

Number Theory Algorithms in PyPy3 / Python 3

~ by Neel

Primes and Factors : Sieve of Eratosthenes and Variants

This section collects practical sieve templates used in competitive programming and mathematical computing, along with concise explanations, complexities, and usage notes. The implementations are in Python and rely on the integer square root function `isqrt` to set tight loop limits without floating point error.

What is here

- Classic boolean sieve for primality up to n with complexity $O(n \log \log n)$ and memory $O(n)$.
- Prime list extraction using the boolean sieve.
- Linear sieve computing the smallest prime factor (SPF) for every $x \leq n$ in $O(n)$, marking each composite exactly once.
- Fast factorization of an integer x using precomputed SPF in $O(\log x)$ time.
- Segmented sieve to enumerate primes in a range $[l, r]$ using base primes up to $\lfloor \sqrt{r} \rfloor$.

Why `math.isqrt`?

The function `math.isqrt(n)` returns $\lfloor \sqrt{n} \rfloor$ as an integer, avoiding floating point roundoff and extra conversions in loop bounds like $p \leq \sqrt{n}$. This keeps loops exact, fast, and safe for large n .

Reference implementation (Python 3)

Listing 1: Sieve templates and helpers

```
1 from math import isqrt
2
3 # Sieve templates: classic sieve, linear sieve (with SPF), factorization helper.
4
5 def sieve_bool(n: int): # time complexity O(n log log n)
6     """
7     Returns:
8         is_prime: list[bool] where is_prime[i] is True iff i is prime (0..n)
9     """
10    if n < 1:
11        return [False] * (n + 1)
12    is_prime = [True] * (n + 1)
13    is_prime[0] = False
14    if n >= 1:
15        is_prime[1] = False
16    limit = isqrt(n)
17    for p in range(2, limit + 1):
18        if is_prime[p]:
19            step = p
20            start = p * p
21            # Mark multiples of p starting from p^2
22            is_prime[start:n + 1:step] = [False] * ((n - start) // step + 1)
23    return is_prime
24
25
26 def sieve_primes(n: int):
27     """
28     Returns:
```

```

29     primes: list of primes <= n
30 """
31 is_prime = sieve_bool(n)
32 return [i for i, v in enumerate(is_prime) if v]
33
34
35 def linear_sieve(n: int): # time complexity O(n)
36 """
37 Returns:
38     primes: list of primes <= n (in order)
39     spf: smallest prime factor for each number 0..n (spf[1]=1)
40 Complexity: O(n)
41 ensures that every composite number is marked exactly once
42 """
43 spf = [0] * (n + 1)
44 primes = []
45 if n >= 1:
46     spf[1] = 1
47 for i in range(2, n + 1):
48     if spf[i] == 0:
49         spf[i] = i
50         primes.append(i)
51     for p in primes:
52         if p > spf[i] or p * i > n:
53             break
54         spf[p * i] = p
55 return primes, spf
56
57
58 def factorize_with_spf(x: int, spf: list[int]): # time complexity O(log x)
59 """
60 Factorizes x using precomputed smallest prime factors.
61 Returns list of (prime, exponent).
62 """
63 if x <= 1:
64     return []
65 res = []
66 while x > 1:
67     p = spf[x]
68     cnt = 0
69     while x % p == 0:
70         x //= p
71         cnt += 1
72     res.append((p, cnt))
73 return res
74
75
76 def segmented_sieve(l: int, r: int): # time complexity ~ size * log log sqrt(r)
77 """
78 Returns list of primes in the inclusive range [l, r] using a segmented sieve.
79 Handles ranges starting below 2.
80 """
81 if r < 2 or r < l:
82     return []
83 if l < 2:
84     l = 2
85 limit = isqrt(r)
86 base_primes = sieve_primes(limit)

```

```

87     size = r - l + 1
88     is_prime_seg = [True] * size
89     for p in base_primes:
90         start = max(p * p, ((l + p - 1) // p) * p)
91         for multiple in range(start, r + 1, p):
92             is_prime_seg[multiple - 1] = False
93     return [l + i for i, flag in enumerate(is_prime_seg) if flag]

```

Notes and usage

- Use `sieve_bool(n)` when boolean primality for all $0 \dots n$ is needed quickly with low constants.
- Use `sieve_primes(n)` when only the list of primes up to n is required.
- Use `linear_sieve(n)` when both the prime list and an SPF table are needed for downstream tasks like factorization or totients.
- For many independent factorizations $x \leq n$, precompute `spf` once with `linear_sieve(n)` and then call `factorize_with_spf(x, spf)` per query.
- For large ranges $[l, r]$ where r may not fit in a dense array, use `segmented_sieve(l, r)` to sieve only the window.
- For More Advanced and very less used algorithms (Miller Rabin , Pollard Rho , Lehmer's Prime Counting, Seive upto 1e9) check GitHub Repo

Wilson's Theorem

Statement

Let p be a positive integer with $p > 1$. Then p is prime if and only if

$$(p - 1)! \equiv -1 \pmod{p}.$$

Equivalently, p is prime if and only if $(p - 1)! + 1$ is divisible by p .

Corollaries and Applications

- Testing primality:** Wilson's theorem provides a theoretical primality test, though computing $(n - 1)!$ is impractical for large n .
- Factorial modulo prime:** For any prime p , we have

$$(p - 2)! \equiv 1 \pmod{p}$$

because $(p - 1)! = (p - 2)! \cdot (p - 1) \equiv -1 \pmod{p}$, so $(p - 2)! \equiv \frac{-1}{p-1} \equiv \frac{-1}{-1} \equiv 1 \pmod{p}$.

- Computing factorials with omissions:** For prime p and $1 \leq k < p$,

$$\prod_{\substack{i=1 \\ i \neq k}}^{p-1} i = \frac{(p-1)!}{k} \equiv \frac{-1}{k} \equiv -k^{-1} \pmod{p},$$

where k^{-1} is the modular inverse of k modulo p .

- For composite $n > 4$:** If n is composite and $n > 4$, then

$$(n - 1)! \equiv 0 \pmod{n}$$

because all prime factors of n are less than n and appear in the product $1 \cdot 2 \cdots (n - 1)$.

Multiplicative Functions

A function $f : \mathbb{N} \rightarrow \mathbb{C}$ is multiplicative if $f(1) = 1$ and $f(ab) = f(a)f(b)$ whenever $\gcd(a, b) = 1$; two key examples are the divisor-count $\tau(n) = \sum_{d|n} 1$ and divisor-sum $\sigma(n) = \sum_{d|n} d$.

Prime-power formulas

If $n = \prod_{i=1}^k p_i^{e_i}$, then

$$\tau(n) = \prod_{i=1}^k (e_i + 1), \quad \sigma(n) = \prod_{i=1}^k \frac{p_i^{e_i+1} - 1}{p_i - 1},$$

since $\tau(p^e) = e + 1$ and $\sigma(p^e) = 1 + p + \dots + p^e = \frac{p^{e+1}-1}{p-1}$, and multiplicativity extends these to all n .

Summatory identities

Using Dirichlet convolution, $\tau = 1 * 1$ and $\sigma = 1 * \text{id}$ yield

$$\sum_{k \leq n} \tau(k) = \sum_{d \leq n} \left\lfloor \frac{n}{d} \right\rfloor, \quad \sum_{k \leq n} \sigma(k) = \sum_{d \leq n} d \left\lfloor \frac{n}{d} \right\rfloor,$$

which motivates the standard floor-grouping trick to compute the sums in $O(\sqrt{n})$ segments.

Euler's totient and exponent reduction

Euler's totient $\varphi(n) = n \prod_{p|n} (1 - \frac{1}{p})$ is multiplicative and satisfies Euler's theorem: if $\gcd(a, m) = 1$ then $a^{\varphi(m)} \equiv 1 \pmod{m}$, allowing exponent reduction $e \mapsto e \bmod \varphi(m)$ in modular powers. When $\gcd(a, m) \neq 1$, a practical rule is to keep e if $e < \varphi(m)$ and otherwise use $e' = \varphi(m) + (e \bmod \varphi(m))$ to stay in the eventual period; replacing φ by the Carmichael function λ further tightens the period in the coprime case (optional refinement).

Euler's Theorem. If $\gcd(a, m) = 1$, then $a^{\varphi(m)} \equiv 1 \pmod{m}$, where $\varphi(m)$ is Euler's totient function counting integers in $[1, m]$ coprime to m . Equivalently, for any exponent e , one may reduce $a^e \bmod m$ by replacing e with $e \bmod \varphi(m)$ when $\gcd(a, m) = 1$. The totient admits the product formula $\varphi(n) = n \prod_{p|n} (1 - \frac{1}{p})$, which is multiplicative and enables fast computation alongside sieve-based prime factors. A refinement uses the Carmichael function $\lambda(m)$, the exponent of $(\mathbb{Z}/m\mathbb{Z})^\times$, giving $a^{\lambda(m)} \equiv 1 \pmod{m}$ for all $\gcd(a, m) = 1$, often yielding a smaller period than $\varphi(m)$.

Implementations (Python)

```
1 from math import gcd
2
3 # Number theory utilities: count and sum of divisors.
4 # Both tau (number_of_divisors) and sigma (sum_of_divisors) are multiplicative.
5
6 def number_of_divisors(n: int) -> int:
7     """
8         Returns the number of positive divisors of n.
9         """
10    if n == 1:
11        return 1
```

```

12
13     count = 1
14     # Factor 2
15     exp = 0
16     while n % 2 == 0:
17         n //=
18         exp += 1
19     if exp:
20         count *= (exp + 1)
21
22     p = 3
23     while p * p <= n:
24         if n % p == 0:
25             exp = 0
26             while n % p == 0:
27                 n //=
28                 exp += 1
29             count *= (exp + 1)
30         p += 2
31
32     if n > 1:
33         count *= 2 # remaining prime factor
34     return count
35
36
37 def sum_of_divisors(n: int) -> int:
38     """
39     Returns the sum of positive divisors of n.
40     """
41     if n == 1:
42         return 1
43
44     total = 1
45     # Factor 2
46     if n % 2 == 0:
47         exp = 0
48         pow2 = 1
49         sum2 = 1
50         while n % 2 == 0:
51             n //=
52             exp += 1
53             pow2 <= 1 # multiply by 2
54             sum2 += pow2
55         total *= sum2
56
57     p = 3
58     while p * p <= n:
59         if n % p == 0:
60             pow_p = 1
61             sum_p = 1
62             while n % p == 0:
63                 n //=
64                 pow_p *= p
65                 sum_p += pow_p
66             total *= sum_p
67         p += 2
68
69     if n > 1:

```

```

70         total *= (1 + n) # remaining prime factor
71     return total
72
73
74 def sum_of_div_MOD(n: int, MOD: int = 10**9 + 7) -> int:
75     """
76     Computes S(n) = sum_{k=1..n} sigma(k) modulo MOD via floor-grouping:
77     S(n) = sum_{d=1..n} d * floor(n/d).
78     """
79     ans = 0
80     l = 1
81     while l <= n:
82         t = n // l
83         r = n // t
84         cnt_sum = (l + r) * (r - l + 1) // 2
85         ans = (ans + (cnt_sum % MOD) * (t % MOD)) % MOD
86         l = r + 1
87     return ans
88
89
90 # Extended exponent reduction (non-coprime case):
91 # For general a, m, and e >= 0:
92 # if gcd(a, m) == 1: reduce e -> e % phi(m)
93 # else: if e < phi(m) keep e; else use e' = phi(m) + (e % phi(m))
94
95 def pow_mod_general(a: int, e: int, m: int) -> int:
96     """
97     Compute a^e mod m using Euler reduction even when a and m are not coprime.
98     """
99     if m == 1:
100        return 0
101    if e == 0:
102        return 1 % m
103    ph = phi(m)
104    if gcd(a, m) == 1:
105        exp = e % ph
106    else:
107        exp = e if e < ph else ph + (e % ph)
108    return pow(a % m, exp, m)
109
110
111 def euler_theorem(a: int, m: int) -> int:
112     """
113     Returns a^{phi(m)} mod m; equals 1 when gcd(a,m)=1 (and m>1).
114     """
115     if m == 1:
116        return 0
117     if gcd(a, m) != 1:
118         raise ValueError("a and m must be coprime")
119     return pow(a, phi(m), m)
120
121
122 def pow_mod_coprime(a: int, e: int, m: int) -> int:
123     """
124     Compute a^e mod m using Euler's theorem when gcd(a, m) = 1.
125     """
126     if m == 1:
127         return 0

```

```

128     if gcd(a, m) == 1:
129         e %= phi(m)
130     return pow(a % m, e, m)
131
132
133 def phi(n: int) -> int:
134     """
135     Euler's Totient: phi(n) = n * product_{p|n} (1 - 1/p).
136     Requires n >= 1.
137     """
138     result = n
139     x = n
140     p = 2
141     while p * p <= x:
142         if x % p == 0:
143             while x % p == 0:
144                 x //= p
145             result -= result // p
146         p += 1 if p == 2 else 2 # check 2 then only odds
147     if x > 1:
148         result -= result // x
149     return result
150
151
152 def phi_sieve(limit: int) -> list:
153     """
154     Computes phi(k) for all 1 <= k <= limit using a sieve.
155     """
156     phi = list(range(limit + 1))
157     for i in range(2, limit + 1):
158         if phi[i] == i: # i is prime
159             for j in range(i, limit + 1, i):
160                 phi[j] -= phi[j] // i
161     return phi

```

Complexity notes

The factorization-based $\tau(n)$ and $\sigma(n)$ implementations run in $O(\sqrt{n})$ time by trial division with small constant factors in practice. The summatory $\sum_{k \leq n} \sigma(k)$ via floor-grouping runs in $O(\sqrt{n})$ segments, and the totient sieve runs in $O(n \log \log n)$ time with $O(n)$ memory .

Euclidean Algorithm

The Euclidean algorithm computes $\gcd(a, b)$ by repeated division with remainder and runs in $O(\log \min(a, b))$, forming the backbone for many number theory routines [no-cite] .

Bézout's Identity

Theorem. For integers a, b with $d = \gcd(a, b)$, there exist integers x, y such that

$$ax + by = d.$$

Moreover, the set of all integer combinations $ax + by$ equals the set of multiples of d ; in particular, a and b are coprime if and only if there exist x, y with $ax + by = 1$ [no-cite] .

Chicken McNugget Theorem (Two Coins)

If m, n are coprime positive integers, then the largest integer not representable as $am + bn$ with $a, b \in \mathbb{Z}_{\geq 0}$ is

$$mn - m - n,$$

and exactly $\frac{(m-1)(n-1)}{2}$ positive integers are non-representable; when $\gcd(m, n) > 1$, infinitely many integers are non-representable [no-cite] .

Extended Euclidean Algorithm

Given a, b , the extended algorithm returns (g, x, y) with $g = \gcd(a, b)$ and $ax + by = g$; these coefficients realize Bézout combinations and can be used for modular inverses when $\gcd(a, m) = 1$ [no-cite] .

Listing 2: Extended Euclidean Algorithm: returns (g, x, y) with $ax + by = g = \gcd(a, b)$

```
1 from typing import Tuple
2
3 def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
4     if b == 0:
5         return a, 1, 0
6     g, x1, y1 = extended_gcd(b, a % b)
7     x = y1
8     y = x1 - (a // b) * y1
9     return g, x, y
```

Modular Arithmetic

Basics

For integers a, b, m with $m \geq 1$, write $a \equiv b \pmod{m}$ if $m \mid (a - b)$. A residue a has a multiplicative inverse modulo m if and only if $\gcd(a, m) = 1$, and the inverse is unique modulo m .

Linear Congruence Solver

Given $ax \equiv b \pmod{m}$, let $g = \gcd(a, m)$.

- Solutions exist iff $g \mid b$.
- If $g \nmid b$, there is no solution.
- Otherwise set $a' = \frac{a}{g}$, $b' = \frac{b}{g}$, $m' = \frac{m}{g}$ so that $\gcd(a', m') = 1$.
- The unique solution modulo m' is $x_0 \equiv (a')^{-1}b' \pmod{m'}$.
- All solutions modulo m are $x \equiv x_0 + k m'$ (\pmod{m}) for $k = 0, 1, \dots, g - 1$.

Listing 3: Linear congruence solver and modular inverse

```
1 from typing import List, Tuple
2 from math import gcd
3
4 def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
5     if b == 0:
6         return a, 1, 0
7     g, x1, y1 = extended_gcd(b, a % b)
8     x = y1
9     y = x1 - (a // b) * y1
10    return g, x, y
11
12 def mod_inverse(a: int, m: int) -> int:
13     a %= m
14     g, x, _ = extended_gcd(a, m)
15     if g != 1:
16         return -1 # inverse does not exist
17     return x % m
18
19 def solve_linear_congruence(a: int, b: int, m: int) -> List[int]:
20     a %= m
21     b %= m
22     g = gcd(a, m)
23     if b % g != 0:
24         return []
25     if g == 1:
26         inv = mod_inverse(a, m)
27         return [(inv * b) % m]
28     # Reduce and solve modulo m' = m/g
29     a_ = a // g
30     b_ = b // g
31     m_ = m // g
32     inv = mod_inverse(a_, m_)
33     x0 = (inv * b_) % m_
34     # Lift to all g solutions modulo m
```

```

35     sols = [(x0 + k * m_) % m for k in range(g)]
36     sols.sort()
37     return sols

```

Chinese Remainder Theorem (CRT)

Given $x \equiv a_i \pmod{m_i}$ for $i = 1, \dots, k$:

- If the moduli m_i are pairwise coprime, there exists a unique solution modulo $M = \prod_i m_i$.
- For general moduli, a solution exists iff for every pair (i, j) one has $a_i \equiv a_j \pmod{\gcd(m_i, m_j)}$.
- Merging two congruences $x \equiv a_1 \pmod{m_1}$, $x \equiv a_2 \pmod{m_2}$ reduces to solving

$$m_1 t \equiv a_2 - a_1 \pmod{m_2}, \quad \text{then } x = a_1 + m_1 t, \quad \text{modulus } \text{lcm}(m_1, m_2) = \frac{m_1 m_2}{\gcd(m_1, m_2)}.$$

Listing 4: CRT: general merge and list interface

```

1 from typing import List, Tuple
2
3 def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
4     """Return (g,x,y) with g = gcd(a,b) and ax + by = g."""
5     if b == 0:
6         return (a, 1, 0)
7     g, x1, y1 = extended_gcd(b, a % b)
8     return (g, y1, x1 - (a // b) * y1)
9
10 def mod_inverse(a: int, mod: int) -> int:
11     """Return inverse of a modulo mod; raises if gcd(a,mod)!=1."""
12     g, x, _ = extended_gcd(a, mod)
13     if g != 1:
14         raise ValueError("Inverse does not exist")
15     return x % mod
16
17 def chinese_remainder_pair(a1: int, m1: int, a2: int, m2: int) -> Tuple[int, int]:
18     """
19     Merge:
20         x  a1 (mod m1)
21         x  a2 (mod m2)
22     Return (x, lcm) or (-1, -1) if no solution.
23     """
24     if m1 <= 0 or m2 <= 0:
25         return (-1, -1)
26     g, p, q = extended_gcd(m1, m2)
27     diff = a2 - a1
28     if diff % g != 0:
29         return (-1, -1) # inconsistent
30     # Solve m1 * t  diff (mod m2)
31     m2_reduced = m2 // g
32     t = (diff // g) * (p % m2_reduced) % m2_reduced
33     x = a1 + m1 * t
34     mod = (m1 // g) * m2
35     x %= mod
36     return (x, mod)
37
38 def chinese_remainder(A: List[int], M: List[int]) -> Tuple[int, int]:

```

```

39 """
40 Generalized CRT over lists.
41 Return (x, L) with x the smallest non-negative solution modulo L,
42 or (-1, -1) if invalid input or no solution.
43 """
44 if len(A) != len(M) or not A:
45     return (-1, -1)
46 x = A[0] % M[0]
47 mod = M[0]
48 if mod <= 0:
49     return (-1, -1)
50 for a, m in zip(A[1:], M[1:]):
51     res_x, res_m = chinese_remainder_pair(x, mod, a % m, m)
52     if res_x == -1:
53         return (-1, -1)
54     x, mod = res_x, res_m
55 return (x, mod)
56
57 # Optional fast path for pairwise coprime moduli
58 def chinese_remainder_coprime(A: List[int], M: List[int]) -> Tuple[int, int]:
59 """
60 Assumes M are pairwise coprime.
61 Return (x, product) or (-1,-1) if invalid.
62 """
63 if len(A) != len(M) or not A:
64     return (-1, -1)
65 x = 0
66 prod = 1
67 for a, m in zip(A, M):
68     if m <= 0:
69         return (-1, -1)
70     inv = mod_inverse(prod % m, m) # gcd(prod, m)=1
71     x = (a - x) % m * inv % m * prod + x
72     prod *= m
73 return (x % prod, prod)

```

Linear Equations

Linear Diophantine in Two Variables

For integers a, b, c , the equation $ax + by = c$ has integer solutions iff $g = \gcd(a, b)$ divides c . If $g \mid c$, let (x_0, y_0) be any particular solution to $ax + by = c$; then all solutions are

$$x = x_0 + k \cdot \frac{b}{g}, \quad y = y_0 - k \cdot \frac{a}{g}, \quad k \in \mathbb{Z}.$$

When bounding boxes are given, $x \in [x_1, x_2]$, $y \in [y_1, y_2]$, the feasible k form an integer interval obtained by intersecting the inequalities from both coordinates.

Listing 5: Extended GCD, one solution, shifting, and counting within a box

```
1 import sys
2 from math import gcd
3
4 def extended_gcd(a, b):
5     """Return (g,x,y) with g = gcd(a,b) and a*x + b*y = g."""
6     if b == 0:
7         return a, 1, 0
8     g, x1, y1 = extended_gcd(b, a % b)
9     return g, y1, x1 - (a // b) * y1
10
11 def find_any_solution(a, b, c):
12     """
13     Find one integer solution (x0,y0) to a*x + b*y = c.
14     Returns (ok, x0, y0, g), where ok indicates existence and g = gcd(a,b).
15     """
16     if a == 0 and b == 0:
17         if c == 0:
18             return True, 0, 0, 0 # infinite grid (handle outside)
19         return False, 0, 0, 0
20     g, x, y = extended_gcd(abs(a), abs(b))
21     if c % g != 0:
22         return False, 0, 0, g
23     x *= c // g
24     y *= c // g
25     if a < 0: x = -x
26     if b < 0: y = -y
27     return True, x, y, g
28
29 def shift_solution(x, y, a, b, k):
30     """
31     Shift a particular solution by k steps along the solution line:
32     x' = x + k*(b/g), y' = y - k*(a/g) (caller uses divided coefficients).
33     """
34     return x + k * b, y - k * a
35
36 def count_solutions(a, b, c, x1, x2, y1, y2):
37     """
38     Count integer solutions (x,y) to a*x + b*y = c with x in [x1,x2], y in [y1,y2].
39     Returns the count (0 if no solution).
40     """
41     ok, x0, y0, g = find_any_solution(a, b, c)
42     if not ok:
43         return 0
44     if a == 0 and b == 0:
```

```

45     # Here c == 0, every pair in the box is a solution.
46     return (x2 - x1 + 1) * (y2 - y1 + 1)
47 if a == 0:
48     # b*y = c
49     if c % b: return 0
50     y = c // b
51     return (x2 - x1 + 1) if y1 <= y <= y2 else 0
52 if b == 0:
53     # a*x = c
54     if c % a: return 0
55     x = c // a
56     return (y2 - y1 + 1) if x1 <= x <= x2 else 0
57
58 a_div = a // g
59 b_div = b // g
60
61 # General solution: x = x0 + k*b_div, y = y0 - k*a_div
62 # Constrain x to [x1,x2] -> bounds on k
63 if b_div > 0:
64     kx_low = (x1 - x0 + b_div - 1) // b_div
65     kx_high = (x2 - x0) // b_div
66 else:
67     kx_low = (x0 - x2 + (-b_div) - 1) // (-b_div)
68     kx_high = (x0 - x1) // (-b_div)
69
70 # Constrain y to [y1,y2] -> y0 - k*a_div in [y1,y2]
71 if a_div > 0:
72     ky_low = (y0 - y2 + a_div - 1) // a_div
73     ky_high = (y0 - y1) // a_div
74 else:
75     ky_low = (y0 - y1 + (-a_div) - 1) // (-a_div)
76     ky_high = (y0 - y2) // (-a_div)
77
78 k_low = max(kx_low, ky_low)
79 k_high = min(kx_high, ky_high)
80 if k_low > k_high:
81     return 0
82 return k_high - k_low + 1

```

Counting Bounded Nonnegative Solutions

For nonnegative integers x_1, \dots, x_n with upper bounds $0 \leq x_i \leq f_i$ and sum $x_1 + \dots + x_n = s$, inclusion–exclusion yields

$$\#\{x \geq 0 : \sum x_i = s, x_i \leq f_i\} = \sum_{S \subseteq \{1, \dots, n\}} (-1)^{|S|} \binom{s - \sum_{i \in S} (f_i + 1) + n - 1}{n - 1},$$

interpreting $\binom{N}{r} = 0$ when $N < r$ or $N < 0$. Without upper bounds, the stars-and-bars count is $\binom{s+n-1}{n-1}$. The code below computes the bounded count modulo a prime MOD, precomputing $\text{inv}[1..R]$ where $R \geq n - 1$ to support $\binom{N}{r}$ with $r = n - 1$.

Listing 6: Bounded stars-and-bars via inclusion–exclusion modulo MOD

```

1 import sys
2 MOD = 10**9 + 7

```

```

4
5 def mod_pow(a, b):
6     res = 1
7     while b:
8         if b & 1:
9             res = res * a % MOD
10        a = a * a % MOD
11        b >>= 1
12    return res
13
14 # Precompute modular inverses up to needed r = n-1.
15 MAX_INV = 55 # ensure MAX_INV > n-1 in use
16 inv = [0] * MAX_INV
17 for i in range(1, MAX_INV):
18     inv[i] = mod_pow(i, MOD - 2)
19
20 def nCr_largeN(N, r):
21     """
22     Compute C(N, r) mod MOD for small r using multiplicative formula:
23     C(N,r) = N*(N-1)*...*(N-r+1) / r!
24     Assumes 0 <= r < MAX_INV.
25     """
26     if r < 0:
27         return 0
28     if r == 0:
29         return 1
30     if r > N - r:
31         r = N - r
32     if r < 0:
33         return 0
34     ans = 1
35     for i in range(r):
36         ans = ans * ((N - i) % MOD) % MOD
37         ans = ans * inv[i + 1] % MOD
38     return ans
39
40 # Input:
41 # n s
42 # f_1 f_2 ... f_n (upper bounds, each >= 0)
43 data = sys.stdin.read().strip().split()
44 it = iter(data)
45 n = int(next(it))
46 s = int(next(it))
47 f = [int(next(it)) for _ in range(n)]
48
49 # Inclusionexclusion over subsets S: subtract (f_{i+1}) for i in S
50 ans = 0
51 for mask in range(1 << n):
52     x = s
53     bits = 0
54     m = mask
55     i = 0
56     while m:
57         if m & 1:
58             x -= f[i] + 1
59             bits += 1
60         m >>= 1
61         i += 1

```

```
62     if x < 0:  
63         continue  
64     temp = nCr_largeN(x + n - 1, n - 1)  
65     if bits & 1:  
66         temp = -temp  
67     ans = (ans + temp) % MOD  
68  
69 print(ans % MOD)
```

Notes

Complexity: extended GCD and diophantine finding run in $O(\log \min(|a|, |b|))$; counting within a box reduces to constant-time arithmetic after one solution. The bounded stars-and-bars routine runs in $O(2^n)$ by inclusion–exclusion; use when n is small and s may be large.

Floor Sum

Definition and goal

Fix $\text{MOD} = 10^9 + 7$. The weighted floor sum is

$$S(n) = \sum_{i=1}^n i \lfloor \frac{n}{i} \rfloor \pmod{\text{MOD}},$$

which aggregates the weights i over the number of multiples of i not exceeding n , reduced modulo MOD .

Quotient grouping trick

The map $i \mapsto \lfloor \frac{n}{i} \rfloor$ takes only about $2\sqrt{n}$ distinct values on $1 \leq i \leq n$. If $q = \lfloor \frac{n}{i} \rfloor$ and $L = i$, then all indices with the same quotient form a contiguous interval $i \in [L, R]$ where $R = \lfloor \frac{n}{q} \rfloor$. Over this block, the contribution is constant in q , so

$$\sum_{i=L}^R i \lfloor \frac{n}{i} \rfloor = q \sum_{i=L}^R i = q \cdot \frac{(R - L + 1)(L + R)}{2}.$$

Advancing $i \leftarrow R + 1$ jumps between blocks, giving an $O(\sqrt{n})$ algorithm.

Key identities and relations

Two classical divisor-sum identities follow by swapping summations:

$$\sum_{i=1}^n \left\lfloor \frac{n}{i} \right\rfloor = \sum_{m=1}^n \tau(m), \quad \sum_{i=1}^n i \left\lfloor \frac{n}{i} \right\rfloor = \sum_{m=1}^n \sigma(m),$$

where $\tau(m) = \sum_{d|m} 1$ and $\sigma(m) = \sum_{d|m} d$. Moreover, the remainder-sum identity connects floor sums to modular remainders:

$$\sum_{k=1}^n (n \bmod k) = n^2 - \sum_{k=1}^n k \left\lfloor \frac{n}{k} \right\rfloor.$$

Use cases

- Fast evaluation of summatory divisor functions $\sum_{m \leq n} \tau(m)$ and $\sum_{m \leq n} \sigma(m)$ via the identities above and the $O(\sqrt{n})$ quotient grouping.
- Counting lattice points under a hyperbola $ij \leq n$ and related convolution sums arising in Dirichlet hyperbola method decompositions.
- Precomputing weighted counts of multiples for problems that need aggregated contributions by divisor or multiple classes (e.g., harmonic grouping in number-theoretic DP).
- Efficient evaluation of sums involving $\lfloor \frac{n}{i} \rfloor$ or its weighted variants in combinatorics and analytic number theory.

Implementation (Python)

Listing 7: Weighted floor sum in $O(\sqrt{n})$ via quotient grouping

```
1 MOD = 10**9 + 7
2
3 def weighted_floor_sum(n: int, mod: int = MOD) -> int:
4     """
5         Computes  $S = \sum_{i=1..n} i * \lfloor n / i \rfloor$  modulo mod.
6
7         Theory:
8             For fixed  $q = \lfloor n / i \rfloor$ , the indices  $i$  form a contiguous interval:
9                  $i$  in  $[L, R]$  where  $L = \text{current } i$ ,  $R = n // q$ .
10            Over that block,  $\lfloor n / i \rfloor = q$  is constant.
11            So contribution =  $q * \sum_{i=L}^R i = q * (R - L + 1)*(L + R)//2$ .
12            Advancing by blocks gives  $O(\sqrt{n})$  complexity because  $q$  changes
13            only about  $2\sqrt{n}$  times.
14        """
15        total = 0
16        i = 1
17        while i <= n:
18            q = n // i
19            r = n // q
20            # Sum i from L=i to R=r: arithmetic progression
21            block_sum = (r - i + 1) * (i + r) // 2
22            total = (total + (q % mod) * (block_sum % mod)) % mod
23            i = r + 1
24        return total
```

Complexity and notes

The loop advances across $O(\sqrt{n})$ blocks, so the time complexity is $O(\sqrt{n})$ and the space usage is $O(1)$. Use modular reduction on both the quotient q and the block sum to avoid overflow and to keep the result in $[0, \text{MOD} - 1]$.

Fibonacci Numbers

For $n \geq 0$, define $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$, yielding a sequence with rich algebraic identities and fast $O(\log n)$ computation methods

Properties

Fibonacci numbers (F_n) possess numerous notable identities and number-theoretic properties. [

1. Cassini's Identity: $F_{n-1}F_{n+1} - F_n^2 = (-1)^n$; a succinct proof follows from the determinant of the 2×2 matrix representation of the recurrence.
2. Addition Rule: $F_{n+k} = F_kF_{n+1} + F_{k-1}F_n$; special case $F_{2n} = F_n(F_{n+1} + F_{n-1})$.
3. Divisibility: For any positive integer k , F_{nk} is a multiple of F_n , and conversely F_m is a multiple of F_n iff m is a multiple of n .
4. GCD Identity: $\gcd(F_m, F_n) = F_{\gcd(m,n)}$, aligning the sequence structurally with the Euclidean algorithm.
5. Algorithmic Note (Lamé): Consecutive Fibonacci numbers form the worst-case inputs for the Euclidean algorithm, maximizing the number of division steps for given magnitudes.
6. Closed Form (Binet): $F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$ with $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2}$; for $n \geq 0$, rounding $\varphi^n/\sqrt{5}$ to the nearest integer yields F_n since $|\psi| < 1$.
7. Pisano Period: For modulus $m \geq 2$, $(F_n \bmod m)$ is purely periodic with least period $\pi(m)$ such that $(F_{\pi(m)}, F_{\pi(m)+1}) \equiv (0, 1) \pmod{m}$.

Implementations (Python)

Listing 8: Multiple ways to compute F_n

```
1 from math import sqrt
2
3 # Binet's closed-form (floating point rounding)
4 _phi = (1 + sqrt(5)) / 2
5 _psi = (1 - sqrt(5)) / 2
6 _sqrt5 = sqrt(5)
7
8 def fib_binet(n: int) -> int:
9     """Return F_n using Binet's formula (accurate for n >= 0 within integer rounding).
10     """
11     return int(round((_phi**n - _psi**n) / _sqrt5))
12
13 # Fast doubling (O(log n)), exact arithmetic
14 def fib_fast_doubling(n: int) -> int:
15     """Return F_n using fast doubling (O(log n)), exact for large n."""
16     if n < 0:
17         raise ValueError("n must be non-negative")
18     def _fd(k):
19         if k == 0:
20             return (0, 1)
21         a, b = _fd(k // 2)
22         c = a * (2 * b - a)
23         d = a * a + b * b
24         if k % 2 == 0:
```

```

24         return (c, d)
25     else:
26         return (d, c + d)
27 return _fd(n)[0]
28
29 # 2x2 matrix exponentiation ( $O(\log n)$ )
30 def fib_matrix(n: int) -> int:
31     """Return  $F_n$  using 2x2 matrix exponentiation ( $O(\log n)$ )."""
32     # Identity matrix and Fibonacci Q-matrix
33     res = (1, 0, 0, 1) # (a b; c d)
34     base = (1, 1, 1, 0)
35     k = n
36     while k:
37         if k & 1:
38             #  $res = res * base$ 
39             r00 = res[0] * base[0] + res[1] * base[2]
40             r01 = res[0] * base[1] + res[1] * base[3]
41             r10 = res[2] * base[0] + res[3] * base[2]
42             r11 = res[2] * base[1] + res[3] * base[3]
43             res = (r00, r01, r10, r11)
44         #  $base = base * base$ 
45         b00 = base[0] * base[0] + base[1] * base[2]
46         b01 = base[0] * base[1] + base[1] * base[3]
47         b10 = base[2] * base[0] + base[3] * base[2]
48         b11 = base[2] * base[1] + base[3] * base[3]
49         base = (b00, b01, b10, b11)
50         k >>= 1
51     return res[1]
52
53 # Modular golden-ratio field method (CSES-style), set MOD accordingly
54 MOD = 998244353
55
56 def modinv(x: int) -> int:
57     return pow(x, MOD - 2, MOD)
58
59 class Field:
60     # represents  $a + b\sqrt{5}$  over  $F_{MOD}$ 
61     __slots__ = ("a", "b")
62     def __init__(self, a=0, b=0):
63         self.a = a % MOD
64         self.b = b % MOD
65     def __add__(self, o):
66         return Field(self.a + o.a, self.b + o.b)
67     def __sub__(self, o):
68         return Field(self.a - o.a, self.b - o.b)
69     def __mul__(self, o):
70         if isinstance(o, Field):
71             #  $(a_1 + b_1\sqrt{5})(a_2 + b_2\sqrt{5}) = (a_1a_2 + 5b_1b_2) + (a_1b_2 + a_2b_1)\sqrt{5}$ 
72             return Field(self.a * o.a + 5 * self.b * o.b,
73                         self.a * o.b + self.b * o.a)
74         else:
75             return Field(self.a * o, self.b * o)
76     def inv(self):
77         #  $1 / (a + b\sqrt{5}) = (a - b\sqrt{5}) / (a^2 - 5b^2)$ 
78         denom = (self.a * self.a - 5 * self.b * self.b) % MOD
79         invd = modinv(denom)
80         return Field(self.a * invd, (-self.b) * invd)
81     def __truediv__(self, o):

```

```

82     if isinstance(o, Field):
83         return self * o.inv()
84     else:
85         invo = modinv(o)
86         return Field(self.a * invo, self.b * invo)
87 def pow(self, e: int):
88     r = Field(1, 0)
89     b = self
90     while e:
91         if e & 1:
92             r = r * b
93             b = b * b
94             e >>= 1
95     return r
96
97 # phi = (1 + sqrt(5)) / 2 => coefficients (1/2, 1/2)
98 inv2 = (MOD + 1) // 2
99 phi = Field(inv2, inv2)
100
101 def fib(n: int) -> int:
102     """Return F_n modulo MOD via field with sqrt(5)."""
103     if n == 0:
104         return 0
105     a = phi.pow(n)
106     b = (Field(1, 0) - phi).pow(n) # equals psi^n with psi = (1 - sqrt(5))/2
107     num = a - b # phi^n - psi^n
108     res = num / Field(0, 1) # divide by sqrt(5)
109     # result should be purely real
110     assert res.b % MOD == 0
111     return res.a % MOD

```

Notes

Fast doubling and 2×2 matrix exponentiation both run in $O(\log n)$ arithmetic operations and are robust for large n , while Binet's closed form is convenient with floating-point rounding for moderate n only.