整数に関しては以下の内容を扱います。

- 最大公約数とユークリッドの互除法
- 拡張ユークリッドの互除法
- 素数判定
  - ナイーブな実装
  - エラトステネスの篩
- 素因数分解
  - ナイーブな実装
  - SPF を用いた実装
- 冪乗: 繰り返し二乗法
- 逆元とフェルマーの小定理

I.　最大公約数とユークリッドの互除法

整数 $a, b(a > b)$ の最大公約数を求めるには、**ユークリッドの互除法**を使用します。

ユークリッドの互除法

$a, b(a > b)$ の最大公約数は a % b と b の最大公約数と等しい。

a, b の剰余を計算繰り返し計算していって b が 0 になった時の a が最大公約数となります。繰り返し同じ計算をするので、**再帰関数**を使って実装します。

コード 1　ユークリッドの互助法実装

```
def gcd(a: int, b: int) -> int:
  # 終了条件
  if b == 0:
      return a

  if a < b:
      a, b = b, a

  return gcd(b, a % b)
```

最大公約数をアルゴリズムを理解するだけではなくて、図形を使って直感的に理解しましょう。参考に記したけんちょんさんの最大公約数の記事を参考にすると、最大公約数は n 次元の立方体を考えたときにそれを均等に分割できる最大の立方体の辺の長さとして考えることができます。このように考えると、最大公約数が何を意味しているのかが直感的に理解できるかもしれません。

i.　問題と解説

1.　ABC 118 C - Monsters Battle Royale

問題からはすぐには気づきにくいですが、最大公約数を利用します。n 次元の立方体を考えたときにそれを均等に分ける最大の立方体の辺の長さを求める問題と同じです。配列 A のどの要素も gcd の倍数になっているので、gcd がこれ以上分割できない最小の辺の長さとなります。どんなに操作を加えても最小の辺の長さは変わりません。このように操作を繰り返しても変わらない数のことを**不変量**といいます。与えられた配列 A の要素同士の引き算から必ず gcd を生み出せるかという疑問が生まれるかもしれませんが、ユークリッドの互除法を使うことで、gcd を求めることができます。

コラム 再帰関数

おそらく初めて再帰関数を見た方は再帰関数の動きがわかりにくいと思います。再帰関数は自分自身を呼び出す関数です。そのため、再帰関数を理解するためには、関数が呼び出された時にどのような動きをするのかを理解する必要があります。関数が呼び出されると、スタックに関数が積まれていきます。最初に呼び出した gcd は一番下に、最後に呼び出した gcd はスタックの一番上に置かれます。一番上の関数戻り値がわかったら、連鎖的にそれより下の関数の値もわかります。そして今回の場合シグネチャからもわかる通り、int のデータ型の値を返す関数です。return する箇所で再び int の値を返す関数を呼び出しているので、スタックに積まれた関数が return するまで関数が呼び出され続けます。イコールが終了条件を満たすまで連なっていると考えるとわかりやすいかもしれません。

II.　拡張ユークリッドの互除法

ユークリッドの互除法では、最大公約数を求めることができますが、拡張ユークリッドの互除法を使うと、最大公約数だけでなく、$ax+by=\gcd(a,b)$ を満たす x,y を求めることができます。

$ax+by=gcd(a,b)$ の a,b に対してユークリッドの互除法を適用していきます。通常のユークリッドの互除法のように、$a\%b,b$ を計算していきますが、計算した結果それがれが次の一次不定方程式の係数となります。$bx_1+\{(a\%b)y_1\}=gcd(a,b)$ となります。ここで、

$$a\%b=a-(a//b)\times b$$

が成立することから、代入して

$$bx_1+ay_1-(a//b)\times by_1=gcd(a,b)$$
$$ay_1+b(x_1-(a//b)y_1)=gcd(a,b)$$

$ax + by = gcd(a, b)$ と a, b について係数比較をすることで、

$$x = y_1, y = x_1 - (a//b) \times y_1 \tag{1}$$

x_1, y_1 は $b, a\%b$ に対して再帰的に求めることができます。終了条件は $b = 0$ のときで、このときの x が求める x となります。

具体例を見ましょう。$a = 108, b = 56$ つまり $108x + 56y = gcd(108, 56)$ です。

$$108x + 56y = gcd(108, 56)$$
$$56x_1 + 52y_1 = gcd(56, 52)$$
$$52x_2 + 4y_2 = gcd(52, 4)$$
$$4x_3 + 0y_3 = gcd(4, 0)$$

$ax+by = gcd(a, b)$ の解 (x, y) が b と $a\%b$ を係数にした一次不定方程式の解 (x_1, y_1) から求められていることを確認してください。上から順に再帰的に (b, a%b) を求めています。$b = 0$ になったとき、$x_3 = 1, y_3 = 0$ になります。式 (1) より、$x_2 = y_3 = 0, y_2 = x_3 - (52//4) \times y_3 = 1$ となります。同様に、$x_1 = y_2 = 1, y_1 = x_2 - (56//54)y_2 = -1$

以上より、$x = y_1 = -1, y = x_1 - (108//56) \times y_1 = 2$ となります。

実装は以下のようになります。

コード 2　拡張ユークリッドの互助法実装

```python
def extended_gcd(a: int, b: int) -> tuple[int, int, int]:
    if b == 0:
        return a, 1, 0
    else:
        gcd, x1, y1 = extended_gcd(b, a % b)

        # 係数の更新
        x = y1
        y = x1 - (a // b) * y1

        return gcd, x, y
```

III.　素数判定

i.　ナイーブな実装

ナイーブな実装では以下の定理を利用して、自然数 n の素数判定を行います。

定理 1

自然数 n が $\sqrt{n}$ 以下の素数で割り切れないならば、n は素数である。

実装は以下のようになります。

コード 3　ナイーブな素数判定

```python
def is_prime(n: int) -> bool:
    # 忘れないように注意
    if n <= 1:
        return False
    i = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += 1

    return True
```

ナイーブな実装では、1 個の自然数が素数か判定するには $O(\sqrt{n})$ でできますが、n 個の数を判定するとなったら $O(n\sqrt{n})$ かかり少し遅いです。

ii.　エラトステネスの篩

多くの自然数の素数判定を高速に行う方法の 1 つに**エラトステネスの篩**があります。求める自然数の最大値を n とすると、1 から n までの自然数の素数判定はエラトステネスの篩を使うと、$O(n \log \log n)$ の計算量で実行可能です。

エラトステネスのの篩では、判定したい自然数 n までのテーブルを作成し、2 から順に以下の操作を $\sqrt{n}$ まで繰り返します。

1. $i = 2$ からスタート
2. i がテーブルでバツがついていない場合、i を除いた i のすべての倍数を素数ではないとする
3. $i = i + 1$ ($i = \sqrt{n}$ まで 2 へ)

例として $n = 100$ の場合を見てみます。

1	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~

i = 2 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 3 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 5 の場合

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

i = 7 の場合

1 を除いて残った数が素数です。

iii.　エラトステネスの篩の実装

コード 4　エラトステネスの篩の実装

```
def is_prime(n: int) -> list[bool]:
    is_prime_list = [True] * (n + 1)
    is_prime_list[0] = is_prime_list[1] = False
    i = 2
    while i * i <= n:
        if is_prime_list[i]:
            for j in range(i * i,n + 1,i):
                is_prime_list[j] = False
        i += 1

    return is_prime_list
```

IV.　素因数分解

i.　ナイーブな実装

ナイーブな素因数分解では、与えられた自然数 n を 2 から $\sqrt{n}$ まで順に割っていき、1 になれば終了します。割り切れた数を素因数としてリストに追加していきます。$\sqrt{n}$ まで続けて n が 1 でない場合は、その数も素因数としてリストに追加します。

コード 5　ナイーブな素因数分解の実装の実装

```python
def enumerate_prime_factors(n: int) -> list[int]:
    primes = []
    i = 2
    while i * i <= n:
        if n % i == 0:
            primes.append(i)
            n //= i
            while n % i == 0:
                primes.append(i)
                n //= i
        i += 1

    # 残ったnが素数の場合
    if n > 1:
        primes.append(int(n))

    return primes
```

ナイーブな実装では、自然数 n が与えられたときの計算量は $O(\sqrt{n})$ です。m 個の素因数分解をするときは計算量が $O(m\sqrt{n})$ となります。

ii.　SPF を用いた実装

SPF(Smallest Prime Factor: 自然数 n を割り切る最小の素数) を用いることで、ナイーブな実装よりも高速に素因数分解を行うことができます。SPF を用いた素因数分解はエラトステネスの篩をする際に SPF を記録することで実装できます。

コード 6　SPF を用いた素因数分解の実装

```python
def spf(n: int) -> list[int]:
        spf_table = [i for i in range(n + 1)]
        is_prime = [True] * (n + 1)

        i = 2

        while i * i <= n:
            if is_prime[i]:
                for j in range(i + i, n + 1, i):
                    is_prime[j] = False
```

```
                    if spf_table[j] == j:
                        spf_table[j] = i

            i += 1

        return spf_table

    def prime_factorization(n: int) -> list[int]:
        primes = []
        spf_table = spf(n)

        while n > 1:
            prime = spf_table[n]
            n //= prime
            primes.append(prime)

        return primes
```

iii.　問題 1.

素因数分解に関する問題です。

問題 約数列挙

自然数 N が与えられるので、N の約数を昇順に列挙せよ。ただし、N は $1 \leq N \leq 10^12$ を満たす。

例

入力: 12

出力: 1 2 3 4 6 12

回答:

```python
def enumerate_divisors(n: int) -> list[int]:
    divisors = set()
    for i in range(1, int(n ** 0.5) + 1):
        if n % i == 0:
            divisors.add(i)
            if i != n // i:
                divisors.add(n // i)

    return list(divisors)


def main():
    n = int(input())
    divisors = enumerate_divisors(n)

    print(*divisors)

if __name__ == "__main__":
    main()
```

問題 2.　ABC 057 C - Digits in Multiplication

約数列挙の問題です。

問題 3.　ABC 052 C - Factors of Factorial

約数の個数を求める問題です。約数の個数も素因数分解の結果から求めることを利用します。

V.　冪乗: 繰り返し二乗法

x^n を求める際に単に $x \times x \times \cdots \times x$ と計算すると $O(n)$ の計算量がかかります。繰り返し二乗法を用いると、$O(\log n)$ で計算することができます。また、冪乗では計算結果が非常に大きくなることがあるので、計算結果を mod で割った余りを求めることがあります。剰余で答えを出すときの演算によって処理が異なるので注意が必要です。

- 加算: 加算した後で mod を取る
- 減算: 減算した後で mod を取る
- 乗算: 乗算した後で mod を取る
- 除算: 計算途中で mod を取るときは逆元を用いる (後述)

コード 7　繰り返し二乗法の実装

```python
def exp_mod(n: int, m: int, mod: int) -> int:
    if m == 0:
        return 1
    if m == 1:
        return n % mod
    ans = 1
    m1, m2 = m // 2, m % 2
    ans *= exp_mod(n, m1, mod) % mod
    ans = (ans * ans) % mod

    ans *= exp_mod(n, m2, mod)
    ans %= mod

    return ans
```

VI.　逆元とフェルマーの小定理

除算の mod を取るとき、計算途中で mod をもとった場合と最後に mod を取った場合で結果が異なることがあります。例えば、$a = 20, b = 4, mod = 6$ で考えます。

- **計算途中で mod を取る場合:**
  1. まず、a と b に対して mod を取ります:

$$a \mod m = 20 \mod 6 = 2$$

$$b \mod m = 4 \mod 6 = 4$$

2. 次に、得られた mod の値で除算を行い、さらに mod を取ります:

$$\left(\frac{a \mod m}{b \mod m}\right) \mod m = \left(\frac{2}{4}\right) \mod 6$$

通常の整数除算では、これは 0（小数は考慮しない）となるため、

$$0 \mod 6 = 0$$

結果は 0 になります。

- **最後に mod を取る場合:**

1. まず、a を b で割ります:

$$\frac{20}{4} = 5$$

2. 次に、その結果に対して mod を取ります:

$$5 \mod 6 = 5$$

除算の mod を求めるときは、計算途中で mod を取ると結果が異なることがあるので、逆元を用いて計算します。

i.　フェルマーの小定理と逆元

除算における mod を考えるために**フェルマーの小定理**と**逆元**を導入します。

フェルマーの小定理

a は任意の自然数、m を素数で a, m が互いに素であるとき、以下の式が成り立つ。

$$a^{m-1} \equiv 1 \pmod{m}$$

逆元

m が素数で、a が m では割り切れない整数であるとき、以下の式を満たす x が [1, m) の範囲で一意に存在する。このような x を $\pmod{m}$ における a の逆元と呼ぶ。

$$ax \equiv 1 \pmod{m}$$

ここで、$a^{m-1} \equiv a \times a^{m-2} \equiv 1 \pmod{m}$ となるので、a^{m-2} が a の逆元であることがわかります。以上より以下のことがわかります。

a で割ることは、a の逆元をかけることと等しい

逆元を使った問題を解いてみましょう。

問題 ${}_n\mathrm{C}_k$ を 998244353 で割ったあまりを求めるプログラムを作成せよ。

回答 ${}_n\mathrm{C}_k = \frac{n!}{(n-k)!k!}$ の剰余を求めるます。 mod の世界で除算が登場しているので、逆元を用いて計算します。乗算するのは $(n-k)!, k!$ であることから、これらで割ることは mod の世界では、これらの逆元つまり、それぞれ $(n-k)!^{(DIV-2)}, k!^{(DIV-2)}$ をかけることと等しいです。

コード 8　${}_n\mathrm{C}_k$ を求める

```python
DIV = 998244353

def factorial(n: int) -> list[int]:
    memo = [1] * (n + 1)
    for i in range(1, n + 1):
        memo[i] = (memo[i-1] * i) % DIV

    return memo

def inverse(n: int) -> int:
    return pow(n, DIV - 2, DIV)

def comb(n: int, k: int, memo: list[int]) -> int:
    return (memo[n] * inverse(memo[k]) * inverse(memo[n-k])) % DIV

n, k = map(int, input().split())
memo = factorial(n)
print(comb(n, k, memo) % DIV)
```

VII.　参考

参考になるサイトには扱ったアルゴリズムの別の見方や演習問題などがあるので、以下にリンクを示します。

最大公約数

- https://qiita.com/drken/items/0c88a37eec520f82b788

素数判定

- https://algo-method.com/tasks/318