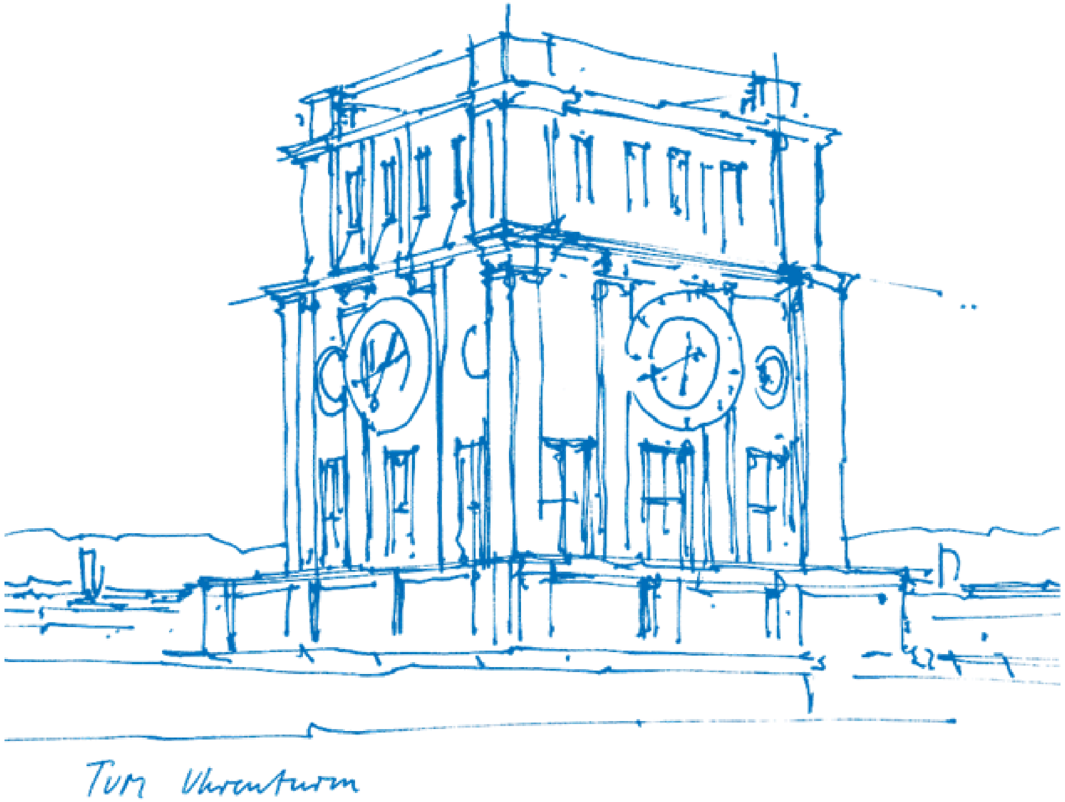
AACPP 2025

Week 9: Number Theory

Mateusz Gienieczko, Mykola Morozov

School of Computation, Information and Technology
Technical University of Munich

2025.07.01



Sixth round – survey

Seventh round

Deadline – 08.07.2025, 10:00 AM.

Only one task this time!

PSS – Purrfect Scent Schedule

Given a sequence of n keys and their values find a consecutive subsequence maximising the sum of values of *unique* keys.

Obvious brute force in $\mathcal{O}(n^3)$ (check every subsequence in linear time).

PSS – Purrfect Scent Schedule

How do we calculate a value of a subsequence?

Sum goes up by $v(a)$ on the first occurrence of a and goes *down* by $v(a)$ on the second occurrence of a. Other occurrences contribute 0. We are interested in the prefix sum.

k_i	5	6	2	5	2	8	5	5	4	4	2
$v(k_i)$	8	5	3	8	3	6	8	8	7	7	3
c_i	$+8$	$+5$	$+3$	-8	-3	$+6$	0	0	$+7$	-7	0
s_i	8	13	16	8	5	11	11	11	18	11	11

Easy change to get $\mathcal{O}(n^2)$ – for each $0 \le i < n$ start at i, go left to right calculating the current sum, return the maximum.

PSS – Purrfect Scent Schedule

Model solution uses a segment tree to maintain these sums.

For each key precompute a queue of the indices at which it occurs.

Initialise with the full sequence.

For $0 \leq i < n$:

- find two next occurrences of a_i, $k > j > i$ (if none one can set $n + 1$);
- set c_i to 0, c_j to $+v(a)$, c_k to $-v(a)$;
- query the sum on the entire tree.

$\mathcal{O}(n \log n)$.

BUR – Cat Burglars

Read the statement carefully because it is rather complicated.

Key observation is that we should consider what the optimal proposals are for cats in reverse order, since the decision of i depends on the decision of $i+1$.

We will denote by $f_i(j)$ (defined for $j \geq i$) the number of bundles j-th cat receives assuming cats $k < i$ were cast away. If j is cast away then $f_i(j) = -1$.

Clearly $f_n(n) = m$.

BUR – Cat Burglars

Let's design a slow solution to better understand the task.

Assume we know f_{i+1} and want to compute f_i.

Cats for whom f_{i+1} equals -1 are freebies, we pay them 0 and get their votes.

The rest we can order by $f_{i+1}(j) + a_j$ in order to pay the least possible. Ties are broken by j, i.e. higher indices get paid first.

Once we no longer need the votes everyone else gets 0.

If we run out of money then $f_i(i) = -1$ and for all $j \neq i$ $f_i(j) = f_{i+1}(j)$.

BUR – Cat Burglars

```
previous.push((m, n))
for i in [n - 1, 1]
  current.clear()
  sort previous by cost,idx
  votes = (n - i) / 2
  money = m
  for cost,j in previous
    if votes == 0 { curr.push((a[j], j)) }
    else if money >= cost
      money -= cost
      votes -= 1
      current.push((cost + a[j], j))
    else { break }
  if votes == 0
    first_to_win = i
    current.push(
      (money + a[j], i))
    swap(previous, current)
  else
    previous.push((0, i))
// end for
for cost,j in previous
  if j < first_to_win
    result[j] = -1
  else
    result[j] = cost - a[j]
```

BUR – Cat Burglars

This gives $\mathcal{O}\left(n^2 \log n\right)$ with the sort.

One can "optimise" to $\mathcal{O}\left(n^2\right)$ by noticing we don't actually need a sort, we need to find the k-th element for $k = \left\lfloor \frac{n}{2} \right\rfloor + 1$ and then to split `previous` by that pivot.

BUR – Cat Burglars

What happens to j in the i-th iteration is entirely defined by the pair $\left(f_{i+1}(j), a_j\right)$.

Cats with the same pair behave the same with the exception of index-based tiebreaking.

We will design a solution that works if the number of unique pairs is low and at the end argue that it's indeed the case.

BUR – Cat Burglars

Assume we grouped all cats by $\left(f_{i+1}(j), a_j\right)$.

Notice that when i votes through its proposal there is always some k such that:

(1) every cat where $f_{i+1}(j) < k$ gets paid exactly $f_{i+1}(j)$;

(2) every cat where $f_{i+1}(j) > k$ gets paid 0;

(3) some cats (with highest indices) where $f_{i+1}(j) = k$ get paid k, others 0.

Cats in (1) transition from (c, a) to $(c + a, a)$.

Cats in (2) transition from (c, a) to (a, a).

Highest-indexed cats in (3) transition to $(c + a, a)$, others to (a, a).

BUR – Cat Burglars

If we keep the sets of cats in each group as a BST that allows splitting and merging, then we can keep a BST for each group in some standard map and perform all operations.

- Iterate by the first element of the pair and find out if we have enough cheap votes.
- If yes, handle (1) and (2) directly; then
- for (3) find the index that separates the paid from non-paid cats, split the tree on that index, and then merge them into different new groups.

For the last one we need to binary-search for the index and then query each BST in $(k, \_)$ to count the number of cats in the appropriate suffix.

Mateusz Gienieczko

BUR – Cat Burglars

Assume there are G unique groups.

We iterate over i and then look at each group.

We need to merge some groups together, with a single merge taking at most $\mathcal{O}(\log n)$ for balanced BSTs.

Finally, the split requires an outer binary search and then at most $\mathcal{O}(\log n)$ in queries.

In total we have $\mathcal{O}\left(nG \log^2 n\right)$.

BUR – Cat Burglars

Lemma 1: *For $j \neq i$, $f_i(j) \leq A$.*

Proof: Inductively, cat i assigns a non-zero number of bundles to at most $\left\lfloor \frac{n-i}{2} \right\rfloor$ other cats. Cat $i - 1$ could assign any bundles only to cats that receive 0 from i, since $n - (i - 1) - \left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right)$ is enough votes; thus, a cat's optimal proposal does not assign more than A to any cat other than themselves.

BUR – Cat Burglars

Lemma 1: *For $j \neq i$, $f_i(j) \leq A$.*

Proof: Inductively, cat i assigns a non-zero number of bundles to at most $\left\lfloor \frac{n-i}{2} \right\rfloor$ other cats. Cat $i-1$ could assign any bundles only to cats that receive 0 from i, since $n-(i-1)-\left(\left\lfloor \frac{n-1}{2} \right\rfloor + 1\right)$ is enough votes; thus, a cat's optimal proposal does not assign more than A to any cat other than themselves.

Lemma 2: *For any i, there are at most A cats j such that $a_j \nmid f_i(j)$. In other words, the second element of $\left(f_i(j), a_j\right)$ divides the first for all cats except of at most A.*

Proof: Once for some cat j $f_i(j) = 0$, their cost for all $f_{<i}$ will always be a multiple of its greed. There is at most A cats that have not yet been zeroed, since every payment increases the cost and Lemma 1 limits it by A.

BUR – Cat Burglars

Lemma 3: $G = \mathcal{O}(A \log A)$.

Proof: Recall each group is identified by $\left(f(j), a_j\right)$. From Lemma 2 we know that they can be represented as $\left(ka_j, a_j\right)$ except for at most A groups. Thus we have

$$G \leq A + \sum_{a_j} \sum_{k=1}^{a_j} \left(\left\lfloor \frac{a_j}{k} \right\rfloor + 1 \right) \leq 2A + A \sum_{k=1}^{A} \left(\left\lfloor \frac{1}{k} \right\rfloor \right) = \mathcal{O}(A \log A)$$

BUR – Cat Burglars

One cat get $\mathcal{O}(nG \log n)$ by using a different data structure.

We can keep dynamic segment trees that keep which consecutive subsequences of $[1, n]$ are in the tree. With that, the binary search can be performed on all trees in $(k, \_)$ "in lockstep", always going everywhere left or right depending on the sum of counts of cats to the left.

This is significantly faster, but was not required to get 10 points.

There's a slightly better analysis as well.

BUR – Cat Burglars

Let $d(x)$ be the number of divisors of x and :

$$D(x) = \max_{1 \le i \le x} d(i)$$

$D(64) = d(60) = 12$. Then we maintain $\mathcal{O}(A \log A)$ groups, but whenever doing splits for a fixed k we can look through the A special cats first and then only process $D(k)$ subtrees in $\log n$, giving $\mathcal{O}(n(A \log A + D(A) \log n))$.

The function $d(x)$ is strictly sublinear, so this is markedly faster.

Recall the plan

- Greedy and dynamic programming (DP)
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- Binary Search Trees
- **Number theory** ← *we are here*
- String algorithms (KMP, tries, suffix tables)
- Some problems can't* even be solved efficiently (NP-completeness)

Fast Exponentiation

How quickly can we compute x^n (usually modulo some m)?

Naively in $\mathcal{O}(n)$.

Fast Exponentiation

How quickly can we compute x^n (usually modulo some m)?

Naively in $\mathcal{O}(n)$.

$\mathcal{O}(\log n)$ using n's binary representation.

```
fn fastpow(x, n)
  res = 1
  while n > 0
    if x & 1
      res *= x
    x *= x
    n /= 2
  return res
```

```
fn fastmodpow(x, n, m)
res = 1
while n > 0
  if x & 1
    res = (res * x) % m
  x = (x % m)
  n /= 2
return res
```

Fast Matrix Exponentiation

This can be used to exponentiate matrices as well, e.g. $M^{13} = M^1 \cdot M^4 \cdot M^8$.

Fibonacci

How to compute the n-th Fibonacci number quickly?

$F_n = F_{n-1}F_{n-2}$

Take a matrix $\mathrm{FM} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Note that $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$.

Then

$$\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

So we can compute FM^n and take the top-left element.

DP as Matrix Multiplication

If we have dynamic programming where $\mathrm{DP}[n]$ is a linear combination of $\mathrm{DP}[n-1], \mathrm{DP}[n-2], \ldots, \mathrm{DP}[n-k]$ then we can construct a $k \times k$ matrix that will compute $\mathrm{DP}[n]$ similarly to the Fibonacci example.

Extended Euclidean Algorithm

Solve the congruence

$$ax - by \equiv \gcd(a, b)$$

```
fn eea(a, b)
  old_r, r = a, b // old_r is the gcd(a,b) at the end
  old_x, x = 1, 0 // old_x is the solution's x at the end
  old_y, y = 0, 1 // old_y is the solution's y at the end
  while r != 0
    q = old_r / r
    old_r, r = r, old_r - q * r
    old_x, x = x, old_x - q * x
    old_y, y = y, old_y - q * y
```

Extended Euclidean Algorithm – example

Dexter has n snacks arranged in a line. Starting from a-th snack, up until c-th, every k-th snack is exceptionally tasty. Moreover, starting from b-th, up until d-th, every l-th snack is exceptionally pretty. Dexter wants to eat only the exceptionally tasty and *pretty snack. Tell him at which snack to start and what are the gaps between each snack he should eat.*

```
5 1024 7
3 911 10
```

```
33 911 70
```

Extended Euclidean Algorithm – example

Find the intersection of $[a : c : k]$ and $[b : d : l]$ expressed as $[x : y : z]$.

We can ignore the ends ($y = \min(c, d)$).

The step is naturally the lowest common multiple of k and l, which we compute by $\frac{kl}{\gcd(k,l)}$.

The issue is in finding the first element.

We could advance two iterators from a and b until they meet, but if the numbers are large (e.g. 10^{18}) with a relatively small k, l then this takes forever.

Extended Euclidean Algorithm – example

W.l.o.g. assume $a \leq b$. The difference of the first element of $[a :: k]$ greater than b and b is:

$$\Delta \equiv k - (b - a) \bmod k$$

To overcome this delta we need to jump some number of times – x – by l, so that we will still fall into $[a :: k]$ again. In other words:

$$\Delta + lx \equiv 0 \bmod k \text{ or,}$$

$$lx \equiv -\Delta \bmod k$$

This is a linear congruence that can be solved with EEA.

Extended Euclidean Algorithm – example

Solve $ax \equiv b \bmod m$. If $\gcd(a, m)$ does not divide $b \bmod m$ then no solutions.

Otherwise, find x, y s.t. $ax - my = \gcd(a, m)$ and as the result take

$$x\frac{b}{\gcd(a, m)} \bmod \frac{m}{\gcd(a, m)}$$

There may be multiple solutions if $\gcd(a, m) > 1$, but this gives the smallest one.

Fundamental Theorem of Arithmetic

Theorem (Fundamental Theorem of Arithmetic): *any integer larger than* 1 *can be uniquely factored into a product of powers of prime numbers*

$$q = p_1^{n_1} p_2^{n_2} p_3^{n_3} \dots p_k^{n_k}$$

Divisors

As mentioned, $d(n)$ (number of divisors of n) satisfies $o(n^{\varepsilon})$ for any $\varepsilon > 0$.

More precisely, $\log d(n) = O\left(\frac{\log n}{\log \log n}\right)$.

Prime Number Theorem

The number of primes lower or equal to n is denoted by $\pi(n)$.

Theorem (Prime Number Theorem):

$$\pi(n) = \Theta\left(\frac{n}{\log n}\right)$$

The n-th prime is denoted by p_n.

Dusart's Inequality:

$$p(n) < n \ln n + n \ln \ln n \qquad (\text{for } n \geq 6).$$

Finding primes – Sieve of Eratosthenes

```
is_prime = array initially set to true
is_prime[1] = false
for i in [2..sqrt(n)]
  if is_prime[i]
    j = 2 * i
    while j <= n
      is_prime[j] = false
      j += i
```

Sieve of Eratosthenes – complexity

Complexity: $\mathcal{O}(n \log \log n)$.

We perform $\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} \ldots$ operations.

The sequence $\sum_{n=1} \frac{1}{p_n}$ diverges to $\Theta(\log \log n)$.[1]

The Sieve can be easily modified to store all prime divisors as well.

[1] The proof is tedious, but in https://math.stackexchange.com/questions/4362120 there is a rather fundamental proof just using a bunch of inequalities and algebra.

Primality Tests

How to *test* if a number is prime?

Naively: check all divisors in $\mathcal{O}(\sqrt{n})$.

We have deterministic algorithms in various polylogarithmic complexities, most notably **AKS** in $\tilde{\mathcal{O}}\left(\log(n)^6\right)$[2].

In practice, there are *probabilistic* tests that are much faster.

[2] "Soft-O" notation that hides logarithmic factors, e.g. $\mathcal{O}(f(n) + \log f(n)) = \tilde{\mathcal{O}}(f(n))$.

Fermat's Little Theorem

Theorem (Fermat's little theorem): *If* $a, p \in \mathbb{N}$ *and* p *is prime then*

$$a^p \equiv a \bmod p$$

Equivalently:

$$a^{p-1} \equiv 1 \bmod p$$

Fermat's Little Theorem

Theorem (Fermat's little theorem): *If $a, p \in \mathbb{N}$ and p is prime then*

$$a^p \equiv a \bmod p$$

Equivalently:

$$a^{p-1} \equiv 1 \bmod p$$

Proof (sketch): $G = \{1, 2, ..., p-1\}$ is a group with multiplication modulo p. If k is the order of a in G (smallest integer such that $a^k \equiv 1 \bmod p$) then $\left\{k \in \mathbb{N}, a^k \bmod p\right\}$ is a subgroup of order k. Therefore $k \mid p-1$ and thus

$$a^{p-1} \equiv a^{kx} \equiv \left(a^k\right)^x \equiv 1^x \equiv 1 \bmod p$$

Primality Test – Rabin-Miller

The idea is to check a certain property that:

1. Definitely holds for all prime numbers; and
2. doesn't hold for a lot of composite numbers.

One check of the property will have a high probability of a miss, *but* running multiple rounds will exponentially decrease the risk.

Primality Test – Rabin-Miller

A simple idea is to check Fermat's little theorem – if n is prime then for any number a we have $a^{n-1} \equiv 1 \bmod n$.

However, there are composite numbers that satisfy Fermat's little theorem (called Carmichael numbers).

Let us take $n-1$ as $2^s d$ where $s > 0$ and d is an odd integer.[3] Let's take some a. If n is prime then:

1. $a^d \equiv 1 \bmod n$; or
2. $a^{2^r d} \equiv -1 \bmod n$ for some $0 \leq r < s$.

[3] If n is even and not equal to 2 then it's obviously not prime. So n is odd and $n-1$ is even.

Primality Test – Rabin-Miller

```
fn mr_test(n, a, d, s)
  x = fastmodpow(a, d, n)
  if x == 1 || x == n - 1
    return true
  for [0..s]
    x = (x * x) % n
    if x == n - 1
      return true
  return false
```

```
fn miller_rabin(n, k)
  if n < 4 { return n == 2 or n == 3 }
  (s, d) = factor(n - 1) // n - 1 = 2^s * d
  for [0..k]
    a = select_base(n)
    if !miller_rabin_round(n, a, d, s)
      return false
  return true
```

And `select_base` should select a random number in $[2, n-2]$.

Primality Test – Rabin-Miller

It can be proven that if n is *not* prime then for at most $\frac{1}{4}$ bases in $[2, n-2]$ a single round of Miller-Rabin passes.

Thus, if we select a randomly, we get an error rate of $\left(\frac{1}{4}\right)^k$ for k rounds.

For 5 rounds that's 0.001% chance.

For 15 rounds that's less than 10^{-9}.

For 30 rounds – less than 10^{-18}.

Complexity is $\mathcal{O}(k \log n)$.

Primality Test – Rabin-Miller practical opts

A common optimisation is to precompute some primes and test if any of them divide n before running Rabin-Miller. This gives massive improvements in practice.

Moreover, for numbers below 2^{32} it's enough to check $2, 3, 5, 7$ as bases.

For 2^{64} it's the first 12 prime numbers.

Euler's φ function

Euler's φ function, called *Euler's totient function*, counts the number of relatively prime numbers, i.e.

$$\varphi(n) := |\{x \in \mathbb{N} \mid \gcd(n, x) = 1\}|$$

For example, $\varphi(12) = 7$.

Some properties:

- $\varphi(1) = 1$;
- for prime p $\varphi(p) = p - 1$;
- for coprime n, m $\varphi(nm) = \varphi(n)\varphi(m)$ (proof by CRT, see later).

Euler's φ function

To calculate n we need to know the prime numbers below n. Then:

$$\varphi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right)$$

Equivalently and algorithmically more usefully:

$$\varphi(n) = p_1^{n_1 - 1}(p_1 - 1)p_2^{n_2 - 1}(p_2 - 1)\ldots p_k^{n_k - 1}(p_k - 1)$$

The proofs are too time-consuming for inclusion in this course.

Modular inverse

Modular arithmetic is simple for addition and multiplication – just do `% m` everywhere.

For subtraction the sign is important. For $a - b \bmod m$ we write `(a - b + m) % m`.

For division it gets much more complicated.

Modular inverse

Modular arithmetic is simple for addition and multiplication – just do `% m` everywhere.

For subtraction the sign is important. For $a - b \bmod m$ we write `(a - b + m) % m`.

For division it gets much more complicated.

Division is multiplication by the inverse.

Modular multiplicative inverse

The *modular multiplicative inverse* of a is an x such that

$$ax \equiv 1 \bmod m$$

Modular multiplicative inverse exists if and only if a and m are coprime.

One way to compute this is the Extended Euclidean Algorithm. We're solving

$$ax + my = 1$$

since we can rewrite that to

$$ax - 1 = (-y)m$$

and thus $ax \equiv 1 \bmod m$.

Euler's theorem

Theorem (Euler's): *For coprime a, m:*

$$a^{\varphi(m)} \equiv 1 \bmod m$$

This is a generalisation of Fermat's little theorem.

The algebraic proof is actually identical.

This can also be used for computing the multiplicative inverse – it's equal to $a^{\varphi(m)-1} \bmod m$.

Linear congruence

A linear congruence is an equation in the shape:

$$a \cdot x \equiv b \bmod m$$

which we want to solve for x.

Linear congruence

A linear congruence is an equation in the shape:

$$a \cdot x \equiv b \bmod m$$

which we want to solve for x.

If a and m are coprime then we can find the inverse of a and multiply both sides:

$$x \equiv b \cdot a^{-1} \bmod m$$

Linear congruence

A linear congruence is an equation in the shape:

$$a \cdot x \equiv b \bmod m$$

which we want to solve for x.

If a and m are coprime then we can find the inverse of a and multiply both sides:

$$x \equiv b \cdot a^{-1} \bmod m$$

Otherwise, if b is not divisible by $\gcd(a, m)$ then there is no solution.

Linear congruence

$$a \cdot x \equiv b \bmod n$$

If b is divisible by $d = \gcd(a, m)$ then we can instead solve the equation:

$$\frac{a}{d} \cdot x \equiv \frac{b}{d} \bmod \frac{m}{d}$$

This is the *smallest* solution, but there are d solutions in total: for each $i \in [0..g-1]$

$$x_i \equiv \left(x_0 + i \cdot \frac{m}{d}\right) \bmod m$$

Linear congruence

$$a \cdot x \equiv b \bmod n$$

This can be also solved with EEA, since we can write:

$$ax + mk = b$$

and solve for x.

Sun Zi's Theorem

Sun Zi's Theorem (commonly called **the Chinese Remainder Theorem** (CRT)[4]), says how to solve specific *systems* of congruences.

$$\begin{cases} x \equiv a_1 \bmod m_1 \\ x \equiv a_2 \bmod m_2 \\ \qquad \vdots \\ x \equiv a_k \bmod m_k \end{cases}$$

Where $m_1, \ldots, m_k$ are pairwise coprime. Sun Zi's theorem states that such a system has *exactly one* solution modulo m and how to find them.

[4] It's a much more common name, but I don't like it and I make the slides, so… Also come on, "Master Sun's Theorem" sounds so much cooler.

Sun Zi's Theorem

Let's tackle a system of two congruences first.

$$\begin{cases} x \equiv a_1 \bmod m_1 \\ x \equiv a_2 \bmod m_2 \end{cases}$$

Remember the EEA example? The idea here is similar, find the solution to

$$m_1 x + m_2 y = 1$$

Then if we take

$$x = a_1 y m_2 + a_2 x m_1$$

We can verify this x satisfies both congruences.

Sun Zi's Theorem

We can solve the entire problem by induction now.

$$\begin{cases} x \equiv a_1 \bmod m_1 \\ x \equiv a_2 \bmod m_2 \\ \qquad \vdots \\ x \equiv a_k \bmod m_k \end{cases}$$

solve the first two obtaining a candidate x_0 and replace them with

$$\begin{cases} x \equiv x_0 \bmod m_1 m_2 \\ \qquad \vdots \\ x \equiv a_k \bmod m_k \end{cases}$$

Sun Zi's Theorem

The only important thing to show is that $m_1 m_2$ is coprime with all other m_i, but that's trivial.

Assuming all results fit in registers this will work in $\mathcal{O}(k \log M)$ where $M = m_1 m_2 \ldots m_n$.

If working with big numbers, it pays off to solve congruences in pairs, i.e. first and second, third and fourth, etc., keeping the products of moduli small.

At the end we get a system with $\frac{k}{2}$ equations and reapply the algorithm.

Sun Zi's Theorem

CRT can be generalised to non-coprime moduli.

Assume we have

$$\begin{cases} x \equiv a \bmod m \\ x \equiv b \bmod n \end{cases}$$

If $a \equiv b \bmod \gcd(n, m)$ then there is a unique solution, otherwise there are none.

If we solve (with EEA)

$$mx + ny = \gcd(n, m)$$

then the solution is $x = \frac{ayn+bxm}{\gcd(n,m)}$.

Discrete logarithm

Solution for

$$a^x \equiv b \bmod m$$

Solution doesn't always exist, e.g. $2^x \equiv 3 \bmod 7$ has no solutions.

There is no easy-to-check condition for existence.

This is one of those important hard problems, where we don't know a relatively fast solution, and thus it's useful for cryptography.

Best algorithm runs in $\mathcal{O}(\sqrt{m})$.

Discrete logarithm – baby-step giant-step

First let's assume a and m are coprime, we will lift this restriction at the end.

The idea is to select a candidate x and split $x = np - q$ (we will explain the best choice for n later).

$$a^{np-q} \equiv b \bmod m$$

Because a and m are relatively prime we can do this:

$$a^{np} \equiv ba^q \bmod m$$

We will now directly compute left-hand-side values for *all* p and right-hand-side values for *all* q. Once we do that we can find the values at which they two match (e.g. by sorting one array and binary-searching or with a hashmap).

Discrete logarithm – baby-step giant-step

The important thing here is that any number $x \in [0, m-1]$ can be represented with p, q and $p \in \left[1, \left\lceil \frac{m}{n} \right\rceil\right]$ while $q \in [0, n]$.

The name of the algorithm comes from the fact that increasing p by one increases x drastically (by n) – giant step – while increasing q by one decreases it by just 1 – baby step.

Discrete logarithm – complexity

For fixed p, q a^{np} can be calculated in $\mathcal{O}(\log m)$, as well as ba^q.

To compute the left-hand-side for all p we use $\mathcal{O}\left(\frac{m}{n} \log m\right)$ time.

For the right-hand-side we use $\mathcal{O}(n \log m)$.

The sort+binsearch or lookups are negligible.

Together we get $\mathcal{O}\left(\frac{m}{n} \log m + n \log m\right) = \mathcal{O}\left(\left(\frac{m}{n} + n\right)\right) \log m)$.

If we select $n = \sqrt{m}$ we get the best complexity – $\mathcal{O}\left(\sqrt{m} \log n\right)$.

Note that with some tricks we can get rid of exponentiation directly – when computing all LHS/RHS values in a loop we can just keep a variable for the current power of a, getting rid of the log factor for $\mathcal{O}\left(\sqrt{m}\right)$.

Discrete logarithm – generalisation

When a and m are not coprime then b has to be divisible by $d = \gcd(a, m)$, otherwise there are no solutions.

Otherwise, factor all variables by d. Say $a = da'$, $b = db'$, $m = dm'$.

$$
\begin{aligned}
a^x &\equiv b \mod m \\
(da')a^{x-1} &\equiv db' \mod dm' \\
a'a^{x-1} &\equiv b' \mod m'
\end{aligned}
$$

and $a'a$ is coprime with m'. We can extend our algorithm to work for arbitrary equations of the form

$$ka^x \equiv \mod m$$

Discrete root – generators

Find x such that

$$x^k \equiv a \bmod m$$

The key concept here is a *generator* in the group of multiplication modulo m.

A number g is a generator in $(\mathbb{Z}/m\mathbb{Z})^\times$ if and only if for any integer a coprime with m there exists a power k such that

$$g^k \equiv a \bmod m$$

Intuitively, a generator can be used to "generate" any number coprime with m by successive multiplication. In a sense, all such numbers are represented by g.

Discrete root – using generators

Assume for a second we have found a generator modulo m. Then the discrete root problem can be restated as:

$$(g^y)^k \equiv a \bmod n$$

and we're looking for $x \equiv g^y \bmod n$. But this is the same as:

$$\left(g^k\right)^y \equiv a \bmod n$$

… which is a discrete log problem!

Mateusz Gienieczko

Discrete root – all solutions

That gives us one solution, but there might be more.

If we have solved $x_0 \equiv g^{y_0} \bmod n$ then for any $l \in \mathbb{Z}$

$$x^k \equiv g^{y_0 k + l\varphi(n)} \equiv a \bmod n$$

$$x \equiv g^{y_0 + \frac{l\varphi(n)}{k}} \qquad \bmod n$$

For this to make sense the fraction must be integral, so the numerator $l\varphi(n)$ has to be divisible by $\operatorname{lcm}(k, \varphi(n))$. This gives us all results by the formula (for $i \in \mathbb{Z}$)

$$x = g^{y_0 + i\left(\frac{\varphi(n)}{\gcd(k,\varphi(n))}\right)}$$

Discrete root – finding generators

A naive solution of course is to check all numbers in $[1, n-1]$ and see if they're the generator by checking all its powers (they have to be different).

We need a few results to get a better solution.

Discrete root – finding generators

A generator in $(\mathbb{Z}/m\mathbb{Z})^\times$ exists if and only if:

- $m \in \{1, 2, 4\}$,
- $m = p^k$ for an odd prime p and $k \in \mathbb{N}^+$,
- $m = 2p^k$ for an odd prime p and $k \in \mathbb{N}^+$.

This is a fundamental result in algebra proven by Gauss.

Discrete root – Euler strikes back

It can also be proven that iff g is a generator modulo m then the smallest k for which

$$g^k \equiv 1 \bmod m$$

is equal to $\varphi(m)$.

Discrete root – finding generators

Lemma: *it is enough to check* $\frac{\varphi(m)}{p_i}$ *for all prime divisors* p_i *of* $\varphi(m)$.

Using this lemma we get an algorithm:

- find $\varphi(m)$ and prime factors,
- iterate through $[1, n-1]$ and:
  - for each p_i compute $g^{\frac{\varphi(m)}{p_i}}$
  - if all values are different from 1, g is a generator.

If we precompute prime factors this takes $\mathcal{O}(\text{Ans} \cdot \log \varphi(m) \cdot \log m)$.

The answer is small in practice.[5]

---

[5] It's proven to be $\mathcal{O}(\log^6 m)$ under the generalised Riemann hypothesis.

Discrete root – finding generators

Lemma: *it is enough to check* $\frac{\varphi(m)}{p_i}$ *for all prime divisors* p_i *of* $\varphi(m)$.

Proof: Because of group properties, we definitely only need to check all divisors of $\varphi(m)$.

Let d be any divisor of $\varphi(m)$. Then there exists some j such that there exists a k: $dk = \frac{\varphi(m)}{p_j}$. If g is a generator then:

$$g^{\frac{\varphi(m)}{p_j}} \equiv g^{dk} \equiv \left(g^d\right)^k \equiv 1^k \equiv 1 \bmod m$$

So checking each $\frac{\varphi(m)}{p_j}$ checks all divisors indirectly.

Mateusz Gienieczko

Discrete Fourier Transform

The Fourier Transform is a scary thing that does stuff to continuous functions using integration and expresses it in terms of sinuses or something, I don't know, I'm not a mathematician.

Anyway, the *Discrete* Fourier Transform makes sense.

Given a polynomial $p(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}$ it gives its values at some *magical* n values such that there exists an inverse function of DFT recovering the polynomial.

$$\mathrm{DFT}^{-1}(\mathrm{DFT}(p)) = p$$

Discrete Fourier Transform – applications

If we can compute DFT and DFT^{-1} efficiently, then this is great – performing operations on values is much easier than on polynomials.

For example, multiplying polynomials can just be done on values. If we have polynomials p, q we can do:

$$\mathrm{DFT}(p) = (y_0, y_1, ..., y_{n-1}), \mathrm{DFT}(q) = (z_0, z_1, ..., z_{n-1})$$

and compute

$$p \cdot q = \mathrm{DFT}^{-1}((y_0 \cdot z_0, y_1 \cdot z_1, ..., y_{n-1} \cdot z_{n-1}))$$

Discrete Fourier Transform – applications

Big numbers (i.e. outside of the range of CPU registers, well above 2^{64}) can be interpreted as the value of a polynomial p at 10 where the coefficients are subsequent digits of the number.

FFT can be used to implement fast multiplication of bignums.

After multiplication a rather straightforward normalisation (carry-propagation) is needed.

Discrete Fourier Transform – details

To understand the algorithm we need a few more details on DFT.

The *magical* points at which we compute the polynomial are *n-th roots of unity*, denoted by w_n^k for $k \in [0..n-1]$.

These are complex numbers so I'm not even going to try to explain what this means mathematically. For us it suffices to treat this algebraically - we have a polynomial and some value w_n, and by applying DFT we obtain

$$p(w_n) = \left(p\left(w_n^0\right), p\left(w_n^1\right), \ldots, p\left(w_n^{n-1}\right)\right) = (y_0, y_1, \ldots, y_{n-1})$$

and there are two more identities: $w_n^n = 1$ and $w_n^{\frac{n}{2}} = -1$.

Fast Fourier Transform

The core idea is to apply divide-and-conquer. We first extend the polynomial with zero coefficients such that n is a power of two. Then, divide the polynomial coefficients into two vectors of size $\frac{n}{2}$, compute DFT recursively, and combine the results.

More specifically, we divide $p(x)$ into even and odd coefficients

$$\begin{cases} p_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{\frac{n}{2}-1} \\ p_1 = a_1 x^1 + a_3 x^1 + \ldots + a_{n-1} x^{\frac{n}{2}-1} \end{cases}$$

Then $p(x) = p_0\left(x^2\right) + x p_1\left(x^2\right)$.

If we can combine the results in linear time we get $\mathcal{O}(n \log n)$.

Fast Fourier Transform

From recursion we get two vectors of length $\frac{n}{2} - 1$.

$$\begin{cases} \mathrm{DFT}(p_0) = \left(y_{0,0}, y_{0,1}, \ldots, y_{0,\frac{n}{2}-1}\right) \\ \mathrm{DFT}(p_1) = \left(y_{1,0}, y_{1,1}, \ldots, y_{1,\frac{n}{2}-1}\right) \end{cases}$$

The first $\frac{n}{2}$ values in the combined result can be computed directly:

$$y_k = y_{0,k} + w_n^k y_{1,k}$$

Fast Fourier Transform

The second half of values needs a different equation:

$$
\begin{aligned}
y_{k+\frac{n}{2}} &= p\left(w_n^{k+\frac{n}{2}}\right) \\
&= p_0\left(w_n^{2k+n}\right) + w_n^{k+\frac{n}{2}} p_1\left(w_n^{2k+n}\right) \\
&= p_0\left(w_n^{2k} w_n^{n}\right) + w_n^{k} w_n^{\frac{n}{2}} p_1\left(w_n^{2k} w_n^{n}\right) \\
&= p_0\left(w_n^{2k} - w_n^{k} p_1\left(w_n^{2k}\right)\right) \\
&= y_{0,k} - w_n^{k} y_{1,k}
\end{aligned}
$$

Fast Fourier Transform

In total we get:

$$\begin{cases} y_k = y_{0,k} + w_n^k y_{1,k} & \text{for } k \in \left[0..\frac{n}{2} - 1\right] \\ y_k = y_{0,k-\frac{n}{2}} - w_n^{k-\frac{n}{2}} y_{1,k-\frac{n}{2}} & \text{for } k \in \left[\frac{n}{2}..n\right] \end{cases}$$

Fast Fourier Transform – implementation

We're using magical complex numbers in equations.

FFT is unfortunately a numerical algorithm. In C++ it's easiest to use the standard `std::complex<double>` type. Rust has no standard complex type.

The root of unity w_n is given by the complex number $\cos(\alpha) + \sin(\alpha)i$ for $\alpha = \frac{2\pi}{n}$.

Fast Fourier Transform – inverse

To invert DFT we need to interpolate the polynomial along the n points.

The DFT can be written in matrix form as a ~~Voldemort~~ Vandermonde matrix with w_n^i as the entries. Inverting DFT is then multiplying by the inverse of this matrix.

From the properties of a Vandermonde matrix and roots of unity we get a formula:

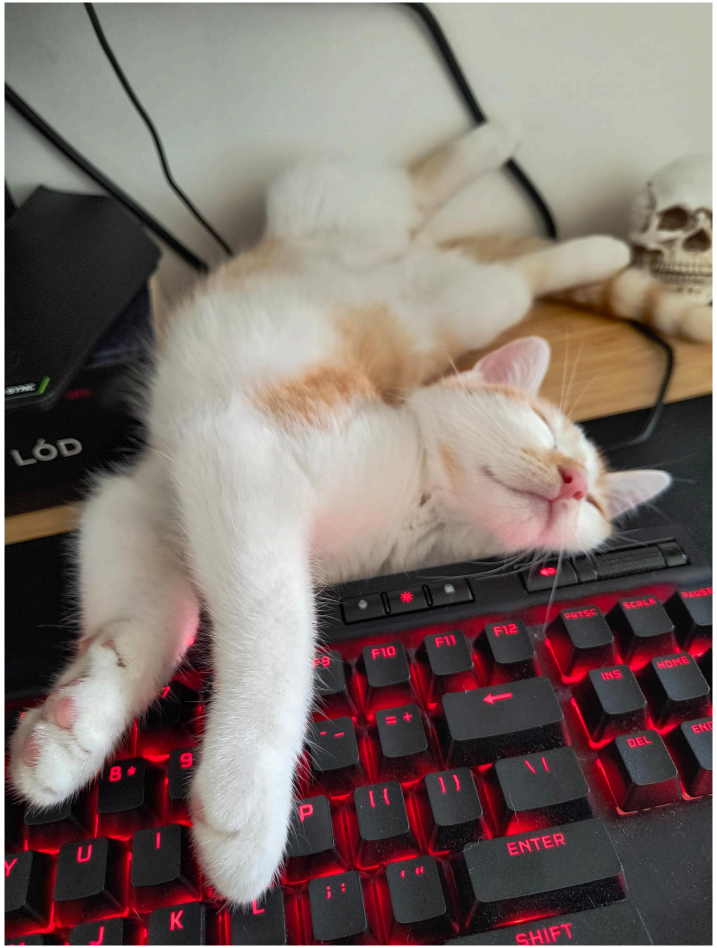
$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

But this is the same formula as for DFT, only k is negated and we divide by n.

See you next week

TAB: 08.07.2025, 10:00 AM

Good luck!



See you next week

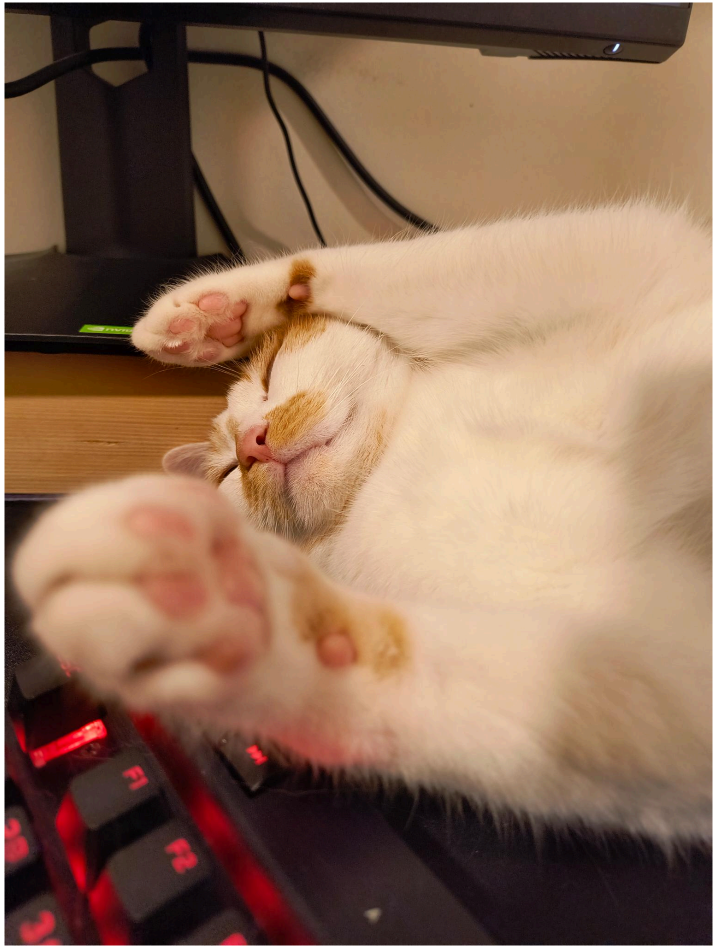
TAB: 08.07.2025, 10:00 AM

Good luck!

See you next week

TAB: 08.07.2025, 10:00 AM

Good luck!



See you next week

TAB: 08.07.2025, 10:00 AM

Good luck!