h = 0$ then we have $\alpha^n = 0$, i.e., $\alpha = 0$).

Now assume that $|\alpha| > 1$.

How can we test whether $A \cdot B = C$?

Using (1) and the laws $|a + b| \le |a| + |b|$, $|a \cdot b| = |a| \cdot |b|$ for all $a, b \in \mathbb{R}$, we get

$$
\begin{aligned}
|\alpha|^n &\le |a_{n-1}| \cdot |\alpha|^{n-1} + \cdots + |a_1| \cdot |\alpha| + |a_0| \\
&\le h \cdot (|\alpha|^{n-1} + \cdots + |\alpha| + 1) \\
&= h \cdot \frac{|\alpha|^n - 1}{|\alpha| - 1}.
\end{aligned}
$$

Since $|\alpha| > 1$, we obtain:

$$
|\alpha| - 1 \le h \cdot \frac{|\alpha|^n - 1}{|\alpha|^n} < h \qquad \square
$$

How can we test whether $A \cdot B = C$?

Let us define

$$
\begin{aligned}
a &= \max\{|a_{i,j}| \mid 1 \le i, j \le n\}, \\
b &= \max\{|b_{i,j}| \mid 1 \le i, j \le n\}, \\
c &= \max\{|c_{i,j}| \mid 1 \le i, j \le n\}.
\end{aligned}
$$

The absolute values of the coefficients of the polynomials $p_i(x)$ can be bounded as follows:

$$
|d_{i,j}| = |\sum_{k=1}^{n} a_{i,k} b_{k,j} - c_{i,j}| \le \sum_{k=1}^{n} |a_{i,k}| \cdot |b_{k,j}| + |c_{i,j}| \le n \cdot a \cdot b + c.
$$

Let $d = n \cdot a \cdot b + c$ and $r = 1 + d$.

How can we test whether $A \cdot B = C$?

If $p_i(x)$ is not the zero polynomial, then we can write

$$p_i(x) = d_{i,1} + d_{i,2}x + d_{i,3}x^2 + \cdots + d_{i,k}x^{k-1} \in \mathbb{Z}[x]$$

with $d_{i,k} \neq 0$.

Since A, B, C are matrices over $\mathbb{Z}$, all $d_{i,j}$ are integers and thus $|d_{i,k}| \geq 1$.

Then, by the Cauchy bound, every root α of $p_i(x)$ satisfies

$$|\alpha| < 1 + \frac{\max\{|d_{i,j}| \mid 1 \leq j \leq k-1\}}{|d_{i,k}|} \leq 1 + d = r.$$

Hence, for all $1 \leq i \leq n$ we have

$$p_i(x) = 0 \iff p_i(r) = 0.$$

How can we test whether $A \cdot B = C$?

We can therefore check with the following algorithm, whether $A \cdot B = C$:

1. Compute a, b, c, and $r = 1 + n \cdot a \cdot b + c$ according to Slide 27.
   ($3n^2$ many comparisons, 2 additions, 2 multiplications)

2. Compute the column vector $u = (1, r, r^2, \ldots, r^{n-1})^T = v(r)$.
   ($n - 2$ multiplications)

3. Compute the vectors $p := B \cdot u$, $s := A \cdot p$ and $t := C \cdot u$
   ($\mathcal{O}(n^2)$ many arithmetic operations)

4. We obtain

$$
\begin{aligned}
s = t \quad &\Longleftrightarrow \quad A \cdot B \cdot v(r) = C \cdot v(r) \\
&\Longleftrightarrow \quad [A \cdot B - C] \cdot v(r) = 0 \\
&\Longleftrightarrow \quad p_i(r) = 0 \text{ for all } 1 \le i \le n \\
&\Longleftrightarrow \quad p_i(x) = 0 \text{ for all } 1 \le i \le n \\
&\Longleftrightarrow \quad A \cdot B = C
\end{aligned}
$$

$\square$

Parallel algorithms and NC

1. Introduction to parallel architectures
2. The class NC and parallel matrix multiplication
3. Parallel computation of prefix sums
4. Parallel integer addition, multiplication and division
5. Parallel computation of determinants

Introduction to parallel architectures

- Parallel random access machines (PRAM)
    - CRCW (concurrent read concurrent write)
    - CREW (concurrent read exclusive write)
    - EREW (exclusive read exclusive write)
    - ERCW (exclusive read concurrent write)
- Vector machines
    - SIMD or MIMD
- Boolean and arithmetic circuits
    - DAG (directed acyclic graph) with input and output gates
    - gates for basic boolean / arithmetic operations

Example for a boolean circuit

Gates on the same level can be evaluated in parallel.
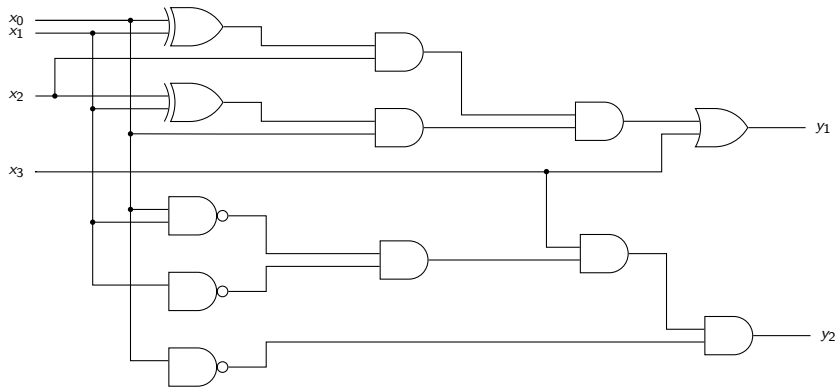
Parallel time corresponds to depth (number of gates on a longest path from an input to an output) of the circuit (here 4).

The class NC

NC denotes the class of all problems that can be "efficiently parallelized".

NC stands for "Nick's Class" (after Nick Pippenger).

Definition(s):

- All problems that can solved by a PRAM with $n^{\mathcal{O}(1)}$ processors in time $(\log n)^{\mathcal{O}(1)}$.
- All problems that can be solved by boolean circuits of depth $(\log n)^{\mathcal{O}(1)}$ and $n^{\mathcal{O}(1)}$ gates (one circuit for every bit length n of the input).

The class is robust against small changes in the machine model.
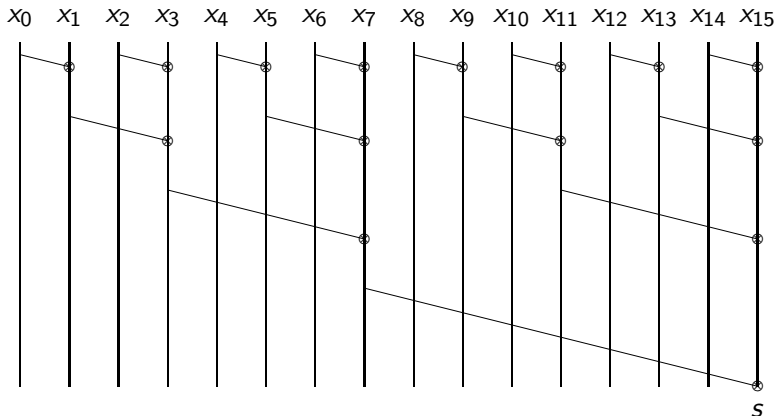
The question $NC \stackrel{?}{=} P$ is open.

There are so-called P-complete problems (e.g. evaluation of boolean circuits) that belong to P, but most people belief that they do not belong to NC.

Compute a sum of n numbers

Example

The sum $s = \sum_{i=0}^{n-1} x_i$ can be computed with n processors in time $\log n$.

The balanced binary tree technique:

Parallel matrix multiplication

Theorem 6

The product of two $(n \times n)$-matrices can be computed with n^3 processors in time $1 + \log n$.

Proof: Let $A = (a_{ij})$ and $B = (b_{ij})$.

We have $(A \cdot B)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

- Compute with n^3 processors all n^3 products $a_{ik} b_{kj}$.
- Assign n processors to each of the n^2 sums.
- Compute in time $\log n$ all n^2 sums (using the balanced binary tree technique). $\qquad\square$

Parallel prefix sums

The problem:

- Input: x_i $(0 \le i \le n-1)$
- Output: all prefix sums $\sum_{j=0}^{i} x_j$ for all $0 \le i \le n-1$

Theorem 7

All prefix sums can be computed with n processors in time $\log n$.

Proof:. Let $x_i = 0$ for all $i < 0$.

for $d := 0$ to $\lceil \log n \rceil - 1$ **do**
    for all $i \in \{0, \dots, n-1\}$ **do in parallel**
        $x_i := x_{i-2^d} + x_i$
    endfor
endfor

Parallel prefix sums

Let $x_{i,k}$ be the value of the variable x_i after the k-th iteration of the outer for-loop (note that $d = k - 1$ in the k-th iteration)

By induction on k we show: $x_{i,k} = \sum_{j=i-2^k+1}^{i} x_j$.

For $k = 0$ we have $x_{i,0} = x_i = \sum_{j=i}^{i} x_j = \sum_{j=i-2^0+1}^{i} x_j$.

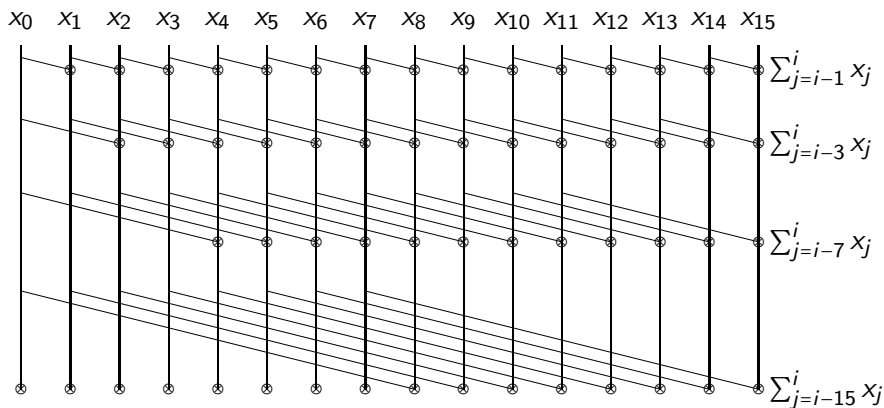Now assume that $x_{i,k-1} = \sum_{j=i-2^{k-1}+1}^{i} x_j$.

Then after the k–th iteration of the outer for-loop we have:

$$
\begin{aligned}
x_{i,k} &= x_{i-2^{k-1},k-1} + x_{i,k-1} \\
&= \sum_{j=i-2^{k-1}-2^{k-1}+1}^{i-2^{k-1}} x_j + \sum_{j=i-2^{k-1}+1}^{i} x_j = \sum_{j=i-2^k+1}^{i} x_j
\end{aligned}
$$

Parallel prefix sums

Finally, for $k = \lceil \log n \rceil$ we have for all $0 \le i \le n-1$:

$$x_{i, \lceil \log n \rceil} = \sum_{j=i-2^{\lceil \log n \rceil}+1}^{i} x_j = \sum_{j=0}^{i} x_j.$$

Parallel Integer Arithmetic

In the following we want to find efficient parallel algorithms for the important arithmetic operations on integers:

- addition (basically the same algorithm also works for subtraction)

- multiplication

- division with remainder

The size parameter n is always the number of bits of the two input integers a and b.

The elementary operations carried out by the processors are bit manipulations (computing the logical and, or, xor of two bits).

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rl}
a & = \quad 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = \quad 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry & = \\
\hline
sum & =
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry & = & 1\,0 \\
\hline
sum & = & 0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a &=& 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b &=& 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry &=& 1\,1\,0 \\
\hline
sum &=& 0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry & = & 0\,1\,1\,0 \\
\hline
sum & = & 1\,0\,0 \\
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel
algorithm!

Example:

$$
\begin{array}{rl}
a & = \quad 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = \quad 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\mathit{carry} & = \qquad\qquad\qquad\qquad 0\,0\,1\,1\,0 \\
\hline
\mathit{sum} & = \qquad\qquad\qquad\qquad\quad 1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel
algorithm!

Example:

$$
\begin{array}{rl}
a & = \quad 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = \quad 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry & = \qquad\qquad\qquad\quad 0\,0\,0\,1\,1\,0 \\
\hline
sum & = \qquad\qquad\qquad\qquad 1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & \hphantom{1\,1\,0\,1\,0\,1\,0\,0\,1}0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & \hphantom{1\,1\,0\,1\,0\,1\,0\,0\,1\,0}1\,1\,1\,1\,0\,0 \\
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\mathit{carry} & = & 1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
\mathit{sum} & = & 0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
\hline
carry & = & 0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 1\,0\,0\,1\,1\,1\,1\,0\,0 \\
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel
algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 1\,0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 0\,0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 0\,1\,1\,0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Adding binary integers

The school method for adding binary integers is not a good parallel algorithm!

Example:

$$
\begin{array}{rcl}
a & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b & = & 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
carry & = & 1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0 \\
\hline
sum & = & 1\,0\,1\,1\,0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0 \\
\end{array}
$$

Integer Addition in NC

Theorem 8

Two binary n-bit integers can be added with n processors in time $\mathcal{O}(\log n)$.

Proof: Let $a = (a_{n-1}\cdots a_1 a_0)_2$ and $b = (b_{n-1}\cdots b_1 b_0)_2$ be the input number (least significant bit = right-most bit).

Step 1: Compute with n processors in time $\mathcal{O}(1)$ the carry propagation string $c_n c_{n-1}\cdots c_2 c_1 c_0$:

$$c_i = \begin{cases} 0 & a_{i-1} = b_{i-1} = 0 \ \lor \ i = 0 \\ 1 & a_{i-1} = b_{i-1} = 1 \\ p & \text{else} \end{cases}$$

Integer Addition in NC

Step 2: Compute with n processors in time $\mathcal{O}(\log n)$ carry_i from c_i with the parallel prefix sum algorithm applied to the following binary associative operation:

$$
\begin{aligned}
0 \cdot x &= 0 \\
1 \cdot x &= 1 \\
p \cdot x &= x
\end{aligned}
$$

Note: the parallel prefix sum algorithm works for any binary associative operation $((x \cdot y) \cdot z = x \cdot (y \cdot z))$ instead of $+$.

Step 3: Compute the i-th bit of the sum $a + b$ as the XOR (exclusive or) of a_i, b_i and carry_i (where $a_n = b_n = 0$). $\qquad\square$

Recall: $0 \text{ xor } 0 = 1 \text{ xor } 1 = 0$, $0 \text{ xor } 1 = 1 \text{ xor } 0 = 1$

Integer Addition in NC

Example:

$$
\begin{array}{rcl}
a &=& 1\,0\,0\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1 \\
b &=& 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1 \\
c &=& 1\,p\,0\,1\,0\,1\,0\,p\,1\,p\,p\,p\,0\,p\,1\,0 \\
\underline{carry} &=& \underline{1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0} \\
sum &=& 1\,0\,1\,1\,0\,1\,0\,1\,0\,0\,1\,1\,1\,1\,0\,0
\end{array}
$$

Integer Multiplikation in NC

Next goal: compute product of $a = (a_{n-1} \cdots a_1 a_0)_2$, $b = (b_{n-1} \cdots b_1 b_0)_2$.

Note: $a \cdot b$ has at most $2n$ bits.

Reduction to iterated integer addition:

$$a \cdot b = \Big(\sum_{i=0}^{n-1} a_i \cdot 2^i \Big) \cdot b = \sum_{i=0}^{n-1} (a_i \cdot 2^i \cdot b)$$

Note: $2^i \cdot b = (b_{n-1} \cdots b_1 b_0 \underbrace{0 \cdots 0}_{i \text{ many}})_2$ (i-fold shift) is easy to compute.

From Theorem 8 we get: two n-bit integers can be multiplied with n^2 processors in time $(\log n)^2$ (use a balanced binary tree of depth $\log n$ where in each node two integers of bit length at most $2n$ are added).

Integer Multiplication in NC

Lemma 9

From three n-bit integers a, b, c we can compute with n processors in time $\mathcal{O}(1)$ two $(n+1)$-bit integers d, e such that $a + b + c = d + e$.

Proof:

```
          100111
          011100
         +111101
              10
             01
             11      111101
             10     +000110
             10
         +10
```

Integer Multiplication in NC

Theorem 10

The sum n many n-bit integers can be computed with n^2 processors in time $\mathcal{O}(\log n)$.

Proof:

- Divide the n input numbers $a_1, a_2, \ldots, a_n$ into blocks of three numbers (and possibly one block of at most two numbers).
- For each block $[a_i, a_{i+1}, a_{i+2}]$ ($n/3$ many) compute in constant time with $\mathcal{O}(n^2)$ processors two numbers (with at most $n + 1$ bits) whose sum is $a_i + a_{i+1} + a_{i+2}$.
- In this way, we obtain $(n + 1)$-bit numbers $b_1, b_2, \ldots, b_m$ with $m \approx 2n/3$ such that $\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} b_i$.
- After roughly $\log_{3/2}(n)$ iterations of this step only two numbers x_1, x_2 with roughly $n + \log_{3/2}(n)$ bits such that $\sum_{i=1}^{n} a_i = x_1 + x_2$ remain.
- Compute $x_1 + x_2$ in time $\mathcal{O}(\log n)$. $\qquad\square$

Integer Multiplication in NC

Using the reduction from Slide 44 we get:

Corollary

Two n-bit integers can be multiplied with n^2 processors in time $\mathcal{O}(\log n)$.

Integer Division in NC

Goal: Compute for two given integers $s, t > 0$ with at most n bits the unique numbers q and r such that $s = qt + r$ and $0 \le r < t$.

We will accomplish this in time $\mathcal{O}((\log n)^2)$ with $\mathcal{O}(n^4)$ processors.

Main tool: Newton approximation for roots.

Let $f : \mathbb{R} \to \mathbb{R}$.

Guess an initial value x_0 and compute the sequence $(x_i)_{i \ge 0}$ using the recursion

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \text{ where } f' = df/dx$$

If you are lucky this sequence $(x_i)_{i \ge 0}$ converges to a root of f (a value y with $f(y) = 0$).

Integer Division in NC

Take $f(x) = t - \frac{1}{x}$. Hence, $\frac{1}{t}$ is the unique root of f.

$f'(x) = \frac{1}{x^2}$, hence Newton's recursion becomes

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} = 2x_i - tx_i^2$$

Let x_0 be the unique number of the form $\frac{1}{2^j}$ $(j \geq 0)$ in the interval $(\frac{1}{2t}, \frac{1}{t}]$.

We compute x_0 in time $\mathcal{O}(1)$ using n processors as follows: find the unique power of two 2^j in the interval $[t, 2t)$, reverse the order of bits and place a decimal point after the first 0.

Example: $t = 11011 \rightarrow 100000 \rightarrow 0.00001$

Integer Division in NC

Lemma 11

The sequence $(x_i)_{i \geq 0}$ with $x_{i+1} = 2x_i - tx_i^2$ and x_0 the unique number of the form $\frac{1}{2^j}$ in the interval $(\frac{1}{2t}, \frac{1}{t}]$ satisfies $0 \leq 1 - t \cdot x_i < \frac{1}{2^{(2^i)}}$.

Proof: Induction over i:

By definition of x_0 we have $\frac{1}{2t} < x_0 \leq \frac{1}{t}$, i.e., $0 \leq 1 - tx_0 < \frac{1}{2}$.

For $i \geq 0$ we obtain $1 - t \cdot x_{i+1} = 1 - t(2 \cdot x_i - t \cdot x_i^2) = (1 - t \cdot x_i)^2$.

Since by the induction hypothesis we have $0 \leq 1 - t \cdot x_i < \frac{1}{2^{(2^i)}}$, we get

$$0 \leq 1 - t \cdot x_{i+1} = (1 - t \cdot x_i)^2 < \left(\frac{1}{2^{(2^i)}}\right)^2 = \frac{1}{2^{(2^{i+1})}}.$$

$\square$

Integer Division in NC

From the previous lemma we obtain for $k = \lceil \log(\log(s)) \rceil \in \mathcal{O}(\log n)$:

$$0 \le 1 - t \cdot x_k < \frac{1}{2^{(2^{\lceil \log(\log(s)) \rceil})}} \le \frac{1}{s} \le \frac{t}{s}.$$

Hence, $0 \le s - s \cdot x_k \cdot t < t$, i.e., $0 \le \frac{s}{t} - s \cdot x_k < 1$.

It follows that the integer part $q = \lfloor \frac{s}{t} \rfloor$ of $\frac{s}{t}$ is either $\lceil s \cdot x_k \rceil$ or $\lfloor s \cdot x_k \rfloor$ (the correct value can be found by a test involving a single multiplication).

The remainder r can be computed as $r = s - qt$.

Integer Division in NC

Estimate for the parallel time: let b_i be the number of bits of x_i.

Recall: $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ and $b_0 \le n + 1$.

$2 \cdot x_i$ has at most $b_i + 1$ bits, $t \cdot x_i^2$ has at most $n + 2b_i$ bits.

We obtain $b_{i+1} \le 2b_i + n + 1$ and hence $b_i \le (2^{i+1} - 1) \cdot (n + 1) \in \mathcal{O}(2^i n)$.

Theorefore, x_k (and all x_i with $i < k$) have at most
$\mathcal{O}(2^{\lceil \log(\log(s)) \rceil} n) \le \mathcal{O}(\log(s) \cdot n) \le \mathcal{O}(n^2)$ many bits.

It follows that the computation of $x_{i+1} = 2 \cdot x_i - t \cdot x_i^2$ from x_i needs time
$\mathcal{O}(\log(n))$ with $\mathcal{O}(n^4)$ processors.

Since $k \in \mathcal{O}(\log(n))$, the total parallel running time is $\mathcal{O}((\log n)^2)$.

Csansky's algorithm for inverting matrices

Goal: An NC-algorithm for inverting an $(n \times n)$-matrix (if the matrix is invertible).

Convention: In the following considerations we assume that a single processor can carry out a single arithmetic operation (addition, multiplication, division with remainder) in time $\mathcal{O}(1)$.

Our previous results show that with this assumption we stay in the class NC (assuming that all numbers that arise during the computations have at most $\mathcal{O}(n^c)$ bits for some constant c).

Remark: If A is an $(n \times m)$-matrix and B is an $(m \times p)$-matrix, then we can compute $A \cdot B$ with $n \cdot m \cdot p$ processors in time $\mathcal{O}(\log(m))$.

Mathematical background: permutations

A permutation on $[1, n] := \{1, \ldots, n\}$ is a bijection $\sigma : [1, n] \to [1, n]$.

The set of all permutations of $[1, n]$ is denoted with S_n.

Composition of functions yields a natural product operation on S_n:

For $\sigma, \tau \in S_n$ we denote with $\sigma\tau$ the permutation such that for all $i \in [1, n]$: $(\sigma\tau)(i) = \tau(\sigma(i))$.

A transposition (on $[1, n]$) is a permutation σ that swaps two elements $i, j \in [1, n]$ with $i \neq j$:

- $\sigma(i) = j$

- $\sigma(j) = i$

- $\sigma(k) = k$ for all $k \in [1, n] \setminus \{i, j\}$

We write this transposition as (i, j) or equivalently (j, i).

Mathematical background: permutations

Every permutation σ be written as a product of transpositions.

Example: Consider the permutation $\sigma : [1,8] \to [1,8]$ with

a	1	2	3	4	5	6	7	8
$\sigma(a)$	5	4	7	2	3	8	1	6

We have $\sigma = (1,7)(3,5)(5,7)(2,4)(6,8)$ (the transpositions are evaluated from left to right).

The sign of a permutation σ is

$$\mathrm{sign}(\sigma) = \begin{cases} +1 & \text{if } \sigma \text{ is a product of an even number of transpositions} \\ -1 & \text{if } \sigma \text{ is a product of an odd number of transpositions} \end{cases}$$

One cannot write a permutation as a product of an even number of transpositions and at the same time write it as a product of an odd number of transpositions!

Mathematical background: determinants

For computing the determinant of a matrix, we can use the famous Leibniz formula:

Leibniz formula for the determinant

Let $A = (a_{i,j})_{1 \leq i,j \leq n}$ be an $(n \times n)$-matrix. We have

$$\det(A) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

Recall: a matrix A is invertible if and only if $\det(A) \neq 0$.

Csansky's algorithm for inverting matrices

Step 1: Inverting a lower triangular matrix.

A matrix is a lower triangular matrix, if (i) all entries above the main diagonal are 0 and (ii) all entries on the main diagonal are non-zero.

Example: The following matrix is lower triangular:

$$\begin{pmatrix} -3 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ -4 & 7 & 1 & 2 \end{pmatrix}$$

Csansky's algorithm for inverting matrices

Consider an lower triangular $(n \times n)$-matrix $A = \begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$.

B and D are lower triangular $(\frac{n}{2} \times \frac{n}{2})$-matrices and C is a $(\frac{n}{2} \times \frac{n}{2})$-matrix.

We have $A^{-1} = \begin{pmatrix} B^{-1} & 0 \\ -D^{-1}CB^{-1} & D^{-1} \end{pmatrix}$ (check it!).

This identity leads to a parallel algorithm for computing A^{-1} with running time $T(n) = T(\frac{n}{2}) + \mathcal{O}(\log(n))$ using n^3 processors.

We get $T(n) \in \mathcal{O}(\log^2(n))$: If $T(n) \leq T(\frac{n}{2}) + c \cdot \log_2(n)$ then

$$T(n) \leq c \cdot (\log_2(n))^2 + T(1) \in \mathcal{O}(\log^2(n)).$$

Csansky's algorithm for inverting matrices

Step 2: Solve a particular system of linear equations.

Consider a system of linear equations of the form

$$x_1 = c_1$$
$$x_2 = a_{2,1}x_1 + c_2$$
$$x_3 = a_{3,1}x_1 + a_{3,2}x_2 + c_3$$
$$\vdots$$
$$x_n = a_{n,1}x_1 + a_{n,2}x_2 + \cdots a_{n,n-1}x_{n-1} + c_n$$

The x_i are the indeterminates, the c_i and $a_{i,j}$ are integers.

Let $A = (a_{i,j})_{1 \le i,j \le n}$, where $a_{i,j} = 0$ for $i \le j$ and let $c = (c_1, \ldots, c_n)^T$.

Csansky's algorithm for inverting matrices

The system of linear equations is equivalent to $Ax + c = x$, i.e.,
$(A - \text{Id})x = -c$, where Id is the $(n \times n)$ identity matrix.

We get $x = (\text{Id} - A)^{-1}c$.

Since $\text{Id} - A$ is a lower triangular matrix, we can compute $(\text{Id} - A)^{-1}$ with
n^3 processors in time $\mathcal{O}(\log^2(n))$.

Hence, x can be computed with n^3 processors in time $\mathcal{O}(\log^2(n))$.

Csansky's algorithm for inverting matrices

Step 3 (the main step): Compute the characteristic polynomial.

The characteristic polynomial of an $(n \times n)$-matrix $A = (a_{i,j})_{1 \le i,j \le n}$ is

$$\det(x \cdot \mathrm{Id} - A) = x^n - s_1 x^{n-1} + s_2 x^{n-2} - \cdots + (-1)^n s_n =$$
$$= \prod_{i=1}^{n} (x - \lambda_i).$$

The $\lambda_1, \ldots, \lambda_n \in \mathbb{C}$ are the eigenvalues of A ($\lambda_i = \lambda_j$ for $i \ne j$ is allowed).

The number of times a certain λ_i occurs in this list is the multiplicity $\mathrm{mult}_A(\lambda_i)$ of the eigenvalue λ_i.

The coefficient s_1 is the trace of A (the first equality can be deduced from the Leibniz formula – do it!):

$$s_1 = \mathrm{tr}(A) = \sum_{i=1}^{n} a_{i,i} = \sum_{i=1}^{n} \lambda_i$$

Csansky's algorithm for inverting matrices

By substituting $x = 0$ in the characteristic polynomial we obtain

$$s_n = (-1)^n \det(-A) = \det(A) = \prod_{i=1}^{n} \lambda_i.$$

In general, we have for all $1 \le i \le n$: $s_k = \sum_{1 \le i_1 < \cdots < i_k \le n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k}$.

Lemma 12

λ_i^m is an eigenvalue of A^m with $\text{mult}_{A^m}(\lambda_i^m) = \text{mult}_A(\lambda_i)$.

Proof: excercise

We obtain $\text{tr}(A^m) = \sum_{i=1}^{n} \lambda_i^m$.

Csansky's algorithm for inverting matrices

Example: Let

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 3 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

We have

$$
\begin{aligned}
\det(x \cdot \mathsf{Id} - A) &= \det \begin{pmatrix} x-2 & -1 & -1 \\ 0 & x-3 & -1 \\ 0 & -1 & x+1 \end{pmatrix} \\
&= (x-2) \cdot \det \begin{pmatrix} x-3 & -1 \\ -1 & x+1 \end{pmatrix} \\
&= (x-2) \cdot ((x-3)(x+1) - 1) \\
&= x^3 - 4x^2 + 8
\end{aligned}
$$

We have $\det(A) = -8$, $\mathsf{tr}(A) = 4$, and the eigenvalues are 2, $1 + \sqrt{5}$ and $1 - \sqrt{5}$ (and all of them have multiplicity one).

Csansky's algorithm for inverting matrices

Let $f_k^m = \displaystyle\sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \notin \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m$

We get $f_k^0 = (n-k)s_k$, $f_0^m = \mathrm{tr}(A^m)$ and (for $k \ge 1$)

$$
\begin{aligned}
s_k \cdot \mathrm{tr}(A^m) &= \left(\sum_{1 \le i_1 < \cdots < i_k \le n} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \right) \cdot \sum_{j=1}^{n} \lambda_j^m \\
&= \sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \notin \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m + \sum_{\substack{1 \le i_1 < \cdots < i_k \le n \\ j \in \{i_1, \ldots, i_k\}}} \lambda_{i_1} \lambda_{i_2} \cdots \lambda_{i_k} \lambda_j^m \\
&= f_k^m + f_{k-1}^{m+1}
\end{aligned}
$$

Csansky's algorithm for inverting matrices

From this we obtain

$$
\begin{aligned}
s_k \cdot \mathrm{tr}(A^0) &- s_{k-1} \cdot \mathrm{tr}(A^1) + s_{k-2} \cdot \mathrm{tr}(A^2) - \cdots \\
&+ (-1)^{k-1} s_1 \cdot \mathrm{tr}(A^{k-1}) + (-1)^k \mathrm{tr}(A^k) \\
= (f_k^0 + f_{k-1}^1) &- (f_{k-1}^1 + f_{k-2}^2) + \cdots \\
&+ (-1)^{k-1}(f_1^{k-1} + f_0^k) + (-1)^k f_0^k \\
= f_k^0 = (n &- k) s_k
\end{aligned}
$$

and therefore (recall that $\mathrm{tr}(A^0) = n$)

$$
s_k = \frac{1}{k} \left(s_{k-1} \mathrm{tr}(A^1) - s_{k-2} \mathrm{tr}(A^2) + \cdots \right.
$$
$$
\left. - (-1)^{k-1} s_1 \mathrm{tr}(A^{k-1}) - (-1)^k \mathrm{tr}(A^k) \right).
$$

Csansky's algorithm for inverting matrices

Therefore, we can compute the coefficients s_k of the characteristic polynomial as follows:

1. Compute all powers $A^1, A^2, \cdots, A^n$ in time $\mathcal{O}(\log^2(n))$ using n^4 processors (use prefix sum algorithm for input $A, A, \ldots, A$).

2. Compute $\text{tr}(A^1), \ldots, \text{tr}(A^n)$ in time $\mathcal{O}(\log(n))$ using n^2 processors.

3. Compute the s_i in time $\mathcal{O}(\log^2(n))$ using n^3 processors by solving a system of linear equations of the form considered on Slide 59.

Remark: The last step is only possible of the characteristic p of the underlying field is zero or larger than n. This ensures that $\frac{1}{k}$ exists for all $1 \le k \le n$.

Csansky's algorithm for inverting matrices

Step 4: Inverting an arbitrary non-singular matrix.

Theorem of Cayley-Hamilton

Every square matrix satisfies it own characteristic equation:

$$A^n - s_1 \cdot A^{n-1} + s_2 \cdot A^{n-2} - \cdots + (-1)^{n-1} s_{n-1} \cdot A + (-1)^n s_n \cdot \text{Id} = 0$$

If A^{-1} exists (this is equivalent to $s_n \neq 0$) we have:

$$A^{-1} = \frac{(-1)^{n-1}}{s_n} \left(A^{n-1} - s_1 A^{n-2} + s_2 A^{n-3} - \cdots + (-1)^{n-1} s_{n-1} \cdot \text{Id} \right).$$

Therefore we can compute A^{-1} in time $\mathcal{O}(\log^2(n))$ using n^4 processors by (i) computing all coefficients s_k and (ii) computing the above expression for A^{-1} in time $\mathcal{O}(\log(n))$ using n^2 processors.

Randomisierte algorithms

A randomized algorithm (or probabilistic algorithm) uses random decisions (it tosses a coin).

Examples:

- Quicksort with a randomly chosen pivot element
- Quickselect for computing the median

One distinguishes the following types of randomized algorithms:

- Las Vegas algorithms: They yield a correct result with probability one, but the running time (or space) is a random variable.

  Example: Quicksort with a randomly chosen pivot element has an expected running time of $O(n \log n)$.

- Monte Carlo algorithms: They can yield a faulty result with a small error probability.

Probabilistic tests with polynomials

Let $\mathbb{F}$ be a field at let $x_1, \ldots, x_n$ variables.

With $\mathbb{F}[x_1, \ldots, x_n]$ we denote the ring of all polynomials in the variables $x_1, \ldots, x_n$ and with coefficients from $\mathbb{F}$.

Let $a_1, \ldots, a_k \in \mathbb{F}$, $e_{i,j} \geq 0$ for $1 \leq i \leq k$, $1 \leq j \leq n$ and

$$p(x_1, \ldots, x_n) = \sum_{i=1}^{k} a_i \prod_{j=1}^{n} x_j^{e_{i,j}} \in \mathbb{F}[x_1, \ldots, x_n].$$

The degree of p is $\deg(p) = \max\{e_{i,1} + e_{i,2} + \cdots + e_{i,n} \mid 1 \leq i \leq k\}$.

We have $\deg(p \cdot q) = \deg(p) + \deg(q)$.

Example: $p = 2x_1^2 x_4^7 - 4x_2^5 x_3 x_4^2 - x_2 x_3 x_4^3 + 5x_3 x_4^2 - 8$.

We have $\deg(p) = 9$ (due to $x_1^2 x_4^7$).

Probabilistic tests with polynomials

Theorem 13

Let $S \subseteq \mathbb{F}$ finite and $p(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n] \smallsetminus \{0\}$, i.e., p is not the zero polynomial.

The equation $p(x_1, \ldots, x_n) = 0$ has at most $\deg(p) \cdot |S|^{n-1}$ solutions in S^n.

Proof: Induction over n and $\deg(p)$.

Case 1: $n = 1$, i.e. p is a polynomial in a single variable (this includes the case $\deg(p) = 0$, where $p \in \mathbb{F} \smallsetminus \{0\}$).

A polynomial p in a single variable has at most $\deg(p) = \deg(p) \cdot |S|^{n-1}$ roots in S.

Probabilistic tests with polynomials

Case 2: $\deg(p) = 1$, i.e. p has the form $a + a_1x_1 + \cdots + a_nx_n$.

Since $\deg(p) \neq 0$, there is an i with $a_i \neq 0$. W.l.o.g. assume that $a_1 \neq 0$.

The equation $p(x_1, \ldots, x_n) = 0$ is equivalent to $x_1 = \dfrac{1}{a_1}\Big(-a - \sum_{i=2}^{n} a_ix_i\Big)$.

There are exactly $|S|^{n-1}$ assignments for $x_2, \ldots, x_n$ with values from S.

Therefore, $p = 0$ has at most $|S|^{n-1} = \deg(p) \cdot |S|^{n-1}$ solutions in S^n.

Case 3: $\deg(p) \geq 2$ and $n \geq 2$.

Case 3.1: p is not irreducible, i.e., $p = q \cdot r$ with $\deg(q) < \deg(p)$ and $\deg(r) < \deg(p)$.

Neither q nor r is the zero polynomial.

Probabilistic tests with polynomials

By induction, the equation $q = 0$ has at most $\deg(q) \cdot |S|^{n-1}$ solutions in S^n and the equation $r = 0$ has at most $\deg(r) \cdot |S|^{n-1}$ solutions in S^n.

Since $(a_1, \ldots, a_n) \in S^n$ is a solution of $q \cdot r = 0$ if and only if it is a solution of $q = 0$ or a solution of $r = 0$, the equation $p = 0$ has at most

$$\deg(q) \cdot |S|^{n-1} + \deg(r) \cdot |S|^{n-1} = (\deg(q) + \deg(r)) \cdot |S|^{n-1} = \deg(p) \cdot |S|^{n-1}$$

solutions in S^n.

Fall 3.2: p is irreducible.

Let $\overline{x} = (x_1, \ldots, x_{n-1})$, i.e.,. $p = p(\overline{x}, x_n)$.

For each $s \in S$ we consider the polynomial $p(\overline{x}, s) \in \mathbb{F}[\overline{x}]$.

Claim: $p(\overline{x}, s)$ is not the zero polynomial.

Probabilistic tests with polynomials

In order to prove this claim, we assume that $p(\overline{x}, s) = 0$.

Write $p(\overline{x}, x_n)$ as a polynomial in the single variable x_n and with coefficients from $\mathbb{F}[\overline{x}]$ (the set of these polynomials is $\mathbb{F}[\overline{x}][x_n]$):

$$p(\overline{x}, x_n) = \sum_{k=0}^{m} t_k(\overline{x}) \cdot x_n^k \tag{2}$$

Let $\deg_{x_n}(p) = m$ (the degree of p in the variable x_n).

In our example $p = 2x_1^2 x_4^7 - 4x_2^5 x_3 x_4^2 - x_2 x_3 x_4^3 + 5x_3 x_4^2 - 8$ we have

$$p = (2x_1^2) \cdot x_4^7 + (-4x_2^5 x_3 + 5x_3) \cdot x_4^2 + (-x_2 x_3 x_4^3 - 8) \in \mathbb{F}[x_1, x_2, x_3][x_4].$$

Polynomial division with remainder by $x_n - s$ yields

$$p(\overline{x}, x_n) = q(\overline{x}, x_n) \cdot (x_n - s) + r(\overline{x}), \tag{3}$$

where $\deg_{x_n}(r) = 0$.

Probabilistic tests with polynomials

To see this, note that

$$
\begin{aligned}
x_n^k &= x_n^k - s^k + s^k \\
&= \sum_{i=1}^{k} s^{i-1} x_n^{k-(i-1)} - \sum_{i=1}^{k} s^i x_n^{k-i} + s^k \\
&= \Big(\sum_{i=1}^{k} s^{i-1} x_n^{k-i} \Big) \cdot (x_n - s) + s^k
\end{aligned}
$$

and apply this to all terms $t_k(\overline{x}) \cdot x_n^k$ in (2).

Setting $x_n = s$ in (3) yields $r(\overline{x}) = p(\overline{x}, s) = 0$.

Hence, we have $p(\overline{x}, x_n) = q(\overline{x}, x_n) \cdot (x_n - s)$.

Contradiction to the irreducibility of p ($\deg(p) \geq 2$ is important here)!

Probabilistic tests with polynomials

This proves the claim $p(\overline{x}, s) \neq 0$.

Since $p(\overline{x}, s)$ is not the zero polynomial we can use the induction hypothesis for $p(\overline{x}, s)$:

Hence, $p(\overline{x}, s) = 0$ has at most $\deg(p(\overline{x}, s)) \cdot |S|^{n-2} \leq \deg(p) \cdot |S|^{n-2}$ solutions in S^{n-1}.

Since there are only $|S|$ different values for s, the equation $p(\overline{x}, x_n) = 0$ has at most $|S| \cdot \deg(p) \cdot |S|^{n-2} = \deg(p) \cdot |S|^{n-1}$ solutions in S^n.

This concludes the proof of Theorem 13. $\qquad\qquad\square$

In the following theorem we choose the tuple $(s_1, \ldots, s_n) \in S^n$ randomly according to the uniform distribution.

This means that we assign the same probability $1/|S|^n$ to every tuple $(s_1, \ldots, s_n) \in S^n$.

Probabilistic tests with polynomials

Theorem of Zippel and Schwartz

Let $p(x_1, \ldots, x_n)$ be a non-zero polynomial with coefficients from a field $\mathbb{F}$, and let $S \subseteq \mathbb{F}$ be finite.

If we choose $(s_1, \ldots, s_n) \in S^n$ randomly according to the uniform distribution, then

$$\text{Prob}[p(s_1, \ldots, s_n) = 0] \leq \frac{\deg(p)}{|S|}.$$

Proof: There are in total $|S|^n$ choices for $(s_1, \ldots, s_n)$, but by Theorem 13 at most $\deg(p) \cdot |S|^{n-1}$ of these choices satisfy $p(s_1, \ldots, s_n) = 0$.

Hence, we have $\text{Prob}[p(s_1, \ldots, s_n) = 0] \leq \dfrac{\deg(p) \cdot |S|^{n-1}}{|S|^n} = \dfrac{\deg(p)}{|S|}$.

Application: perfect matchings

Let $G = (V, E)$ be a finite undirected graph with node set V and edge set E.

Formally, an edge $e \in E$ is a set $e = \{u, v\}$ with $u, v \in V$ and $u \neq v$.

A matching of G is a subset $M \subseteq E$ such that $e \cap e' = \varnothing$ for all $e, e' \in M$ with $e \neq e'$.

A matching M of G is a perfect matching, if $|M| = |V|/2$. This means that every node of G belongs to exactly one edge of M.

Obviously, a graph can only have a perfect matching if it has an even number of nodes.

Application: perfect matchings
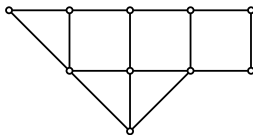
Example:

Application: perfect matchings

Example: a matching, which is not perfect.
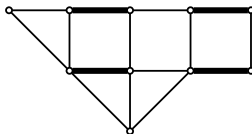
Application: perfect matchings

Example: a perfect matching
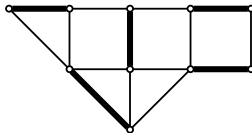
Application: perfect matchings

We will develop a randomized NC-algorithm for testing, whether a given graph has a perfect matching.

For this we will construct a polynomial, which is not the zero polynomial if and only if G has a perfect matching.

Remark: There exist deterministic polynomial time algorithms for testing whether a given graph has a perfect matching, but it is not known whether there exists a deterministic NC-algorithm for this problem.

Application: perfect matchings

Let $G = (V, E)$ be an undirected graph with node set $V = \{1, 2, \ldots, n\}$.

To every edge $\{u, v\} \in E$ with $u < v$ we assign a variable $x_{u,v}$.

The Tutte matrix of G is the matrix $T_G = (T_{u,v})_{1 \le u,v \le n}$ with

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u, v\} \in E \text{ and } u < v \\ -x_{v,u} & \text{falls } \{u, v\} \in E \text{ and } u > v \text{ .} \\ 0 & \text{sonst} \end{cases}$$

We are interested in the determinant of T_G that can be computed with the Leibniz formula, see Slide 56.

Note: $\det(T_G)$ is a polynomial with $|E| \le n^2$ variables having degree at most n.

Application: perfect matchings

Example: Consider the following graph G:
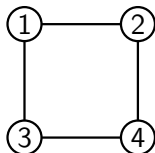
Its Tutte matrix is

$$T_G = \begin{pmatrix} 0 & x_{1,2} & x_{1,3} & 0 \\ -x_{1,2} & 0 & 0 & x_{2,4} \\ -x_{1,3} & 0 & 0 & x_{3,4} \\ 0 & -x_{2,4} & -x_{3,4} & 0 \end{pmatrix}$$

Application: perfect matchings

Tutte's theorem

G has a perfect matching if and only if $\det(T_G)$ is not the zero polynomial.

We postpone the proof of Tutte's theorem.

Caution: we cannot not use the Leibniz formula for computing the polynomial $\det(T_G)$ efficiently (in polynomial time), since this involves a sum over all $\sigma \in S_n$ ($n!$ many summands).

Instead, we will test probabilistically whether $\det(T_G)$ is the zero polynomial.

Application: perfect matchings

Example: Let us compute the determinant of the Tutte matrix from Slide 81:

$$
\begin{aligned}
\det(T_G) &= \det \begin{pmatrix} 0 & x_{1,2} & x_{1,3} & 0 \\ -x_{1,2} & 0 & 0 & x_{2,4} \\ -x_{1,3} & 0 & 0 & x_{3,4} \\ 0 & -x_{2,4} & -x_{3,4} & 0 \end{pmatrix} \\[2mm]
&= x_{1,2} \det \begin{pmatrix} x_{1,2} & x_{1,3} & 0 \\ 0 & 0 & x_{3,4} \\ -x_{2,4} & -x_{3,4} & 0 \end{pmatrix} - x_{1,3} \det \begin{pmatrix} x_{1,2} & x_{1,3} & 0 \\ 0 & 0 & x_{2,4} \\ -x_{2,4} & -x_{3,4} & 0 \end{pmatrix} \\[2mm]
&= -x_{1,2} x_{3,4} \det \begin{pmatrix} x_{1,2} & x_{1,3} \\ -x_{2,4} & -x_{3,4} \end{pmatrix} + x_{1,3} x_{2,4} \det \begin{pmatrix} x_{1,2} & x_{1,3} \\ -x_{2,4} & -x_{3,4} \end{pmatrix} \\[2mm]
&= 2(x_{1,2}^2 x_{3,4}^2 - x_{1,2} x_{1,3} x_{2,4} x_{3,4})
\end{aligned}
$$

Application: perfect matchings

Theorem 14

There is a randomized NC-algorithm (that needs time $(\log(n))^{\mathcal{O}(1)}$ with $n^{\mathcal{O}(1)}$ processors) with the following properties:

- The input is a finite undirected graph G.

- If G has no perfect matching, then the algorithm rejects G with probability 1.

- If G has a perfect matching, then the algorithm accepts G with probability $\geq 1/2$.

Application: perfect matchings

Proof of Theorem 14:

The algorithm works as follows on a graph $G = (V, E)$ with $|V| = n$:

- Construct the Tutte matrix T_G in time $\mathcal{O}(1)$ using n^2 processors.

- Choose randomly (uniform distribution) a tuple $(a_1, a_2, \ldots, a_m) \in \{1, \ldots, 2n\}^m$, where $m = |E|$ is the number of variables in T_G.

- Compute $D = \det(T_G)(a_1, \ldots, a_m) = \det(T_G(a_1, \ldots, a_m))$ in NC using Csansky's algorithm.

- If $D \neq 0$ then accept, otherwise reject.

Application: perfect matchings

If G has no perfect matching then by Tutte's theorem we have $\det(T_G) = 0$. Hence, the algorithm will reject with probability 1.

If G has a perfect matching, then by Tutte's theorem $\det(T_G)$ is not the zero polynomial.

The theorem of Zippel and Schwartz (with $S = \{1, \ldots, 2n\}$) implies

$$\text{Prob}[\text{algorithm rejects } G] \leq \frac{1}{2n} \deg(\det(T_G)) \leq \frac{n}{2n} = \frac{1}{2}$$

In other words:

$$\text{Prob}[\text{algorithm accepts } G] \geq \frac{1}{2}$$

$\square$

Application: perfect matchings

Remark (probability amplification): The probability $\frac{1}{2}$ in Theorem 14 can be increased to $1 - \frac{1}{2^k}$ by repeating the algorithm k times (with independent random choices in each repetition):

- If the algorithm accepts G in one of the k repetitions, then the overall algorithm accepts G.
- If the algorithm rejects G in all k repetitions, then the overall algorithm rejects G.

If G has no perfect matching, then the algorithm will reject G in each of the k repetitions with probability 1.

Hence, the overall algorithm will reject G with probability 1.

If G has a perfect matching, then in each of the k repetitions the algorithm will reject G with probability $\leq 1/2$.

Hence, the probability that the overall algorithm rejects G is $\leq \frac{1}{2^k}$.

Therefore, the overall algorithm accepts G with probability $\geq 1 - \frac{1}{2^k}$.

Application: perfect matchings

It remains to prove Tutte's theorem:

$G = (\{1, \ldots, n\}, E)$ has a perfect matching $\Leftrightarrow \det(T_G) \neq 0$.

Recall that $T_G = (T_{u,v})_{1 \leq u,v \leq n}$ with

$$T_{u,v} = \begin{cases} x_{u,v} & \text{falls } \{u, v\} \in E \text{ and } u < v \\ -x_{v,u} & \text{falls } \{u, v\} \in E \text{ and } u > v \ . \\ 0 & \text{sonst} \end{cases}$$

Additional background on permutations (recall slides 54–55):

Let $\sigma : [1, n] \to [1, n]$ be a permutation. We can write σ uniquely as a product of disjoint cycles (the order of the cycles is arbitrary).

Sei $E_n = \{\sigma \in S_n \mid \sigma \text{ only contains cycles of even length}\}$.

Application: perfect matchings

Example: Consider the permutation $\sigma : [1,8] \to [1,8]$ with

a	1	2	3	4	5	6	7	8
$\sigma(a)$	5	4	7	2	3	8	1	6

Then σ looks as follows
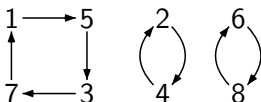
We also write $\sigma = (1,5,3,7)(2,4)(6,8) = (6,8)(2,4)(1,5,3,7) = \cdots$.

We have $\sigma \in E_n$, since all cycles have even length.

Note: every transposition τ consists of a cycle of length two and $n-2$ cylces of length one (fix points). Thus, $\tau \in E_n$ if and only if $n = 2$.

Application: perfect matchings

Recall Leibniz' formula: $\det(T_G) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)}$.

Lemma 15

$$\det(T_G) = \sum_{\sigma \in E_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)}$$

Proof: We show

$$\sum_{\sigma \in S_n \setminus E_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = 0.$$

Application: perfect matchings

If $\sigma \in S_n$ contains a fix point (i.e., $\sigma(j) = j$ for some $j \in [1, n]$ — such a permutation belongs to $S_n \smallsetminus E_n$) then $\prod_{i=1}^{n} T_{i,\sigma(i)} = 0$ (since $T_{j,\sigma(j)} = 0$).

Therefore we only have to consider permutations from $S_n \smallsetminus E_n$ that do not have a fix point.

Let $U_n = \{\sigma \in S_n \smallsetminus E_n \mid \forall i \in [1, n] : \sigma(i) \neq i\}$.

It remains to show: $\displaystyle\sum_{\sigma \in U_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = 0$.

Let $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k \in U_n$, where the σ_i are pairwise disjoint cycles.

W.l.o.g. $\sigma_1 = (a_0, a_1, \ldots, a_{\ell-1})$ has an odd length $\ell \geq 3$.

Application: perfect matchings

Let $\tau = \sigma_1^{-1}\sigma_2\cdots\sigma_k = (a_{\ell-1}, \ldots, a_1, a_0)\sigma_2\cdots\sigma_k \in U_n$.

For all $0 \le i \le \ell - 1$ (where $i + 1$ should be understood as $i + 1 \bmod \ell$) we have:

$$T_{a_i,\sigma(a_i)} = T_{a_i,a_{i+1}} = -T_{a_{i+1},a_i} = -T_{a_{i+1},\tau(a_{i+1})}$$

We therefore obtain

$$\prod_{i=0}^{\ell-1} T_{a_i,\sigma(a_i)} = (-1)^\ell \prod_{i=0}^{\ell-1} T_{a_{i+1},\tau(a_{i+1})} = -\prod_{i=0}^{\ell-1} T_{a_i,\tau(a_i)}$$

This implies

$$\prod_{i=1}^{n} T_{i,\sigma(i)} = -\prod_{i=1}^{n} T_{i,\tau(i)}.$$

Application: perfect matchings

We claim that $\text{sign}(\sigma) = \text{sign}(\tau)$, from which we get

$$\text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = -\text{sign}(\tau) \prod_{i=1}^{n} T_{i,\tau(i)}.$$

We have $\sigma\tau = \sigma_1\sigma_2\cdots\sigma_k\sigma_1^{-1}\sigma_2\cdots\sigma_k = (\sigma_2\cdots\sigma_k)^2$.

Since every permutation of the form ρ^2 is a product of an even number of transpositions, we get $1 = \text{sign}(\sigma\tau) = \text{sign}(\sigma) \cdot \text{sign}(\tau)$.

The latter implies $\text{sign}(\sigma) = \text{sign}(\tau)$.

We now can extend the pairing between σ and τ to all permutations from U_n, which finally shows

$$\sum_{\sigma \in U_n} \text{sign}(\sigma) \prod_{i=1}^{n} T_{i,\sigma(i)} = 0.$$

Application: perfect matchings

For this we define on the set U_n an involution $g : U_n \to U_n$
($g(\rho) \neq \rho$ and $g^2(\rho) = \rho$ for all $\rho \in U_n$):

Let $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k \in U_n$, where the σ_i are pairwise disjoint cycles.

Take the unique cycle σ_j of odd length, which contains the minimum of all elements from $[1, n]$ appearing on a cycle σ_i of odd length.

Define $g(\sigma) = \sigma_1 \cdots \sigma_{j-1} \sigma_j^{-1} \sigma_{j+1} \cdots \sigma_k$.

We have $g(\sigma) \neq \sigma$ (since the cycle σ_j has length ≥ 3) and $g^2(\sigma) = \sigma$.

Moreover, the argument from the previous slide yields

$$\forall \sigma \in U_n : \operatorname{sign}(\sigma) \prod_{i=1}^n T_{i,\sigma(i)} = -\operatorname{sign}(g(\sigma)) \prod_{i=1}^n T_{i,g(\sigma)(i)}.$$

This prove the lemma. $\qquad\square$

Application: perfect matchings

Example: For $n = 5$ the mapping $g : U_5 \to U_5$ looks as follows:

$$
\begin{aligned}
(1,2,3)(4,5) &\leftrightarrow (1,3,2)(4,5) \\
(1,2,4)(3,5) &\leftrightarrow (1,4,2)(3,5) \\
(1,2,5)(3,4) &\leftrightarrow (1,5,2)(3,4) \\
(1,3,4)(2,5) &\leftrightarrow (1,4,3)(2,5) \\
(1,3,5)(2,4) &\leftrightarrow (1,5,3)(2,4) \\
(1,4,5)(2,3) &\leftrightarrow (1,5,4)(2,3) \\
(2,3,4)(1,5) &\leftrightarrow (2,4,3)(1,5) \\
(2,3,5)(1,4) &\leftrightarrow (2,5,3)(1,4) \\
(2,4,5)(1,3) &\leftrightarrow (2,5,4)(1,3) \\
(3,4,5)(1,2) &\leftrightarrow (3,5,4)(1,2)
\end{aligned}
$$

Application: perfect matchings

We can conclude the proof of Tutte's theorem:

(1) Assume that $\det(T_G)$ is not the zero polynomial.

By Lemma 15 there exists $\sigma \in E_n$ such that $\prod\limits_{i=1}^{n} T_{i,\sigma(i)} \neq 0$.

Hence, we have $\{i, \sigma(i)\} \in E$ for all $1 \leq i \leq n$.

We obtain a perfect matching for G by selecting every second edge on each cycle in σ.

Example:
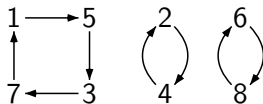
A permutation $\sigma \in E_8$

Application: perfect matchings

We can conclude the proof of Tutte's theorem:

(1) Assume that $\det(T_G)$ is not the zero polynomial.

By Lemma 15 there exists $\sigma \in E_n$ such that $\prod\limits_{i=1}^{n} T_{i,\sigma(i)} \neq 0$.

Hence, we have $\{i, \sigma(i)\} \in E$ for all $1 \leq i \leq n$.

We obtain a perfect matching for G by selecting every second edge on each cycle in σ.

Example:
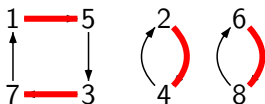
A permutation $\sigma \in E_8$
A perfecte matching obtained by selecting
every second edge on each cycle

Application: perfect matchings

(2) Assume that G has a perfect matching $M \subseteq E$.

We assign values to the variables $x_{u,v}$ ($\{u, v\} \in E$, $u < v$):

$$x_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in M \\ 0 & \text{otherwise} \end{cases}$$

Substituting these values into the Tutte matrix T_G yields a matrix that contains in each row as well as in each column exactly one non-zero entry (either 1 or -1); a so-called permutation matrix.

Leibniz' formula implies that this matrix has a non-zero determinant.

Therefore, $\det(T_G)$ cannot be the zero polynomial.

This concludes the proof of Tutte's theorem. □

Pattern matching with fingerprints

Let Σ be a finite alphabet.

Let $T = a_1 a_2 \cdots a_n$ be a text and $P = b_1 b_2 \cdots b_m$ be a pattern
$(a_i, b_j \in \Sigma,\ m \le n)$.

Goal: find all occurrences of P in T, i.e., all positions $1 \le i \le n - m + 1$
such that

$$T[i, i + m - 1] := a_i a_{i+1} \cdots a_{i+m-1} = P.$$

The algorithm of Knuth, Morris and Pratt solves this problem in
(sequential) running time $\mathcal{O}(m + n)$.

Here, we want to develop a randomized parallel algorithm.

For the following we assume that $\Sigma = \{0, 1\}$.

Pattern matching with fingerprints

Example: All occurrences of the pattern 0110 in the following text:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	1	0	1	1	0	0	1	0	1	0	0	1	1	0	1

Pattern matching with fingerprints

Define $f : \Sigma \to \mathbb{Z}^{2 \times 2}$ by

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

We extend f to a homomorphism $f : \Sigma^* \to \mathbb{Z}^{2 \times 2}$:

$$f(a_1 a_2 \cdots a_k) = f(a_1) f(a_2) \cdots f(a_k) \text{ for all } a_1, \ldots, a_k \in \Sigma$$

In particular, $f(\varepsilon) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Lemma 16

The following hold:

- The homomorphism f ist injective (if $u \neq v$ then $f(u) \neq f(v)$).

- If $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ and $|w| = \ell$ then $a_i \leq 2^\ell$ for all $1 \leq i \leq 4$.

Pattern matching with fingerprints

Proof: We have

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix} \tag{4}$$

and

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} a_1 + a_3 & a_2 + a_4 \\ a_3 & a_4 \end{pmatrix} \tag{5}$$

(A) If $w \in \{0,1\}^*$ and $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ then $a_1, a_4 > 0$, $a_2, a_3 \geq 0$.

Proof of (A): Induction over $|w|$.

Pattern matching with fingerprints

(B) If $w \neq \varepsilon$ then $f(w) \neq f(\varepsilon) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Proof of (B):

If $w = 0u$ and $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ then $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix}$.

If $f(w) = f(\varepsilon)$ then we obtain $a_1 = a_2 = a_3 = 0$, which contradicts $a_1 > 0$ (see (A)).

For $w = 1u$ we can argue analogously.

Pattern matching with fingerprints

(C) $f(0u) \neq f(1v)$ for all $u, v \in \{0, 1\}^*$.

Proof von (C): Let $f(u) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ and $f(v) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$.

If $f(0u) = f(1v)$ then we obtain:

$$\begin{pmatrix} a_1 & a_2 \\ a_1 + a_3 & a_2 + a_4 \end{pmatrix} = \begin{pmatrix} b_1 + b_3 & b_2 + b_4 \\ b_3 & b_4 \end{pmatrix}$$

Therefore:

$$a_1 = b_1 + b_3, \qquad a_1 + a_3 = b_3$$
$$a_2 = b_2 + b_4, \qquad a_2 + a_4 = b_4$$
$$b_1 + a_3 = 0, \text{i.e. } a_3 = b_1 = 0, \text{ a contradiction with } b_1 > 0$$
$$a_4 + b_2 = 0, \text{i.e. } a_4 = b_2 = 0, \text{ a contradiction with } a_4 > 0$$

Pattern matching with fingerprints

Now we are in the position to finish the proof of Lemma 16.

Assume that $u, v \in \{0, 1\}^*$, $u \neq v$ and $f(u) = f(v)$.

We will deduce a contradiction.

The matrices $f(0)$ and $f(1)$ are invertible ($\det(f(0)) = \det(f(1)) = 1$). Therefore every matrix $f(x)$ is invertible.

Case 1: $u = vw$ with $w \neq \varepsilon$.

We get $f(v)f(w) = f(u) = f(v)$, i.e., $f(w) = \mathsf{Id}_2 = f(\varepsilon)$.

This contradicts (B).

Case 2: $v = uw$ with $w \neq \varepsilon$: analogously

Case 3: There exist u', v', w with $u = w0u'$ and $v = w1v'$.

We get $f(w)f(0u') = f(w)f(1v')$, i.e., $f(0u') = f(1v')$.

This contradicts (C).

Pattern matching with fingerprints

Case 4: There exist $u'v', w$ with $u = w1u'$ and $v = w0v'$:
analogously

This proves the first statement of the lemma.

The second statement can be shown by induction on $|w|$:

If $w = \varepsilon$ then $f(w) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $a_i \leq 1 = 2^0$ for $1 \leq i \leq 4$.

Assume that $w = au$ for $a \in \{0, 1\}$ and $f(u) = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$ with $b_i \leq 2^{|u|}$ for
$1 \leq i \leq 4$.

If $f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$ then (4) and (5) yield $a_i \leq 2 \cdot 2^{|u|} = 2^{|u|+1} = 2^{|w|}$ for
$1 \leq i \leq 4$. $\qquad \square$

Pattern matching with fingerprints

First idea for pattern matching: compare $f(P)$ with $f(T[i, i + m - 1])$ for all $1 \leq i \leq n - m + 1$.

Problem: $f(P)$ can have entries of size F_{m+1} ($(m + 1)$-th Fibonacci number), which need $\Omega(m)$ bits. Therefore, the comparison of $f(P)$ and $f(T[i, i + m - 1])$ needs time $\Omega(m)$, and we gain nothing compared to a direct comparison of P and $T[i, i + m - 1]$.

Solution: Compute modulo a sufficiently large prime number.

For a word $w \in \{0, 1\}^*$ and a prime number p let

$$f_p(w) = \begin{pmatrix} a_1 \bmod p & a_2 \bmod p \\ a_3 \bmod p & a_4 \bmod p \end{pmatrix}, \text{wobei } f(w) = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$$

The matrix $f_p(w)$ is called the fingerprint of the string w (with respect to the prime number p).

Pattern matching with fingerprints

Example: Compute the fingerprint of the string 01101 with respect to the prime 3:

We have

$$
\begin{aligned}
f(01101) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 3 & 2 \\ 4 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 3 & 5 \\ 4 & 7 \end{pmatrix}
\end{aligned}
$$

Hence, we get $f_3(01101) = \begin{pmatrix} 0 & 2 \\ 1 & 1 \end{pmatrix}$.

Pattern matching with fingerprints

Let p be a prime number and let $X, Y \in \{0,1\}^*$ two strings.

We say that p is bad for (X, Y) if

- $X \neq Y$ (and hence $f(X) \neq f(Y)$) and

- $f_p(X) = f_p(Y)$.

For $k \in \mathbb{N}$ let $\mathrm{Primes}(k)$ the set of all prime numbers p with $2 \leq p \leq k$ and $\pi(k) = |\mathrm{Primes}(k)|$.

The following facts hold:

- $\frac{k}{\ln(k)} \leq \pi(k) \leq 1.2551 \cdot \frac{k}{\ln(k)}$ (ln x is the logarithm of x to the base $e = 2.71828\ldots$)

- If $k \geq 29$ and $u \leq 2^k$, then the number of different prime factors of u is at most $\pi(k)$.

Pattern matching with fingerprints

Lemma 17

Let $X_1, Y_1, \ldots, X_t, Y_t$ be strings of length m with m sufficiently large and let $M \geq 1$.

For a randomly chosen prime $p \in \text{Primes}(M)$ (uniform distribution) we have:

$$\text{Prob}[\, \exists 1 \leq i \leq t : p \text{ is bad for } (X_i, Y_i)] \leq \frac{\pi(4 \cdot m \cdot t)}{\pi(M)}$$

Proof: For $1 \leq i \leq t$ let

$$f(X_i) = \begin{pmatrix} a_{i,1} & a_{i,2} \\ a_{i,3} & a_{i,4} \end{pmatrix} \text{ and } f(Y_i) = \begin{pmatrix} b_{i,1} & b_{i,2} \\ b_{i,3} & b_{i,4} \end{pmatrix}.$$

All $a_{i,j}$ and $b_{i,j}$ belong to $[0, 2^m]$ by Lemma 16.

Pattern matching with fingerprints

We have:

$\exists 1 \le i \le t : p$ is bad for (X_i, Y_i)

$\Longleftrightarrow$

$\exists 1 \le i \le t : f(X_i) \ne f(Y_i)$ and $f_p(X_i) = f_p(Y_i)$

$\Longrightarrow$

p divides the product $\prod \{|a_{i,j} - b_{i,j}| \mid 1 \le i \le t, 1 \le j \le 4, a_{i,j} \ne b_{i,j}\}$

$\Longrightarrow$

p divides a number $u \le 2^{4 \cdot t \cdot m}$

Since the number of different prime factors of $u \le 2^{4 \cdot t \cdot m}$ is at most $\pi(4 \cdot m \cdot t)$ (if $4 \cdot m \cdot t \ge 29$) we have

$$\text{Prob}[\, \exists 1 \le i \le t : p \text{ is bad for } (X_i, Y_i)] \le \frac{\pi(4 \cdot m \cdot t)}{\pi(M)}.$$

$\square$

Pattern matching with fingerprints

Lemma 18

Let $X_1, Y_1, \ldots, X_t, Y_t$ be strings of length m with m sufficiently large.

Fix a constant $k \geq 1$ and let $M = m \cdot t^k$.

For a randomly chosen prime $p \in \text{Primes}(M)$ we have

$$\text{Prob}[\, \exists 1 \leq i \leq t : p \text{ is bad for } (X_i, Y_i)\,] \leq \mathcal{O}\left(\frac{1}{t^{k-1}}\right)$$

Pattern matching with fingerprints

Proof: We have $\pi(4 \cdot m \cdot t) \leq 1.2551 \cdot \frac{4 \cdot m \cdot t}{\ln(m \cdot t)} = 5.0204 \cdot \frac{m \cdot t}{\ln(m \cdot t)}$ and

$$\pi(M) = \pi(m \cdot t^k) \geq \frac{m \cdot t^k}{\ln(m \cdot t^k)} = \frac{m \cdot t^k}{\ln(m) + k \cdot \ln(t)}.$$

With Lemma 17 we get:

$$\text{Prob}[\,\exists 1 \leq i \leq t : p \text{ is bad for } (X_i, Y_i)\,]$$

$$\leq \quad 5.0204 \cdot \frac{m \cdot t \cdot (\ln(m) + k \ln(t))}{\ln(m \cdot t) \cdot m \cdot t^k}$$

$$\leq \quad 5.0204 \cdot k \cdot \frac{1}{t^{k-1}}$$

$$\leq \quad \mathcal{O}\left(\frac{1}{t^{k-1}}\right) \text{ (since } k \text{ is a constant).} \qquad \square$$

Pattern matching with fingerprints

Recall: $T = a_1 a_2 \cdots a_n$ (the text) and $P = b_1 b_2 \cdots b_m$ (the pattern)

Assumption: On a single processor an arithmetic operation on integers with $\mathcal{O}(\log(n))$ bits needs constant time.

For a prime number $p \leq n^{\mathcal{O}(1)}$ we can compute with n processors in time $\mathcal{O}(\log(n))$ all finger prints $f_p(T[i, i + m - 1])$ $(1 \leq i \leq n - m + 1)$:

For this we use the prefix sum algorithm to compute all products

$$R_i = f_p(T[1, i]) = f_p(a_1) f_p(a_2) \cdots f_p(a_i)$$

$(1 \leq i \leq n)$ in time $\mathcal{O}(\log(n))$ using n processors.

Since we compute module a prime number of size $n^{\mathcal{O}(1)}$, all numbers that occur during the computation have $\mathcal{O}(\log(n))$ bits.

Pattern matching with fingerprints

Let $R_i = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $0 \le a, b, c, d \le p - 1$.

Then the inverse matrix R_i^{-1} (in the field $\mathbb{F}_p$) can be computed using the formula

$$R_i^{-1} = \frac{1}{\det(R_i)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Since $\det(f(0)) = \det(f(1)) = 1$ and $\det(A \cdot B) = \det(A) \cdot \det(B)$ we have $\det(f(w)) = 1$ for all $w \in \{0, 1\}^*$.

Hence, $\det(f_p(w)) = 1$ for all $w \in \{0, 1\}^*$, in particular $\det(R_i) = 1$.

Pattern matching with fingerprints

Finally compute we compute in time $\mathcal{O}(1)$ using n processors the fingerprint of all $T[i, i + m - 1]$:

$$
\begin{aligned}
& f_p(T[i, i + m - 1]) \\
&= f_p(a_i) f_p(a_{i+1}) \cdots f_p(a_{i+m-1}) \\
&= f_p(a_{i-1})^{-1} \cdots f_p(a_2)^{-1} f_p(a_1)^{-1} f_p(a_1) f_p(a_2) \cdots f_p(a_{i+m-1}) \\
&= (f_p(a_1) f_p(a_2) \cdots f_p(a_{i-1}))^{-1} f_p(a_1) f_p(a_2) \cdots f_p(a_{i+m-1}) \\
&= R_{i-1}^{-1} \cdot R_{i+m-1}
\end{aligned}
$$

Pattern matching with fingerprints

Theorem 19

Fix a constant k. Using $\mathcal{O}(n)$ processors we can compute in time $\mathcal{O}(\log(n))$ an array $\text{MATCH}[1, \ldots, n]$ with the following properties:

- If $T[i, i + m - 1] = P$, then $\text{MATCH}[i] = 1$ with probability 1.

- The probability that there exists an i with $\text{MATCH}[i] = 1$ and $T[i, i + m - 1] \neq P$ is bounded by $\mathcal{O}\left(\frac{1}{n^k}\right)$.

Pattern matching with fingerprints

Proof:

1. Let $M = m \cdot n^{k+1} \leq n^{k+2}$.

2. Choose randomly a prime number $p \in \{1, \ldots, M\}$.

3. Compute $f_p(P)$ in time $\mathcal{O}(\log(m))$ using $m \leq n$ processors.

4. For $1 \leq i \leq n - m + 1$ compute in parallel all fingerprints $L_i := f_p(T[i, i + m - 1])$ using the algorithm from Slide 113.

5. For all $1 \leq i \leq n - m + 1$ set in parallel $\text{MATCH}[i] = 1$ if and only if $L_i = f_p(P)$.

By Lemma 18 (applied with $t = n - m + 1$, $X_i = T[i, i + m - 1]$ and $Y_i = P$), the probability that an entry $\text{MATCH}[i]$ is incorrectly set to 1 is bounded by $\mathcal{O}\left(\frac{1}{n^k}\right)$. $\qquad\square$

Streaming algorithms

A streaming algorithm receives a sequence $S = (a_1, a_2, \ldots, a_\ell)$
(a so called stream) of element from some universe of size n.

At time t the algorithm only has direct access to a_t and its internal
memory state.

In particular: no direct access to the previous values $a_1, a_2, \ldots, a_{t-1}$ unless
the algorithm explicitly stores these values.

Goal: Compute important statistical data without storing all values from
the stream.

Often, a good approximation of such statistical data suffices.

Streaming algorithms

Let $S = (a_1, a_2, \ldots, a_\ell)$ be a stream of elements $a_i \in \{1, \ldots, n\}$.

For $1 \le i \le n$ let $m_i(S) = |\{t \mid 1 \le t \le \ell, a_t = i\}|$ be the number of occurrences of i in the stream S.

For $k \ge 0$ we define the F_k-norm (or the k-th moment) of S as

$$F_k(S) = \sum_{i=1}^{n} m_i(S)^k$$

Note that:

- $F_0(S)$ is the number of different elements that appear in S
  ($0^0 = 0$ and $m^0 = 1$ for $m \ge 1$).

- $F_1(S) = \ell$ is the length of the stream (not really interesting).

Streaming algorithms

Goal: Space efficient algorithm for computing $F_0(S)$.

A naive solution can do this by storing n bits:

- For each $1 \leq i \leq n$ we store a bit s_i which is initially zero and set to one, once the data value i appears in the stream.
- At the end we output $\sum_{i=1}^{n} s_i$.

Can we do better, i.e., compute $F_0(S)$ with $o(n)$ space?

Depends on what we want:

- The exact value $F_0(S)$ cannot be computed in space $o(n)$ — even if we allow randomized streaming algorithms with an error probability of $< 1/2$. ↪ randomized communication complexity
- But: a good approximation of $F_0(S)$ can be computed in space $\mathcal{O}(\log n)$ with a randomized streaming algorithm with a small error probability. ↪ algorithm of Alon, Matias and Szegedy (AMS)

Pairwise independent hash functions

Main tool of the AMS-algorithm: pairwise independent hashing.

Family of pairwise independent hash functions

Let A and B be finite sets, $|A| \geq 2$.

A set $\mathcal{H} \subseteq \{h \mid h : A \to B\}$ is a family of pairwise independent hash functions if the following holds for all $a_1, a_2 \in A$ with $a_1 \neq a_2$ and all $b_1, b_2 \in B$:

If we choose a mapping $h \in \mathcal{H}$ uniformly at random
(every $h \in \mathcal{H}$ is chosen with probability $1/|\mathcal{H}|$) then

$$\text{Prob}[\, h(a_1) = b_1 \land h(a_2) = b_2 \,] = 1/|B|^2.$$

Pairwise independent hash functions

Let $\mathcal{H} \subseteq \{h \mid h : A \to B\}$ be a family of pairwise independent hash functions and choose a mapping $h \in \mathcal{H}$ uniformly at random. We have:

- $\text{Prob}[\, h(a) = b\,] = 1/|B|$ for all fixed $a \in A$ and $b \in B$.

  Take an $a' \in A$ with $a \neq a'$ (recall that $|A| \geq 2$). Then we have

$$
\begin{aligned}
\text{Prob}[\, h(a) = b\,] &= \text{Prob}\left[\bigvee_{c \in B} (h(a) = b \wedge h(a') = c)\right] \\
&= \sum_{c \in B} \text{Prob}[h(a) = b \wedge h(a') = c] \\
&= \frac{|B|}{|B|^2} \\
&= 1/|B|
\end{aligned}
$$

  Hence, for a fixed $a \in A$ the value $h(a)$ is uniformly distributed over the set B.

Pairwise independent hash functions

- We get for $a_1 \neq a_2$ the pairwise independence property:

$$\text{Prob}[h(a_1) = b_1 \wedge h(a_2) = b_2] = \text{Prob}[h(a_1) = b_1] \cdot \text{Prob}[h(a_2) = b_2].$$

- For all $a_1 \neq a_2$ we have $\text{Prob}[h(a_1) = h(a_2)] = 1/|B|$:

$$
\begin{aligned}
\text{Prob}[h(a_1) = h(a_2)] &= \text{Prob}\left[\bigvee_{b \in B}(h(a_1) = b \wedge h(a_2) = b)\right] \\
&= \sum_{b \in B}\text{Prob}\left[h(a_1) = b \wedge h(a_2) = b\right] \\
&= 1/|B|
\end{aligned}
$$

One says that $\mathcal{H}$ is a universal family of hash functions.

Pairwise independent hash functions

We now construct a family of pairwise independent hash functions on the finite field $\mathbb{F}_p$ ($\{0, 1, \ldots, p-1\}$ with addition and multiplication modulo p) for a prime number p.

For $x, y \in \mathbb{F}_p$ define the mapping $h_{x,y} : \mathbb{F}_p \to \mathbb{F}_p$ by

$$h_{x,y}(a) = (ax + y) \bmod p \text{ for } a \in \mathbb{F}_p.$$

Let $\mathcal{H}_p = \{h_{x,y} \mid x, y \in \mathbb{F}_p\}$.

Theorem 20
$\mathcal{H}_p$ is a family of pairwise independent hash functions on $\mathbb{F}_p$.

Pairwise independent hash functions

Proof: First notice that $|\mathcal{H}_p| = p^2$:

If $ax_1 + y_1 = ax_2 + y_2 \bmod p$ for all $a \in \mathbb{F}_p$ then we must have $x_1 = x_2$ (otherwise $a = (y_2 - y_1)/(x_1 - x_2)$) and hence $y_1 = y_2$.

Hence, $(x_1, y_1) \neq (x_2, y_2)$ implies $h_{x_1,y_1} \neq h_{x_2,y_2}$

Next, let us fix $a_1, a_2, b_1, b_2 \in \mathbb{F}_p$ with $a_1 \neq a_2$.

Then the system

$$a_1 x + y = b_1 \bmod p$$
$$a_2 x + y = b_2 \bmod p$$

has a unique solution $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$.

Pairwise independent hash functions

To see this, note that the system is equivalent to

$$\begin{pmatrix} a_1 & 1 \\ a_2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \text{ (in } \mathbb{F}_p)$$

and that

$$\det \begin{pmatrix} a_1 & 1 \\ a_2 & 1 \end{pmatrix} = a_1 - a_2 \neq 0$$

Hence, the unique solution is

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_1 & 1 \\ a_2 & 1 \end{pmatrix}^{-1} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

Therefore, there is a unique $h_{x,y} \in \mathcal{H}_p$ such that

$$h_{x,y}(a_1) = b_1 \wedge h_{x,y}(a_2) = b_2.$$

Pairwise independent hash functions

Thus: If $x, y \in \mathbb{F}_p$ are chosen uniformly at random, then

$$\text{Prob}[\, h_{x,y}(a_1) = b_1 \wedge h_{x,y}(a_2) = b_2 \,] = 1/p^2. \qquad \square$$

Remark: The above proof works for every finite field instead of $\mathbb{F}_p$.

For every prime number p and every $m \geq 1$ there exists a unique finite field $\mathbb{F}_{p^m}$ with p^m elements (this is not the ring of integers modulo p^m unless $m = 1$).

For the AMS algorithm it is convenient to take a finite field $\mathbb{F}_{2^m}$.

Mathematical background: (pairwise) independence

Let X be a random variable, which takes values from a finite set $A \subseteq \mathbb{R}$.

For every $a \in A$, $\mathrm{Prob}[X = a]$ is the probability that X takes the value a.

Independent and pairwise independent random variables

Let $X_1, X_2, \ldots, X_n$ be random variables.

$X_1, X_2, \ldots, X_n$ are independent if:

$$\mathrm{Prob}\left[\bigwedge_{i=1}^{n} X_i = a_i \right] = \prod_{i=1}^{n} \mathrm{Prob}[X_i = a_i].$$

$X_1, X_2, \ldots, X_n$ are pairwise independent if for all $1 \le i, j \le n$ with $i \ne j$:

$$\mathrm{Prob}[X_i = a \wedge X_j = b] = \mathrm{Prob}[X_i = a] \cdot \mathrm{Prob}[X_j = b].$$

Mathematical background: expected value, variance

Expected value and variance

The expected value of X is $E[X] = \sum_{a \in A} \text{Prob}[X = a] \cdot a$.

The variance of X is $\text{Var}[X] = E[(X - E[X])^2]$.

Linearity of expectation

For random variables X and Y and $a \in \mathbb{R}$ we have:

$$E[X + Y] = E[X] + E[Y]$$
$$E[aX] = a \cdot E[X]$$

In particular we get

$$\begin{aligned}
\text{Var}[X] &= E[(X - E[X])^2] \\
&= E[X^2 - 2E[X]X + E[X]^2] \\
&= E[X^2] - 2E[X]^2 + E[X]^2 = E[X^2] - E[X]^2.
\end{aligned}$$

Mathematical background: expected value, variance

In general, $E[X \cdot Y] = E[X] \cdot E[Y]$ does not hold.

Lemma 21

If $X_1, X_2, \ldots, X_n$ are independent random variables then

$$E\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} E[X_i].$$

Mathematical background: expected value, variance

Let $X_1, X_2, \ldots, X_n$ be random variables.

Linearity of expectation implies $E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$.

In general, $\mathrm{Var}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathrm{Var}[X_i]$ is wrong.

Lemma 22

If $X_1, X_2, \ldots, X_n$ are pairwise independent random variables then

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i].$$

Mathematical background: expected value, variance

If X is a random variable and $f : \mathbb{R} \to \mathbb{R}$ is a function, then one can define the random variable $f(X)$:

$$\text{Prob}[f(X) = a] = \text{Prob}[X \in f^{-1}(a)] = \sum_{b \in f^{-1}(a)} \text{Prob}[X = b]$$

If $X_1, \ldots, X_n$ are (pairwise) independent, then also $f(X_1), \ldots, f(X_n)$ are (pairwise) independent.

Jensen's inequality (see Algorithms I) can be stated as

$$E[f(X)] \geq f(E[X])$$

for f a convex function.

Mathematical background: Markov's inequality

Markov's inequality

Let X be a non-negative random variable. For any real number $t > 0$, we have

$$\text{Prob}[X \geq t] \leq \frac{E[X]}{t}.$$

Proof: Fix $t > 0$ and define the random variable I by

$$I = \begin{cases} 1 & \text{if } X \geq t \\ 0 & \text{if } X < t \end{cases}$$

Since $X \geq 0$ we have $I \leq X/t$.

Hence, with linearity of expectation, we get

$$\text{Prob}[X \geq t] = E[I] \leq E[X/t] = \frac{E[X]}{t}. \qquad \square$$

Mathematical background: Chebyshev's inequality

Chebyshev's inequality

Let X be a random variable. For any real number $t > 0$, we have

$$\text{Prob}[|X - E[X]| \geq t] \leq \frac{\text{Var}[X]}{t^2}.$$

Proof: Since $t > 0$ we have

$$\text{Prob}[|X - E[X]| \geq t] = \text{Prob}[(X - E[X])^2 \geq t^2].$$

Since $(X - E[X])^2$ is a non-negative random variable, we can use Markov's inequality to get

$$\text{Prob}[(X - E[X])^2 \geq t^2] \leq \frac{E[(X - E[X])^2]}{t^2} = \frac{\text{Var}[X]}{t^2}. \qquad \square$$

Mathematical background: Chernoff bound

A Bernoulli random variable X is a random variable that only takes the two values 0 and 1.

Note: $E[X] = \text{Prob}[X = 1]$

Chernoff bound

Let $X_1, X_2, \ldots, X_k$ be independent Bernoulli random variables with the same distribution: $\text{Prob}[X_i = 1] = p$ for all $1 \le i \le k$.

For every $0 < \delta < 1$ we have:

$$\text{Prob}\left[\sum_{i=1}^{k} X_i \ge (1 + \delta)pk\right] \le e^{-\delta^2 pk/3}$$

$$\text{Prob}\left[\sum_{i=1}^{k} X_i \le (1 - \delta)pk\right] \le e^{-\delta^2 pk/2} \le e^{-\delta^2 pk/3}$$

Mathematical background: Chernoff bound

Proof: We only prove the first inequality.

Let $t > 0$ be arbitrary. With $\mu := pk = E[\sum_{i=1}^{k} X_i]$ we get (with $\exp(x) = e^x$)

$$
\begin{aligned}
\text{Prob}\left[\sum_{i=1}^{k} X_i \geq (1+\delta)\mu\right] &= \text{Prob}\left[\exp\left(t\sum_{i=1}^{k} X_i\right) \geq \exp(t(1+\delta)\mu)\right] \\[2mm]
&\leq \frac{E\left[\exp\left(\sum_{i=1}^{k} tX_i\right)\right]}{\exp(t(1+\delta)\mu)} \quad \text{(Markov's inequality)} \\[2mm]
&= \frac{E\left[\prod_{i=1}^{k} e^{tX_i}\right]}{e^{t(1+\delta)\mu}} \\[2mm]
&= \frac{\prod_{i=1}^{k} E\left[e^{tX_i}\right]}{e^{t(1+\delta)\mu}} \quad (X_1, \ldots, X_k \text{ independent}) \\[2mm]
&= \frac{\prod_{i=1}^{k}((1-p) + pe^t)}{e^{t(1+\delta)\mu}}
\end{aligned}
$$

Mathematical background: Chernoff bound

$$= \frac{\prod_{i=1}^{k}(1 + p(e^t - 1))}{e^{t(1+\delta)\mu}}$$

$$\leq \frac{\prod_{i=1}^{k} e^{p(e^t-1)}}{e^{t(1+\delta)\mu}} \quad (1 + y \leq e^y \text{ for all } y)$$

$$\leq \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}}$$

Setting $t = \ln(1 + \delta) > 0$ yields

$$\frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}} = \frac{e^{\delta\mu}}{(1+\delta)^{(1+\delta)\mu}} = \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$$

Finally, one can show the following for $0 < \delta < 1$:

$$\frac{e^\delta}{(1+\delta)^{(1+\delta)}} \leq e^{-\delta^2/3}.$$

The AMS algorithm

Fix the universe $\{0,1\}^m$ (bit strings of length m) of size $n = 2^m$.

We identify elements of $\{0,1\}^m$ with elements of the field $\mathbb{F}_{2^m}$.

For $u \in \{0,1\}^m$ define $\rho(u) = \max\{k \mid u = 0^k v \text{ for some } v \in \{0,1\}^{m-k}\}$.

Note that $\rho(u) \leq m = \log_2 n$.

The AMS algorithm

- randomly choose $h = h_{x,y} \in \mathcal{H}_n$
- initialize $z := 0$
- for every new data value $u \in \{0,1\}^m$ in the stream set
  $z := \max\{z, \rho(h(u))\}$
- return $2^{z+1/2}$

Each of the numbers x, y, z fits into $m = \log_2 n$ bits.

The AMS algorithm therefore needs space $\mathcal{O}(\log n)$.

Analysis of the AMS algorithm

Let $S = (u_1, u_2, \ldots, u_\ell)$ be the input stream with $u_1, \ldots, u_\ell \in \{0,1\}^m$.

Let $A = \{u_1, u_2, \ldots, u_\ell\} \subseteq \{0,1\}^m$ be the corresponding set.

The goal is to approximate the size of A.

In the following h denotes the randomly chosen hash function and z_f the final value of z computed by the AMS algorithm.

Note that $z_f = \max\{\rho(h(u)) \mid u \in A\}$.

For $0 \le r \le m$ and $u \in \{0,1\}^m$ we define the random variable $X_{r,u}$ by

$$X_{r,u} = \begin{cases} 1 & \text{if } \rho(h(u)) \ge r \\ 0 & \text{if } \rho(h(u)) \le r - 1 \end{cases}$$

Let $Y_r = \sum_{u \in A} X_{r,u}$.

Note that $Y_r > 0$ if and only if $z_f \ge r$.

Analysis of the AMS algorithm

Since for every $u \in \{0,1\}^m$, $h(u)$ is uniformly distributed over $\{0,1\}^m$ (i.e., $\text{Prob}[h(u) = v] = 1/2^m$ for every $v \in \{0,1\}^m$) we have

$$
\begin{aligned}
E[X_{r,u}] &= \text{Prob}[\rho(h(u)) \geq r] \\
&= \text{Prob}[\text{a randomly chosen } v \in \{0,1\}^m \text{ starts with } 0^r] \\
&= \frac{2^{m-r}}{2^m} = \frac{1}{2^r}.
\end{aligned}
$$

Linearity of expectation yields

$$
E[Y_r] = \sum_{u \in A} E[X_{r,u}] = \frac{|A|}{2^r}.
$$

Claim: For every $0 \leq r \leq m$ the random variables $X_{r,u}$ ($u \in \{0,1\}^m$) are pairwise independent.

Analysis of the AMS algorithm

Proof of the claim: Let $u, v \in \{0,1\}^m$ with $u \neq v$ and $a, b \in \{0,1\}$.

With $L_1 = \{w \in \{0,1\}^m \mid \rho(w) \geq r\}$ and $L_0 = \{0,1\}^m \smallsetminus L_1$ we get

$$
\begin{aligned}
\text{Prob}[X_{r,u} = a \wedge X_{r,v} = b] &= \text{Prob}[h(u) \in L_a \wedge h(v) \in L_b] \\
&= \text{Prob}\left[\bigvee_{x \in L_a} \bigvee_{y \in L_b} (h(u) = x \wedge h(v) = y) \right] \\
&= \sum_{x \in L_a} \sum_{y \in L_b} \text{Prob}[h(u) = x \wedge h(v) = y] \\
&= \sum_{x \in L_a} \sum_{y \in L_b} \text{Prob}[h(u) = x] \cdot \text{Prob}[h(v) = y] \\
&= \sum_{x \in L_a} \text{Prob}[h(u) = x] \cdot \sum_{y \in L_b} \text{Prob}[h(v) = y] \\
&= \text{Prob}[X_{r,u} = a] \cdot \text{Prob}[X_{r,v} = b]
\end{aligned}
$$

Analysis of the AMS algorithm

The claim implies

$$\text{Var}[Y_r] = \sum_{u \in A} \text{Var}[X_{r,u}] \leq \sum_{u \in A} E[X_{r,u}^2] = \sum_{u \in A} E[X_{r,u}] = \frac{|A|}{2^r}.$$

Applying Markov's inequality yields

$$\text{Prob}[Y_r > 0] = \text{Prob}[Y_r \geq 1] \leq E[Y_r] = \frac{|A|}{2^r}.$$

Chebyshev's inequality gives

$$
\begin{aligned}
\text{Prob}[Y_r = 0] &\leq \text{Prob}[|Y_r - E[Y_r]| \geq |A|/2^r] \\
&\leq \frac{\text{Var}[Y_r]}{(|A|/2^r)^2} \\
&\leq \frac{2^r}{|A|}.
\end{aligned}
$$

Analysis of the AMS algorithm

Recall that the AMS-algorithm outputs the value $2^{z_f+1/2}$.

Let $a \geq 0$ be the smallest integer such that $2^{a+1/2} \geq 3|A|$.

Let us first assume that $a \leq m$. Then Y_a is defined and we get

$$
\begin{aligned}
\text{Prob}[2^{z_f+1/2} \geq 3|A|] &= \text{Prob}[z_f \geq a] \\
&= \text{Prob}[Y_a > 0] \\
&\leq \frac{|A|}{2^a} \leq \frac{\sqrt{2}}{3}.
\end{aligned}
$$

If $a > m$ then $2^{m+1/2} < 3|A|$ and we get

$$
\text{Prob}[2^{z_f+1/2} \geq 3|A|] \leq \text{Prob}[2^{z_f+1/2} > 2^{m+1/2}] = 0
$$

since the AMS-algorithm always produces a value $z_f \leq m$.

Analysis of the AMS algorithm

Let $b \geq 0$ be the largest integer such that $2^{b+1/2} \leq |A|/3$.

We can assume that $2^{1/2} \leq |A|/3$ so that b indeed exists
(the case $|A| < 3\sqrt{2}$, i.e., $|A| \leq 4$ is not really interesting).

We must have $b + 1 \leq m$: $b \geq m$ yields $2^{m+1/2} \leq |A|/3$, i.e., $|A| > 2^m$ which cannot be the case.

Hence, Y_{b+1} exists and we get

$$
\begin{aligned}
\text{Prob}[2^{z_f+1/2} \leq |A|/3] &= \text{Prob}[z_f \leq b] \\
&= \text{Prob}[\neg(z_f \geq b+1)] \\
&= \text{Prob}[Y_{b+1} = 0] \\
&\leq \frac{2^{b+1}}{|A|} \leq \frac{\sqrt{2}}{3}.
\end{aligned}
$$

Analysis of the AMS algorithm

Note that $\frac{\sqrt{2}}{3} \approx 0.4714$.

We have

$$
\begin{aligned}
&\mathrm{Prob}[|A|/3 < 2^{z_f+1/2} < 3|A|] \\
&= 1 - \mathrm{Prob}[|A|/3 \geq 2^{z_f+1/2} \vee 2^{z_f+1/2} \geq 3|A|] \\
&= 1 - \left(\mathrm{Prob}[|A|/3 \geq 2^{z_f+1/2}] + \mathrm{Prob}[2^{z_f+1/2} \geq 3|A|] \right) \\
&\geq 1 - \frac{2\sqrt{2}}{3} \geq 0.0571
\end{aligned}
$$

We proved the following result:

Theorem 23

With probability at least 0.0571 the AMS algorithm computes a value α with $|A|/3 < \alpha < 3|A|$.

Probability amplification for the AMS algorithm

Let us fix now an arbitrary (small) $\epsilon > 0$.

We show how to reduce the above error probability $2 \cdot 0.4714$ to ϵ.

- We run k (odd) many independent copies of the AMS algorithm on the input stream.

- In other words: each copy of the algorithm randomly chooses its own hash function and these choices are made independent from each other.

- Let α_i be the output of the i-th copy of the algorithm.

- At the end we output the median $\hat{\alpha}$ of $\alpha_1, \ldots, \alpha_k$.

Probability amplification for the AMS algorithm

Let us analyze the error:

Define a Bernoulli random variable F_i:

$$F_i = \begin{cases} 1 \text{ if } \alpha_i \geq 3|A| \\ 0 \text{ else} \end{cases}$$

Note: $\text{Prob}[F_i = 1] = \text{Prob}[F_j = 1]$ for all $1 \leq i, j \leq k$.

Let $p = \text{Prob}[F_i = 1]$.

Our previous analysis yields $p \leq \frac{\sqrt{2}}{3} < 1/2$.

Let $\delta = \frac{1}{2p} - 1 \geq \frac{3}{2\sqrt{2}} - 1$ and note that $0 < \delta < 1$.

Also note: If $\hat{\alpha} \geq 3|A|$ then $\sum_{i=1}^{k} F_i \geq k/2$.

Probability amplification for the AMS algorithm

With the first Chernoff bound we get (note that $(1 + \delta)p = 1/2$):

$$
\begin{aligned}
\text{Prob}[\hat{\alpha} \ge 3|A|] & \le \text{Prob}\Big[\sum_{i=1}^{k} F_i \ge k/2\Big] \\
& = \text{Prob}\Big[\sum_{i=1}^{k} F_i \ge (1 + \delta)pk\Big] \\
& \le e^{-\delta^2 pk/3}
\end{aligned}
$$

For $k \ge \frac{3}{\delta^2 p} \cdot \ln(2/\epsilon) = \frac{3p}{(1/2-p)^2} \cdot \ln(2/\epsilon) \in \Theta(\log(1/\epsilon))$ we get

$$
\text{Prob}[\hat{\alpha} \ge 3|A|] \le e^{-\ln(2/\epsilon)} = \epsilon/2.
$$

Analogously, the second Chernoff bound yields $\text{Prob}[\hat{\alpha} \le |A|/3] \le \epsilon/2$.

In total: $\text{Prob}[|A|/3 < \hat{\alpha} < 3|A|] \ge 1 - \epsilon$.

Average height of search trees

We want to show that the average height of a binary search tree with node set $\{1, \ldots, n\}$ is $\mathcal{O}(\log n)$.
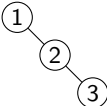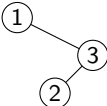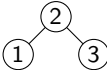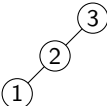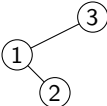
Let $\mathcal{B}_n$ be the set of all binary search trees with node set $\{1, \ldots, n\}$.
($\mathcal{B}_0$ only contains the empty tree $\varnothing$ and we define height($\varnothing$) $= -\infty$).

We generate a search tree $B \in \mathcal{B}_n$ using the following random experiment:

- Choose randomly (uniform distribution) a permutation $[\pi_1, \pi_2, \ldots, \pi_n]$ of $[1, 2, \ldots, n]$.
  Each of the $n!$ permutations is chosen with probability $1/n!$.

- Built a search by inserting the elements $1, \ldots, n$ in the order $\pi_1, \pi_2, \ldots, \pi_n$ into the search tree.

- Note: different permutations can lead to the same search tree.

Average height of search trees

Example:

- $[1, 2, 3]$ leads to with probability $1/6$.

- $[1, 3, 2]$ leads to with probability $1/6$.

- $[2, 1, 3]$ and $[2, 3, 1]$ both lead to with probability $1/3$.

- $[3, 2, 1]$ leads to with probability $1/6$.

- $[3, 1, 2]$ leads to with probability $1/6$.

Average height of search trees

The following random experiment yields for every search tree the same probability as the experiment from the previous slide:

- If $n \geq 2$, choose randomly (using the uniform distribution) an element $i \in \{1, \ldots, n\}$.

  Every element is chosen with probability $1/n$.

- Then produce recursively (using the same experiment) a search tree $L \in \mathcal{B}_{i-1}$ (resp., $R \in \mathcal{B}_{n-i}$).

- Replace in R every node j by $i + j$.

- The full search tree has root i and left (right) subtree L (R).

Note: this random experiment does not yield the uniform distribution on all search trees on the nodes $1, \ldots, n$.
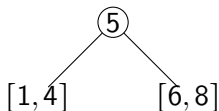
Average height of search trees

Example:

$$[1, 8]$$
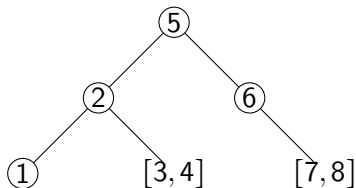
$\text{Prob}[B] = 1/8$

Average height of search trees

Example:

$$\mathsf{Prob}[B] = 1/8 \cdot 1/4 \cdot 1/3$$
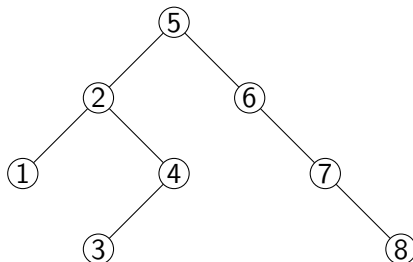
Average height of search trees

Example:

$\mathsf{Prob}[B] = 1/8 \cdot 1/4 \cdot 1/3 \cdot 1/2 \cdot 1/2$

Average height of search trees

Example:

$\mathsf{Prob}[B] = 1/8 \cdot 1/4 \cdot 1/3 \cdot 1/2 \cdot 1/2 = 1/384$

Average height of search trees

We define the following random variables:

- H_n is the height of a randomly generated search tree $B \in \mathcal{B}_n$.
- $X_n = 2^{H_n}$

Theorem 24

For the expected value

$$E[H_n] = \sum_{B \in \mathcal{B}_n} \mathrm{Prob}[B] \cdot \mathrm{height}(B)$$

we have $E[H_n] \leq 3 \cdot \log_2(n)$.

Proof: We first show that $E[X_n]$ is bounded by a polynomial $p(n)$.

Let B be a search tree with root $i \in [1, n]$ and left (right) subtree L (R).

We obtain $\mathrm{height}(B) = 1 + \max\{\mathrm{height}(L), \mathrm{height}(R)\}$ and hence:

Average height of search trees

$$
\begin{aligned}
E[X_n] &= \sum_{B \in \mathcal{B}_n} \mathrm{Prob}[B] \cdot 2^{\mathrm{height}(B)} \\
&= \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \frac{1}{n} \mathrm{Prob}(L) \mathrm{Prob}(R) \, 2^{1+\max\{\mathrm{height}(L), \mathrm{height}(R)\}} \\
&= \frac{2}{n} \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(L) \mathrm{Prob}(R) \max\{2^{\mathrm{height}(L)}, 2^{\mathrm{height}(R)}\} \\
&\leq \frac{2}{n} \sum_{i=1}^{n} \sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(L) \mathrm{Prob}(R) \big(2^{\mathrm{height}(L)} + 2^{\mathrm{height}(R)}\big) \\
&= \frac{2}{n} \sum_{i=1}^{n} \bigg(\sum_{L \in \mathcal{B}_{i-1}} \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(L) \mathrm{Prob}(R) \, 2^{\mathrm{height}(L)} + \\
& \qquad\qquad \sum_{R \in \mathcal{B}_{n-i}} \sum_{L \in \mathcal{B}_{i-1}} \mathrm{Prob}(L) \mathrm{Prob}(R) \, 2^{\mathrm{height}(R)} \bigg)
\end{aligned}
$$

Average height of search trees

$$
= \frac{2}{n} \sum_{i=1}^{n} \left(\sum_{L \in \mathcal{B}_{i-1}} \mathrm{Prob}(L) 2^{\mathrm{height}(L)} + \sum_{R \in \mathcal{B}_{n-i}} \mathrm{Prob}(R) 2^{\mathrm{height}(R)} \right)
$$

$$
= \frac{2}{n} \sum_{i=1}^{n} (E[X_{i-1}] + E[X_{n-i}]) = \frac{4}{n} \sum_{i=0}^{n-1} E[X_i]
$$

Claim 1: $\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$ for $n \geq 1$.

Proof by induction on n:

$n = 1$: $\sum_{i=0}^{0} \binom{i+3}{3} = \binom{3}{3} = 1 = \binom{1+3}{4}$.

If $n \geq 2$, then

$$
\sum_{i=0}^{n-1} \binom{i+3}{3} = \sum_{i=0}^{n-2} \binom{i+3}{3} + \binom{n+2}{3} = \binom{n+2}{4} + \binom{n+2}{3} = \binom{n+3}{4}.
$$

Average height of search trees

In the last equality we use the formula

$$\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}.$$

Claim 2: $E[X_n] \leq \frac{1}{4}\binom{n+3}{3}$.

Proof by induction on n:

$n = 0$: $E[X_0] = 2^{-\infty} = 0 \leq \frac{1}{4}\binom{3}{3}$

$n = 1$: $E[X_1] = 2^0 = 1 = \frac{1}{4}\binom{4}{3}$

Now assume that $n \geq 2$:

Average height of search trees

$$
\begin{aligned}
E[X_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[X_i] \\
&\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\
&= \frac{1}{n} \binom{n+3}{4} \\
&= \frac{1}{n} \frac{(n+3)!}{4!(n-1)!} \\
&= \frac{1}{4} \frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4} \binom{n+3}{3}
\end{aligned}
$$

Average height of search trees

Moreover, $\frac{1}{4}\binom{n+3}{3} = \frac{(n+3)\cdot(n+2)\cdot(n+1)}{4\cdot3\cdot2} \le n^3$ for $n \ge 1$ and hence $E[X_n] \le n^3$.

The function $x \mapsto 2^x$ ist convex.

With Jensen's inequality (see Algorithms I or Slide 132) we get

$$2^{E[H_n]} \le E[2^{H_n}] = E[X_n] \le n^3.$$

We finally obtain $E[H_n] \le 3\log_2(n)$. □

Remark: If we assume the uniform distribution on all search trees (every binary search tree has the same probability) then the average height is $\Theta(\sqrt{n})$.