Computing Order Statistics in the Farey Sequence

Corina E. Pătraşcu[1] Mihai Pătraşcu[2]

[1]Department of Mathematics
Harvard University

[2]MIT CSAIL

ANTS VI

$\mathcal{F}_n = \{\frac{p}{q} \mid 0 < p < q \leq n, \quad gcd(p,q) = 1\}$

(sometimes include $\frac{0}{1}$ and $\frac{1}{1}$)

Total number of fractions: $\frac{3}{\pi^2}n^2 + O(n \log n)$.

Properties

- P1. (Farey 1816) If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n \Rightarrow \frac{p}{q} \leq \frac{p+p'}{q+q'} \leq \frac{p'}{q'}$ (mediant) is also Farey.
- P2. If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n$, then the next fraction $\frac{p''}{q''}$ is given by:
  $p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q$

P2 suggests ideal algorithm for generating $\mathcal{F}_n$ in $O(n^2)$ time and $O(1)$ space.

The Farey Sequence

$$\mathcal{F}_n = \{\tfrac{p}{q} \mid 0 < p < q \le n, \quad gcd(p,q) = 1\}$$

(sometimes include $\frac{0}{1}$ and $\frac{1}{1}$)

Total number of fractions: $\frac{3}{\pi^2}n^2 + O(n \log n)$.

Properties

- P1. (Farey 1816) If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n \Rightarrow \frac{p}{q} \le \frac{p+p'}{q+q'} \le \frac{p'}{q'}$ (mediant) is also Farey.
- P2. If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n$, then the next fraction $\frac{p''}{q''}$ is given by:
  $p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q$

P2 suggests ideal algorithm for generating $\mathcal{F}_n$ in $O(n^2)$ time and $O(1)$ space.

The Farey Sequence

$\mathcal{F}_n = \{\frac{p}{q} \mid 0 < p < q \le n, \quad gcd(p,q) = 1\}$

(sometimes include $\frac{0}{1}$ and $\frac{1}{1}$)

Total number of fractions: $\frac{3}{\pi^2}n^2 + O(n \log n)$.

Properties

- P1. (Farey 1816) If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n \Rightarrow \frac{p}{q} \le \frac{p+p'}{q+q'} \le \frac{p'}{q'}$ (mediant) is also Farey.
- P2. If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n$, then the next fraction $\frac{p''}{q''}$ is given by:
  $p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q$

P2 suggests ideal algorithm for generating $\mathcal{F}_n$ in $O(n^2)$ time and $O(1)$ space.

The Farey Sequence

$\mathcal{F}_n = \{\frac{p}{q} \mid 0 < p < q \leq n, \quad gcd(p, q) = 1\}$

(sometimes include $\frac{0}{1}$ and $\frac{1}{1}$)

Total number of fractions: $\frac{3}{\pi^2}n^2 + O(n \log n)$.

Properties

- P1. (Farey 1816) If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n \Rightarrow \frac{p}{q} \leq \frac{p+p'}{q+q'} \leq \frac{p'}{q'}$ (mediant) is also Farey.
- P2. If $\frac{p}{q}$ and $\frac{p'}{q'}$ consecutive in $\mathcal{F}_n$, then the next fraction $\frac{p''}{q''}$ is given by:
  $p'' = \lfloor\frac{q+n}{q'}\rfloor p' - p, \quad q'' = \lfloor\frac{q+n}{q'}\rfloor q' - q$

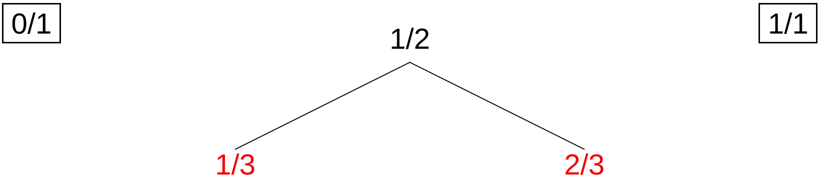
P2 suggests ideal algorithm for generating $\mathcal{F}_n$ in $O(n^2)$ time and $O(1)$ space.

Stern-Brocot Tree

Start with $\frac{0}{1}$ and $\frac{1}{1}$ and insert mediant between any two consecutive fractions in the in-order traversal of the tree.

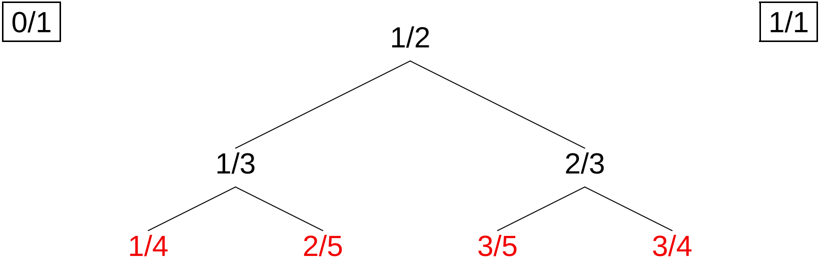
$\boxed{0/1}$ 1/2 $\boxed{1/1}$

Start with $\frac{0}{1}$ and $\frac{1}{1}$ and insert mediant between any two consecutive fractions in the in-order traversal of the tree.



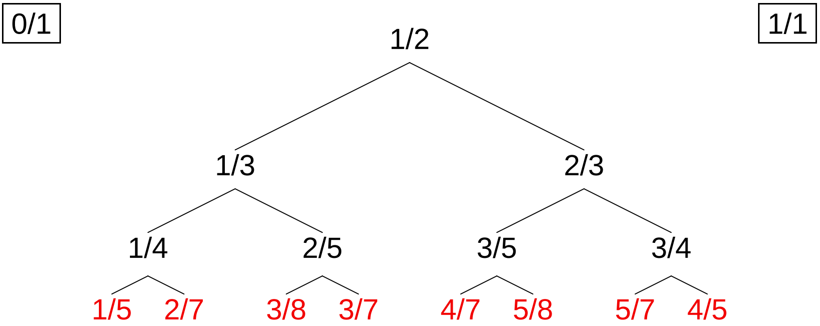
Stern-Brocot Tree

Start with $\frac{0}{1}$ and $\frac{1}{1}$ and insert mediant between any two consecutive fractions in the in-order traversal of the tree.



Stern-Brocot Tree

Start with $\frac{0}{1}$ and $\frac{1}{1}$ and insert mediant between any two consecutive fractions in the in-order traversal of the tree.



Computing order Statistics

Problem 1 (Order Statistic)

Given n and k, generate the k-th element of $\mathcal{F}_n$.

give reduction to:

Problem 2 (Fraction Rank)

Given a fraction, determine its rank in the Farey sequence.

- initial algorithm: time $O(n \log^2 n)$ and space $O(n)$;
- first improvement: time $O(n \log n)$ and space $O(\sqrt{n})$;
- second improvement: time $O(n \log n)$ and space $O(n^{1/3} \log^{2/3} n)$ (fraction rank in $O(n)$ time);
- also find the number of Farey fractions in time $n^{5/6+o(1)}$;
- recent progress: count number of Farey fractions in $n^{4/5+o(1)}$.

Computing order Statistics

Problem 1 (Order Statistic)

Given n and k, generate the k-th element of $\mathcal{F}_n$.

give reduction to:

Problem 2 (Fraction Rank)

Given a fraction, determine its rank in the Farey sequence.

- initial algorithm: time $O(n \log^2 n)$ and space $O(n)$;
- first improvement: time $O(n \log n)$ and space $O(\sqrt{n})$;
- second improvement: time $O(n \log n)$ and space $O(n^{1/3} \log^{2/3} n)$ (fraction rank in $O(n)$ time);
- also find the number of Farey fractions in time $n^{5/6+o(1)}$;
- recent progress: count number of Farey fractions in $n^{4/5+o(1)}$.

Computing order Statistics

Problem 1 (Order Statistic)

Given n and k, generate the k-th element of $\mathcal{F}_n$.

give reduction to:

Problem 2 (Fraction Rank)

Given a fraction, determine its rank in the Farey sequence.

- initial algorithm: time $O(n \log^2 n)$ and space $O(n)$;
- first improvement: time $O(n \log n)$ and space $O(\sqrt{n})$;
- second improvement: time $O(n \log n)$ and space $O(n^{1/3} \log^{2/3} n)$ (fraction rank in $O(n)$ time);
- also find the number of Farey fractions in time $n^{5/6+o(1)}$;
- recent progress: count number of Farey fractions in $n^{4/5+o(1)}$.

Computing order Statistics

Problem 1 (Order Statistic)

Given n and k, generate the k-th element of $\mathcal{F}_n$.

give reduction to:

Problem 2 (Fraction Rank)

Given a fraction, determine its rank in the Farey sequence.

- initial algorithm: time $O(n \log^2 n)$ and space $O(n)$;
- first improvement: time $O(n \log n)$ and space $O(\sqrt{n})$;
- second improvement: time $O(n \log n)$ and space $O(n^{1/3} \log^{2/3} n)$ (fraction rank in $O(n)$ time);
- also find the number of Farey fractions in time $n^{5/6+o(1)}$;
- recent progress: count number of Farey fractions in $n^{4/5+o(1)}$.

Computing order Statistics

Problem 1 (Order Statistic)

Given n and k, generate the k-th element of $\mathcal{F}_n$.

give reduction to:

Problem 2 (Fraction Rank)

Given a fraction, determine its rank in the Farey sequence.

- initial algorithm: time $O(n \log^2 n)$ and space $O(n)$;
- first improvement: time $O(n \log n)$ and space $O(\sqrt{n})$;
- second improvement: time $O(n \log n)$ and space $O(n^{1/3} \log^{2/3} n)$ (fraction rank in $O(n)$ time);
- also find the number of Farey fractions in time $n^{5/6+o(1)}$;
- recent progress: count number of Farey fractions in $n^{4/5+o(1)}$.

Reduction from Order Statistic to Fraction Rank

Want to determine k-th fraction:

- use binary search to determine j such that answer is in the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;
  - guess j and determine $r = \text{rank}(\frac{j}{n})$ in $\mathcal{F}_n$;
  - if $r < k$ search above j; else, search below; if $r = k$, done.
- we use $O(\log n)$ calls to the fraction rank subroutine;
- note that in $\left[\frac{j}{n}, \frac{j+1}{n}\right)$, there is at most one fraction for each denominator (because length of interval is $\frac{1}{n}$).
- this fraction, for denominator q, if it exists, has numerator $\lfloor \frac{(j+1)q-1}{n} \rfloor$;

Reduction from Order Statistic to Fraction Rank

Want to determine k-th fraction:

- use binary search to determine j such that answer is in the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;
  - guess j and determine $r = \text{rank}(\frac{j}{n})$ in $\mathcal{F}_n$;
  - if $r < k$ search above j; else, search below; if $r = k$, done.
- we use $O(\log n)$ calls to the fraction rank subroutine;
- note that in $\left[\frac{j}{n}, \frac{j+1}{n}\right)$, there is at most one fraction for each denominator (because length of interval is $\frac{1}{n}$).
- this fraction, for denominator q, if it exists, has numerator $\left\lfloor \frac{(j+1)q-1}{n} \right\rfloor$;

Reduction from Order Statistic to Fraction Rank

Want to determine k-th fraction:

- use binary search to determine j such that answer is in the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;
  - guess j and determine $r = \text{rank}(\frac{j}{n})$ in $\mathcal{F}_n$;
  - if $r < k$ search above j; else, search below; if $r = k$, done.
- we use $O(\log n)$ calls to the fraction rank subroutine;
- note that in $\left[\frac{j}{n}, \frac{j+1}{n}\right)$, there is at most one fraction for each denominator (because length of interval is $\frac{1}{n}$).
- this fraction, for denominator q, if it exists, has numerator $\lfloor \frac{(j+1)q-1}{n} \rfloor$;

Reduction from Order Statistic to Fraction Rank

- must determine fraction with rank $= k - rank(\frac{j}{n})$ among irreducible fractions in interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;

Algorithm – O(n) time and O(1) memory

- generate all fractions in the range;
- as we generate, keep just the minimum strictly greater than $\frac{j}{n}$;
- finally, reduce $\frac{j}{n}$ and the minimum fraction obtained above $\Rightarrow$ two consecutive fractions in $\mathcal{F}_n$;
- use P2 to generate the next one in constant time etc.; keep a count and return the desired fraction.

Reduction from Order Statistic to Fraction Rank

- must determine fraction with rank $= k - rank(\frac{j}{n})$ among irreducible fractions in interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;

Algorithm – O(n) time and O(1) memory

- generate all fractions in the range;
- as we generate, keep just the minimum strictly greater than $\frac{j}{n}$;
- finally, reduce $\frac{j}{n}$ and the minimum fraction obtained above $\Rightarrow$ two consecutive fractions in $\mathcal{F}_n$;
- use P2 to generate the next one in constant time etc.; keep a count and return the desired fraction.

Reduction from Order Statistic to Fraction Rank

- must determine fraction with rank $= k - rank(\frac{j}{n})$ among irreducible fractions in interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$;

Algorithm – O(n) time and $O(1)$ memory

- generate all fractions in the range;
- as we generate, keep just the minimum strictly greater than $\frac{j}{n}$;
- finally, reduce $\frac{j}{n}$ and the minimum fraction obtained above $\Rightarrow$ two consecutive fractions in $\mathcal{F}_n$;
- use P2 to generate the next one in constant time etc.; keep a count and return the desired fraction.

Algorithm for Fraction Rank Problem

More general: Given x real, determine the number of irreducible fractions $\frac{p}{q} \leq x$, with $q \leq n$.

- A_q = the set of such irreducible fractions with denominator q;
- $\{$all fractions in $[0, x)$ with denominator $q\} \longleftrightarrow \{$reduced fractions with denominator d, for all $d|q\}$;
- this gives formula: $\lfloor x \cdot q \rfloor = \sum_{d \leq q, d|q} A_d$;
- so, to compute $rank(x) = \sum_{q=1}^{n} A_q$, we use: $A_q = \lfloor x \cdot q \rfloor - \sum_{d < q, d|q} A_d$.

Algorithm for Fraction Rank Problem

More general: Given x real, determine the number of irreducible fractions $\frac{p}{q} \leq x$, with $q \leq n$.

- $A_q =$ the set of such irreducible fractions with denominator q;
- $\{$all fractions in $[0, x)$ with denominator $q\} \longleftrightarrow \{$reduced fractions with denominator d, for all $d|q\}$;
- this gives formula: $\lfloor x \cdot q \rfloor = \sum_{d \leq q, d|q} A_d$;
- so, to compute $rank(x) = \sum_{q=1}^{n} A_q$, we use: $A_q = \lfloor x \cdot q \rfloor - \sum_{d < q, d|q} A_d$.

Algorithm for Fraction Rank Problem

More general: Given x real, determine the number of irreducible fractions $\frac{p}{q} \leq x$, with $q \leq n$.

- $A_q =$ the set of such irreducible fractions with denominator q;
- $\{$all fractions in $[0, x)$ with denominator $q\} \longleftrightarrow \{$reduced fractions with denominator d, for all $d|q\}$;
- this gives formula: $\lfloor x \cdot q \rfloor = \sum_{d \leq q, d|q} A_d$;
- so, to compute $rank(x) = \sum_{q=1}^{n} A_q$, we use: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$.

Algorithm for Fraction Rank Problem

More general: Given x real, determine the number of irreducible fractions $\frac{p}{q} \leq x$, with $q \leq n$.

- $A_q =$ the set of such irreducible fractions with denominator q;
- $\{$all fractions in $[0, x)$ with denominator $q\} \longleftrightarrow$ $\{$reduced fractions with denominator d, for all $d|q\}$;
- this gives formula: $\lfloor x \cdot q \rfloor = \sum_{d \leq q, d|q} A_d$;
- so, to compute $rank(x) = \sum_{q=1}^{n} A_q$, we use: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$.

Problem: no fast way to iterate over all divisors.

Solution:

- initialize array $T[1..n]$ by $T[q] = \lfloor x \cdot q \rfloor$;
- consider all q's in increasing order from 1 to n;
- for each q, consider its multiples $m \cdot q$ and subtract $T[q]$ from $T[m \cdot q]$;
- at step q, we will have $T[q] = A_q$;
- $rank(x)$ is obtained by summing all the values in the array at the end;
- running time: $O(n + \sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$.

Thus, we have an $O(n \log n)$ time algorithm for the fraction rank problem $\Rightarrow$ $O(n \log^2 n)$ time algorithm for the order statistic problem. Both use $O(n)$ space.

Problem: no fast way to iterate over all divisors.

Solution:

- initialize array $T[1..n]$ by $T[q] = \lfloor x \cdot q \rfloor$;
- consider all q's in increasing order from 1 to n;
- for each q, consider its multiples $m \cdot q$ and subtract $T[q]$ from $T[m \cdot q]$;
- at step q, we will have $T[q] = A_q$;
- $rank(x)$ is obtained by summing all the values in the array at the end;
- running time: $O(n + \sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$.

Thus, we have an $O(n \log n)$ time algorithm for the fraction rank problem $\Rightarrow$ $O(n \log^2 n)$ time algorithm for the order statistic problem. Both use $O(n)$ space.

Algorithm for Fraction Rank Problem

Problem: no fast way to iterate over all divisors.

Solution:

- initialize array $T[1..n]$ by $T[q] = \lfloor x \cdot q \rfloor$;
- consider all q's in increasing order from 1 to n;
- for each q, consider its multiples $m \cdot q$ and subtract $T[q]$ from $T[m \cdot q]$;
- at step q, we will have $T[q] = A_q$;
- $rank(x)$ is obtained by summing all the values in the array at the end;
- running time: $O(n + \sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$.

Thus, we have an $O(n \log n)$ time algorithm for the fraction rank problem $\Rightarrow$ $O(n \log^2 n)$ time algorithm for the order statistic problem. Both use $O(n)$ space.

Algorithm for Fraction Rank Problem

Problem: no fast way to iterate over all divisors.

Solution:

- initialize array $T[1..n]$ by $T[q] = \lfloor x \cdot q \rfloor$;
- consider all q's in increasing order from 1 to n;
- for each q, consider its multiples $m \cdot q$ and subtract $T[q]$ from $T[m \cdot q]$;
- at step q, we will have $T[q] = A_q$;
- $rank(x)$ is obtained by summing all the values in the array at the end;
- running time: $O(n + \sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$.

Thus, we have an $O(n \log n)$ time algorithm for the fraction rank problem $\Rightarrow$ $O(n \log^2 n)$ time algorithm for the order statistic problem. Both use $O(n)$ space.

Improving the Running Time to $O(n \log n)$

Preprocessing $\Rightarrow$ improve time of every call to fraction rank.

Use previous formula: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$; after recursive expansions of A_q's $\Rightarrow$ $rank(x)$ will be a linear combination of $\lfloor x \cdot q \rfloor, \forall q \le n$.

Idea:

- the coefficients of $\lfloor x \cdot q \rfloor$ are independent of x;
- so, precalculate the coefficient of every $\lfloor x \cdot q \rfloor$;
- the numbers at intermediate steps may be large, but $rank(x) < n^2$ eventually, so perform all computations modulo a number greater than n^2 (important because we can manipulate in constant time only numbers with $O(\log n)$ bits).

Improving the Running Time to $O(n \log n)$

Preprocessing $\Rightarrow$ improve time of every call to fraction rank.

Use previous formula: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$; after recursive expansions of A_q's $\Rightarrow$ $rank(x)$ will be a linear combination of $\lfloor x \cdot q \rfloor, \forall q \leq n$.

Idea:

- the coefficients of $\lfloor x \cdot q \rfloor$ are independent of x;
- so, precalculate the coefficient of every $\lfloor x \cdot q \rfloor$;
- the numbers at intermediate steps may be large, but $rank(x) \leq n^2$ eventually, so perform all computations modulo a number greater than n^2; (important because we can manipulate in constant time only numbers with $O(\log n)$ bits).

Improving the Running Time to $O(n \log n)$

Preprocessing $\Rightarrow$ improve time of every call to fraction rank.

Use previous formula: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$; after recursive expansions of A_q's $\Rightarrow$ $rank(x)$ will be a linear combination of $\lfloor x \cdot q \rfloor, \forall q \leq n$.

Idea:

- the coefficients of $\lfloor x \cdot q \rfloor$ are independent of x;
- so, precalculate the coefficient of every $\lfloor x \cdot q \rfloor$;
- the numbers at intermediate steps may be large, but $rank(x) \leq n^2$ eventually, so perform all computations modulo a number greater than n^2; (important because we can manipulate in constant time only numbers with $O(\log n)$ bits).

Improving the Running Time to $O(n \log n)$

Preprocessing $\Rightarrow$ improve time of every call to fraction rank.

Use previous formula: $A_q = \lfloor x \cdot q \rfloor - \sum_{d<q, d|q} A_d$; after recursive expansions of A_q's $\Rightarrow$ $rank(x)$ will be a linear combination of $\lfloor x \cdot q \rfloor, \forall q \leq n$.

Idea:

- the coefficients of $\lfloor x \cdot q \rfloor$ are independent of x;
- so, precalculate the coefficient of every $\lfloor x \cdot q \rfloor$;
- the numbers at intermediate steps may be large, but $rank(x) \leq n^2$ eventually, so perform all computations modulo a number greater than n^2; (important because we can manipulate in constant time only numbers with $O(\log n)$ bits).

Precalculating the Coefficients

Obtain the recursive formula: $C_q = 1 - \sum_{t>q, q|t} C_t$, $\forall q \leq n$.

Proof

- $\lfloor x \cdot q \rfloor$ appears first in A_q;
- A_q subtracted from all its multiples $t \Rightarrow A_t$ contains $\lfloor x \cdot t \rfloor$ with coefficient 1 and $\lfloor x \cdot q \rfloor$ with coefficient -1;
- all operations made with A_t contribute to the coefficient of $\lfloor x \cdot q \rfloor$ by $-1\times$ the coefficient of $\lfloor x \cdot t \rfloor$;
- since A_t is the only one that contains $\lfloor x \cdot t \rfloor$ initially, all operations involving A_t are described by the final coefficient of $\lfloor x \cdot t \rfloor$.

The algorithm calculates C_n down to $C_1 \Rightarrow$ running time: $O(\sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$; this cost is paid once and every call to fraction rank takes $O(n) \Rightarrow$ total time: $O(n \log n)$.

Precalculating the Coefficients

Obtain the recursive formula: $C_q = 1 - \sum_{t>q, q|t} C_t, \forall q \leq n$.

Proof

- $\lfloor x \cdot q \rfloor$ appears first in A_q;
- A_q subtracted from all its multiples $t \Rightarrow A_t$ contains $\lfloor x \cdot t \rfloor$ with coefficient 1 and $\lfloor x \cdot q \rfloor$ with coefficient -1;
- all operations made with A_t contribute to the coefficient of $\lfloor x \cdot q \rfloor$ by $-1\times$ the coefficient of $\lfloor x \cdot t \rfloor$;
- since A_t is the only one that contains $\lfloor x \cdot t \rfloor$ initially, all operations involving A_t are described by the final coefficient of $\lfloor x \cdot t \rfloor$.

The algorithm calculates C_n down to $C_1 \Rightarrow$ running time: $O(\sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$; this cost is paid once and every call to fraction rank takes $O(n) \Rightarrow$ total time: $O(n \log n)$.

Precalculating the Coefficients

Obtain the recursive formula: $C_q = 1 - \sum_{t>q, q|t} C_t$, $\forall q \leq n$.

Proof

- $\lfloor x \cdot q \rfloor$ appears first in A_q;
- A_q subtracted from all its multiples $t \Rightarrow A_t$ contains $\lfloor x \cdot t \rfloor$ with coefficient 1 and $\lfloor x \cdot q \rfloor$ with coefficient -1;
- all operations made with A_t contribute to the coefficient of $\lfloor x \cdot q \rfloor$ by $-1\times$ the coefficient of $\lfloor x \cdot t \rfloor$;
- since A_t is the only one that contains $\lfloor x \cdot t \rfloor$ initially, all operations involving A_t are described by the final coefficient of $\lfloor x \cdot t \rfloor$.

The algorithm calculates C_n down to $C_1 \Rightarrow$ running time: $O(\sum_{q=1}^{n} \frac{n}{q}) = O(n \log n)$; this cost is paid once and every call to fraction rank takes $O(n) \Rightarrow$ total time: $O(n \log n)$.

Improving Space Complexity to $O(\sqrt{n})$

Lemma

$C_q = C_{q'}$ when $\lfloor n/q \rfloor = \lfloor n/q' \rfloor$.

Proof Idea

- consider some q; the term $\lfloor x \cdot q \rfloor$ is first in A_q;
- A_q is subtracted from $A_{m_1 q}$, for all possible m_1;
- the $A_{m_1 q}$'s are now subtracted from $A_{m_1 m_2 q}$, for all possible m_2 etc.;
- the recursion stops only when $\prod m_i \geq \lfloor n/q \rfloor$ so C_q depends only on $\lfloor n/q \rfloor$.

Observation: there are only $\sqrt{n}$ distinct C_q's for $q > \sqrt{n}$.

Improving Space Complexity to $O(\sqrt{n})$

Lemma

$C_q = C_{q'}$ when $\lfloor n/q \rfloor = \lfloor n/q' \rfloor$.

Proof Idea

- consider some q; the term $\lfloor x \cdot q \rfloor$ is first in A_q;
- A_q is subtracted from $A_{m_1 q}$, for all possible m_1;
- the $A_{m_1 q}$'s are now subtracted from $A_{m_1 m_2 q}$, for all possible m_2 etc.;
- the recursion stops only when $\prod m_i \geq \lfloor n/q \rfloor$ so C_q depends only on $\lfloor n/q \rfloor$.

Observation: there are only $\sqrt{n}$ distinct C_q's for $q > \sqrt{n}$.

Improving Space Complexity to $O(\sqrt{n})$

Lemma

$C_q = C_{q'}$ when $\lfloor n/q \rfloor = \lfloor n/q' \rfloor$.

Proof Idea

- consider some q; the term $\lfloor x \cdot q \rfloor$ is first in A_q;
- A_q is subtracted from $A_{m_1 q}$, for all possible m_1;
- the $A_{m_1 q}$'s are now subtracted from $A_{m_1 m_2 q}$, for all possible m_2 etc.;
- the recursion stops only when $\prod m_i \geq \lfloor n/q \rfloor$ so C_q depends only on $\lfloor n/q \rfloor$.

Observation: there are only $\sqrt{n}$ distinct C_q's for $q > \sqrt{n}$.

Avoid Repetitions

Break into two groups:

- $C_1, \ldots C_{\sqrt{n}}$ stored as before;
- instead of storing C_q for $q > \sqrt{n}$, store array with D_r's such that $C_q = D_{\lfloor n/q \rfloor}$, for any $q > \lfloor n/q \rfloor$.

Observation: both arrays take $O(\sqrt{n})$ space and fraction rank algorithm remains trivial.

Precomputing D_r and C_q in $O(\sqrt{n})$ space

- rewrite: $C_q = 1 - \sum_{t=2}^{\lfloor n/q \rfloor} C_{tq}$;
- $C_q = D_{\lfloor n/q \rfloor} \Rightarrow C_{tq} = D_{\lfloor n/tq \rfloor}$ and since $\lfloor \frac{n}{tq} \rfloor = \lfloor \frac{\lfloor n/q \rfloor}{t} \rfloor$;
- $\Rightarrow$ we obtain $D_r = 1 - \sum_{t=2}^{r} D_{\lfloor r/t \rfloor}$.

Time for computing D_r: quadratic in size of table $\Rightarrow O(n)$.

Avoid Repetitions

Break into two groups:

- $C_1, \ldots C_{\sqrt{n}}$ stored as before;
- instead of storing C_q for $q > \sqrt{n}$, store array with D_r's such that $C_q = D_{\lfloor n/q \rfloor}$, for any $q > \lfloor n/q \rfloor$.

Observation: both arrays take $O(\sqrt{n})$ space and fraction rank algorithm remains trivial.

Precomputing D_r and C_q in $O(\sqrt{n})$ space

- rewrite: $C_q = 1 - \sum_{t=2}^{\lfloor n/q \rfloor} C_{tq}$;
- $C_q = D_{\lfloor n/q \rfloor} \Rightarrow C_{tq} = D_{\lfloor n/tq \rfloor}$ and since $\lfloor \frac{n}{tq} \rfloor = \lfloor \frac{\lfloor n/q \rfloor}{t} \rfloor$;
- $\Rightarrow$ we obtain $D_r = 1 - \sum_{t=2}^{r} D_{\lfloor r/t \rfloor}$.

Time for computing D_r: quadratic in size of table $\Rightarrow O(n)$.

Avoid Repetitions

Break into two groups:

- $C_1, \ldots C_{\sqrt{n}}$ stored as before;
- instead of storing C_q for $q > \sqrt{n}$, store array with D_r's such that $C_q = D_{\lfloor n/q \rfloor}$, for any $q > \lfloor n/q \rfloor$.

Observation: both arrays take $O(\sqrt{n})$ space and fraction rank algorithm remains trivial.

Precomputing D_r and C_q in $O(\sqrt{n})$ space

- rewrite: $C_q = 1 - \sum_{t=2}^{\lfloor n/q \rfloor} C_{tq}$;
- $C_q = D_{\lfloor n/q \rfloor} \Rightarrow C_{tq} = D_{\lfloor n/tq \rfloor}$ and since $\lfloor \frac{n}{tq} \rfloor = \lfloor \frac{\lfloor n/q \rfloor}{t} \rfloor$;
- $\Rightarrow$ we obtain $D_r = 1 - \sum_{t=2}^{r} D_{\lfloor r/t \rfloor}$.

Time for computing D_r: quadratic in size of table $\Rightarrow O(n)$.

Avoid Repetitions

Break into two groups:

- $C_1, \ldots C_{\sqrt{n}}$ stored as before;
- instead of storing C_q for $q > \sqrt{n}$, store array with D_r's such that $C_q = D_{\lfloor n/q \rfloor}$, for any $q > \lfloor n/q \rfloor$.

Observation: both arrays take $O(\sqrt{n})$ space and fraction rank algorithm remains trivial.

Precomputing D_r and C_q in $O(\sqrt{n})$ space

- rewrite: $C_q = 1 - \sum_{t=2}^{\lfloor n/q \rfloor} C_{tq}$;
- $C_q = D_{\lfloor n/q \rfloor} \Rightarrow C_{tq} = D_{\lfloor n/tq \rfloor}$ and since $\lfloor \frac{n}{tq} \rfloor = \lfloor \frac{\lfloor n/q \rfloor}{t} \rfloor$;
- $\Rightarrow$ we obtain $D_r = 1 - \sum_{t=2}^{r} D_{\lfloor r/t \rfloor}$.

Time for computing D_r: quadratic in size of table $\Rightarrow O(n)$.

Relation to Factorization

Conjecture: A polynomial time algorithm for factorization does not exist.

We will show that this implies that no polynomial time algorithm (i.e. $O(\text{poly}\log n)$) exists for the order statistic problem.

Reduction from Fraction Rank to Order Statistic

- assume we have a poly-time algorithm for order statistic;
- do binary search: guess the rank, find fraction with that rank, compare to our fraction and search below or above;
- $\Rightarrow$ $O(\log n)$ calls to order statistic;
- $\Rightarrow$ we have a polynomial time algorithm for the fraction rank problem.

Relation to Factorization

Conjecture: A polynomial time algorithm for factorization does not exist.

We will show that this implies that no polynomial time algorithm (i.e. $O(\mathrm{poly}\log n)$) exists for the order statistic problem.

Reduction from Fraction Rank to Order Statistic

- assume we have a poly-time algorithm for order statistic;
- do binary search: guess the rank, find fraction with that rank, compare to our fraction and search below or above;
- $\Rightarrow$ $O(\log n)$ calls to order statistic;
- $\Rightarrow$ we have a polynomial time algorithm for the fraction rank problem.

Algorithm for Factorization

It is based on yet another problem:

Problem: Given n and $k \leq n$ such that $gcd(k, n) = 1$, report the number of integers in $[2, k]$ that are relatively prime to n.

Algorithm for Factorization

- assume a polynomial time algorithm for the above problem;
- use binary search to find factor of n:
  - guess k; if $(k, n) \neq 1$, we can find a factor using Euclid's algorithm;
  - if $(k, n) = 1$, by above problem, we know the number of numbers in $[2, k]$ relatively prime to n:
    - if this number is $k - 1$, the smallest factor of n is $> k$;
    - otherwise, there is at least a factor below k.

$\Rightarrow$ polynomial time algorithm for factorization.

Algorithm for Factorization

It is based on yet another problem:

Problem: Given n and $k \leq n$ such that $gcd(k, n) = 1$, report the number of integers in $[2, k]$ that are relatively prime to n.

Algorithm for Factorization

- assume a polynomial time algorithm for the above problem;
- use binary search to find factor of n:
  - guess k; if $(k, n) \neq 1$, we can find a factor using Euclid's algorithm;
  - if $(k, n) = 1$, by above problem, we know the number of numbers in $[2, k]$ relatively prime to n:
    - if this number is $k - 1$, the smallest factor of n is $> k$;
    - otherwise, there is at least a factor below k.

$\Rightarrow$ polynomial time algorithm for factorization.

Algorithm for Factorization

It is based on yet another problem:

Problem: Given n and $k \leq n$ such that $gcd(k, n) = 1$, report the number of integers in $[2, k]$ that are relatively prime to n.

Algorithm for Factorization

- assume a polynomial time algorithm for the above problem;
- use binary search to find factor of n:
  - guess k; if $(k, n) \neq 1$, we can find a factor using Euclid's algorithm;
  - if $(k, n) = 1$, by above problem, we know the number of numbers in $[2, k]$ relatively prime to n:
    - if this number is $k - 1$, the smallest factor of n is $> k$;
    - otherwise, there is at least a factor below k.

$\Rightarrow$ polynomial time algorithm for factorization.

Relation between Farey and Previous Problem

Problem: Given n and $k \leq n$ such that $gcd(k,n) = 1$, report the number of integers in $[2,k]$ that are relatively prime to n.

- all $i \in [2,k]$ such that $(i,n) = 1$ give fractions $\frac{i}{n} \in \mathcal{F}_n$;
- first, find $r = rank(\frac{k}{n})$ in $\mathcal{F}_n$ (fraction rank);
- then, find the fraction of rank $r-1$ in $\mathcal{F}_n$ (order statistic);
- since $\frac{k}{n}$ is irreducible and it is the mediant of neighboring fractions $\Rightarrow$ the preceding fraction must have denominator $< n$;
- find the rank of this preceding fraction in $\mathcal{F}_{n-1}$, say t;
- the difference $r - t =$ number of irreducible fractions $\frac{i}{n} \leq \frac{k}{n}$.

Problem: Given n and $k \leq n$ such that $gcd(k,n) = 1$, report the number of integers in $[2,k]$ that are relatively prime to n.

- all $i \in [2,k]$ such that $(i,n) = 1$ give fractions $\frac{i}{n} \in \mathcal{F}_n$;
- first, find $r = rank(\frac{k}{n})$ in $\mathcal{F}_n$ (fraction rank);
- then, find the fraction of rank $r-1$ in $\mathcal{F}_n$ (order statistic);
- since $\frac{k}{n}$ is irreducible and it is the mediant of neighboring fractions $\Rightarrow$ the preceding fraction must have denominator $< n$;
- find the rank of this preceding fraction in $\mathcal{F}_{n-1}$, say t;
- the difference $r - t =$ number of irreducible fractions $\frac{i}{n} \leq \frac{k}{n}$.

Relation between Farey and Previous Problem

Problem: Given n and $k \leq n$ such that $gcd(k, n) = 1$, report the number of integers in $[2, k]$ that are relatively prime to n.

- all $i \in [2, k]$ such that $(i, n) = 1$ give fractions $\frac{i}{n} \in \mathcal{F}_n$;
- first, find $r = rank(\frac{k}{n})$ in $\mathcal{F}_n$ (fraction rank);
- then, find the fraction of rank $r - 1$ in $\mathcal{F}_n$ (order statistic);
- since $\frac{k}{n}$ is irreducible and it is the mediant of neighboring fractions $\Rightarrow$ the preceding fraction must have denominator $< n$;
- find the rank of this preceding fraction in $\mathcal{F}_{n-1}$, say t;
- the difference $r - t =$ number of irreducible fractions $\frac{i}{n} \leq \frac{k}{n}$.

Relation between Farey and Previous Problem

Problem: Given n and $k \leq n$ such that $gcd(k, n) = 1$, report the number of integers in $[2, k]$ that are relatively prime to n.

- all $i \in [2, k]$ such that $(i, n) = 1$ give fractions $\frac{i}{n} \in \mathcal{F}_n$;
- first, find $r = rank(\frac{k}{n})$ in $\mathcal{F}_n$ (fraction rank);
- then, find the fraction of rank $r - 1$ in $\mathcal{F}_n$ (order statistic);
- since $\frac{k}{n}$ is irreducible and it is the mediant of neighboring fractions $\Rightarrow$ the preceding fraction must have denominator $< n$;
- find the rank of this preceding fraction in $\mathcal{F}_{n-1}$, say t;
- the difference $r - t =$ number of irreducible fractions $\frac{i}{n} \leq \frac{k}{n}$.

Reduction to Fraction Rank

Q: Assume a poly-time algorithm just for fraction rank. Does the previous reduction still hold? – this would imply hardness of fraction rank.
A:

- consider the fractions $\frac{k-1}{n}$ and $\frac{k+1}{n}$; since their difference is $\frac{2}{n}$, $\exists$ only one fraction in this range with denominator $n-1$;
- find this fraction ($O(1)$) and reduce it $\Rightarrow$ $O(\log n)$ time;
- find the ranks of this fraction in $\mathcal{F}_{n-1}$ and $\mathcal{F}_n$; their difference will give the number of irreducible fractions $\frac{i}{n} < \frac{k}{n}$ (possibly plus one due to $\frac{k}{n}$; problem solved by comparing our fraction to $\frac{k}{n}$);

Reduction to Fraction Rank

Q: Assume a poly-time algorithm just for fraction rank. Does the previous reduction still hold? – this would imply hardness of fraction rank.
A: Yes.

- consider the fractions $\frac{k-1}{n}$ and $\frac{k+1}{n}$; since their difference is $\frac{2}{n}$, $\exists$ only one fraction in this range with denominator $n-1$;
- find this fraction ($O(1)$) and reduce it $\Rightarrow$ $O(\log n)$ time;
- find the ranks of this fraction in $\mathcal{F}_{n-1}$ and $\mathcal{F}_n$; their difference will give the number of irreducible fractions $\frac{i}{n} < \frac{k}{n}$ (possibly plus one due to $\frac{k}{n}$; problem solved by comparing our fraction to $\frac{k}{n}$);

Reduction to Fraction Rank

Q: Assume a poly-time algorithm just for fraction rank. Does the previous reduction still hold? – this would imply hardness of fraction rank.
A: Yes.

- consider the fractions $\frac{k-1}{n}$ and $\frac{k+1}{n}$; since their difference is $\frac{2}{n}$, $\exists$ only one fraction in this range with denominator $n-1$;
- find this fraction ($O(1)$) and reduce it $\Rightarrow$ $O(\log n)$ time;
- find the ranks of this fraction in $\mathcal{F}_{n-1}$ and $\mathcal{F}_n$; their difference will give the number of irreducible fractions $\frac{i}{n} < \frac{k}{n}$ (possibly plus one due to $\frac{k}{n}$; problem solved by comparing our fraction to $\frac{k}{n}$);

Reduction to Fraction Rank

Q: Assume a poly-time algorithm just for fraction rank. Does the previous reduction still hold? – this would imply hardness of fraction rank.
A: Yes.

- consider the fractions $\frac{k-1}{n}$ and $\frac{k+1}{n}$; since their difference is $\frac{2}{n}$, $\exists$ only one fraction in this range with denominator $n-1$;
- find this fraction ($O(1)$) and reduce it $\Rightarrow$ $O(\log n)$ time;
- find the ranks of this fraction in $\mathcal{F}_{n-1}$ and $\mathcal{F}_n$; their difference will give the number of irreducible fractions $\frac{i}{n} < \frac{k}{n}$ (possibly plus one due to $\frac{k}{n}$; problem solved by comparing our fraction to $\frac{k}{n}$);

THE END

Thank you!