競技プログラミングと初等整数論入門

67 回生 佐竹俊哉

1. はじめに

初めまして、satashun と申します。
普段はのんびり数学やプログラミングをして楽しんでいます。
自分は主にプログラミングの中でも、特に決められた時間の中で問題を解く競技プログラミングというものに興味を持っています。そのようなプログラミングコンテストでは、プログラムの実行速度が重要であり、プログラムを高速化するために数学的知識を要求される問題が出題されることもあるので、今回は特に初等整数論に分類されるものを中心に書こうと思います。(ちょっと基本的すぎるかもしれませんが・・・)また，時々載せてあるコードは C++で記述しています。
コンテストの例としては、

a. JOI/IOI(情報オリンピック)
   中学生・高校生が対象の科学オリンピックの一つで、予選->本選->春合宿というステップを踏んで、良い成績を収めると IOI(国際情報オリンピック)に日本代表として参加することができます。
b. Topcoder (自分は恥ずかしながら 2013/01/13 の Single Round Match 566 が初参加となりました)
   2 つの Division に分けられていて 75 分で 3 問を解きます。他人の誤解答を見つけると自分の得点が増えたりします。
c. Codeforces
   Topcoder に似たコンテストですが、2 時間で 5 問解くという点が異なります。
d. Google Code Jam

形式が情報オリンピックの予選に似ていて、手元で実行した結果を提出するというタイプです。

e. Atcoder Regular Contest

日本語なので初心者に優しくて、問題も面白いです。

などがあります。

また、Project Euler : http://projecteuler.net/ というサイトがあって、数学的な問題ばかりが掲載されていて、プログラミング＆数学(＆英語)の勉強ができて良いと思います。(多倍長演算が必要になることも多いので多倍長演算が標準で使える言語でやるといいかもしれません)

2. ユークリッドの互除法

初めに、おなじみの最大公約数を求めるアルゴリズムです。ユークリッドの互除法は相当有名なので小学生でも知っているかもしれません。

2 つの整数 a, b の最大公約数とは、それら両方を割り切る最大の整数を表し、それを gcd(a, b)で表す。
a を b で割った余りを p, 商を q として、a = b * p + q であり、gcd(b, q)は a, b を割り切るので gcd(a, b)を割り切る。 移項すると q = a – b * p より同様に gcd(a, b)が gcd(b, q)を割り切ることが言えて、gcd(a, b) = gcd(b, a % b)となる。 b が 0 の時、gcd(a, b) = a なので、そこで完了する。また q は 0 以上 b 未満の整数なので、 ユークリッドの互除法を繰り返すと b は単調減少して、gcd(a, b)が必ず求まる。 証明は省略するが、ユークリッドの互除法は b の桁数の 7 倍以下の回数を繰り返せば求まることも言える。(2 を底とした対数を考えればわかる。)

再帰関数を利用し、これをコードに起こすとこうなります。

```
int gcd(int a, int b) {
  if(b == 0)
    return a;
  return gcd(b, a % b);
```

```
}
```

C++であれば、algorithm というヘッダファイルを include すると、標準で__gcd(a, b)という関数があるので是非使いましょう。
ちなみに最小公倍数は

```
int lcm(int a, int b) {
  return a / gcd(a, b) * b;
}
```

とすることで求まります。(b をかける前に gcd を書いているのはオーバーフローを避けるためです)
ここで、「ax + by = 1 の解の整数解 x, y を求めよ」、という問題を考えましょう。

まず gcd(a, b) ≠ 1 の時は、左辺が gcd(a, b)の倍数となり、右辺は明らかに gcd(a, b)の倍数でないのでこの方程式は整数解を持たない。
gcd(a, b) = 1 の時はユークリッドの互除法を応用すると解を求めることができる。この方程式の解を求める関数を extgcd(a, b, x, y)とし、返り値は gcd(a, b)にする。
bx’ + (a % b) * y’ = gcd(a, b)の解 x’, y’がわかっているとすると、a % b = a − (a / b) * b であることを使い、
ay’ + b * (x’ − (a / b) * y’) = gcd(a, b)と変形できる。
b = 0 の時は　a * 1 + b * 0 = a = gcd(a, b)である。

これをさっきと同様にコードにすると、

```
int extgcd(int a, int b, int& x, int& y) {
  int gcd_ = a;
  if (b != 0) {
    gcd_ = extgcd(b, a % b, y, x);
    y -= (a / b) * x;
  } else {
    x = 1; y = 0;
  }
  return gcd_;
}
```

のようになります。(実行し終わった時に x, y に解が入っています)
また、これで得られた解を使うと、ax + by = 1 の全ての解は、(x + kb, y - ka)で表せます。
では、ax + by = 1 の解の求め方が分かったので、ax + by = gcd(a, b)という方程式を考えてみましょう。少し考えると、gcd(a, b)は a, b 両方を割り切るので、この式の解は(a / gcd(a, b)) * x + (b / gcd(a, b)) = 1 の解と一致することがわかり、さっきのアルゴリズムで解を求められます。

3. 素数に関するアルゴリズム

整数 p >= 2 の正の約数が 1 と p だけの時、p を素数と呼び、約数が他にもある時は合成数と呼びます。素数に関するアルゴリズムは色々あります。
ですが、それを紹介する前に、素因数分解の一意性について簡単な証明を載せておきます。(一見自明に思えますが、証明を考えてみると簡単でもないので一応)
この証明にはいくつか準備を必要とします。

定理 3.1
p を素数とし、p が積 ab を割り切るならば、p は a, b の少なくとも一方を割り切る。

証明
p が a を割り切る時、主張は明らかに成り立つので、p は a を割り切らないとする。この時、gcd(a, p)を考えると、これは p を割り切るので 1 or p で、p は a を割り切らないとする仮定より gcd(a, p) = 1 がわかる。ここで前の章で紹介した、px + ay = 1 という方程式を考えると、gcd(p, a) = 1 より整数解を持つことが分かる。解の一つを x = x1, y = y1 とすると、px1 + ay1 = 1 が成り立ち、両辺に b をかけて pbx1 + aby1 = b とな

る。p は pbx1 を割り切り、仮定より p は ab を割り切るので、左辺は p の倍数、故に b も p の倍数となり、示せた。

定理 3.2
p を素数とし、p が積(a1)*(a2)*…*(ar)を割り切るならば、p は a1, a2, …, ar の少なくとも一つを割り切る。

証明
p が a1 を割り切るならば明らかである。そうでないとすると a1 * ((a2)*(a3)*…*(ar))を考えれば定理 3.1 より p は ((a2)*(a3)*…*(ar))を割り切る。p が a2 を割り切るとすれば、明らかである。そうでないとすると…という議論を繰り返すと、p が a1…ar のいずれかを割り切ることが示せる。

定理 3.3 素因数分解の一意性
すべての整数 n ≧ 2 は素数の積 n = p1*p2*p3*…*pr に一意的に分解できる。

証明
① 整数 n ≧ 2 は素数の積で表せる。
② その分解は並べ方の違いを除き一通りである。
の 2 つを示せば良い。

まず①を示す。
(1) n = 2 のときは明らかに成り立つ。
(2) n ≦ k の時成立すると仮定する。
k + 1 が素数の時はそれ自身が素数の積に表されていると言える。k + 1 が合成数の時、2 ≦ n1, n2 ≦ k が存在して k + 1 = n1 * n2 と表せるが、n1, n2 は仮定より素数の積に表せるので、n1 = (p1)*(p2)* …*(pr), n2 = (q1)*(q2)*…*(qs)と表せ、かけると k + 1 = (p1)*(p2)*…*(pr)*(q1)*(q2)*…*(qs)が得られるので、k + 1 も素数の積に表せる。

(1)(2)より数学的帰納法から①が示せた。

次に②を示す。
整数 n が素数の積として n = (p1)*(p2)*…*(pr) = (q1)*(q2)*…*(qs)の 2 通りに表せるとする。定理 3.2 より p1 は q1, …, qs の少なくとも 1 つを割り切るので、適当に並び替えて q1 が p1 の倍数だとして良い。しかし q1 は素数なので約数は 1 と q1 のみである。よって p1 = q1 だと分かる。
ここでさっきの式の両辺から p1, q1 を消去すると、(p2)*(p3)*…*(pr) = (q2)*(q3)*…*(qs)となる。同様に(p3)*(p4)*…*(pr) = (q3)*(q4)*…*(qs)が得られ、この議論は左辺か右辺が 1 になるまで繰り返すことができる。しかし左辺か右辺のいずれか片方に残ってしまうと、1 = (素数の積) の形になってしまって矛盾するので、r = s が分かる。
以上より②も示せた。

これで素因数分解が一意にできることがわかりました。
では素数判定の仕方などを考えてみましょう。
まず、2 以上の自然数 n が与えられた時、素数の定義に従って、2 から n − 1 で割れなければ素数とするという愚直な方法を思いつくかもしれません。ですが、この方法は非常に無駄が多いです。少し考えると、2 から sqrt(n)まで調べればいいことがわかります。これは、n の約数の 1 つを d とすると、n / d が整数かつ n の約数となり、n > sqrt(n), n / d > sqrt(n)と仮定した場合には、両辺を掛けると n > n となってしまって矛盾することから分かります。
これで素数判定や素因数分解などがそれなりに高速にできるようになりました。
これでも非常に大きな数が与えられると膨大な時間がかかってしまいますが、コンテストだとたいてい十分です。しかし素数判定を複数回する時は、もっと効率的なアルゴリズムがあります。エラトステネスの篩という有名なアルゴリズムです。

エラトステネスの篩

2 以上 n 以下の整数を列挙し、その中にある最小の数の倍数をすべて取り除くという操作を繰り返すと素数を列挙することができる、というものです。
これは簡単なプログラムで実現することができます。

```
vector<int> prime;
bool not_prime[1000001];

void sieve(int n) {
  not_prime[0] = not_prime[1] = true;
  for (int i = 2; i <= n; i++) {
    if (!not_prime[i]) {
      prime.push_back(i);
      for (int j = 2 * i; j <= n; j += i)
        not_prime[j] = true;
    }
  }
}
```

not_prime という配列にしているのは、bool を global に宣言すると false で初期化される性質を使いたかったからです。例えばこのプログラムを sieve(1000000)で実行すると 1000000 以下の素数が prime に入れられます。似たようなアルゴリズムとしてアトキンの篩というものもあります。
素数判定や素因数分解には面白い方法がもっとありますが、まだ道具が足りないので後に少しだけ紹介します。
また、素数かどうかを判定するよりも素因数分解をすることのほうが難しいことがわかっていて、このことが RSA 公開鍵暗号方式などに利用されています。

4. 合同式

今まで見てきたとおり、割り切れるかどうかということは数論において強力です。そして、合同式というものを使うと整除性を便利に扱うことができます。

2 つの整数 a, b の差が正整数 m で割り切れる時、a ≡ b と書き、a は m を法として b と合同であるという。(a is congruent to b modulo m)
a ≡ b mod m を満たす a, b について、それぞれ m で割った商を q1, q2, 余りを r1, r2 とすると(r1, r2 は 0 以上 m − 1 以下の整数)a = mq1 + r1, b = mq2 + r2 と表せる。
この時、a − b = m * (q1 − q2) + (r1 − r2)で、これが m の倍数であるのは r1 = r2 の時に限るので、a が b を法として合同とは、a を m で割った余りと b を m で割った余りが等しいということである。

合同式の基本性質は次の 2 つがあげられます。

① 正整数 n が正整数 m の倍数で、かつ a ≡ b mod n ならば a ≡ b mod m となる。
これは、n = mk(k は正整数)、a − b = nl(l は整数)と表せるので、a − b = m(kl)より明らか。

② 任意の整数 m は正整数 m を法として 0, 1, …m − 1 のいずれかと合同である。
これは余りの定義から明らか。

また合同式の上では普通の式のような演算が可能です。

a1 ≡ a2 mod m かつ b1 ≡ b2 mod m ならば
(1) a1 + b1 ≡ a2 + b2 mod m
(2) a1 − b1 ≡ a2 − b2 mod m
(3) a1 * b1 ≡ a2 * b2 mod m

証明はほとんど自明なので、(3)だけ示しておきます。

a2 = a1 + mk(k は整数), b2 = b1 + ml(l は整数)と表せて、a2 * b2 = (a1 + mk) * (b1 + ml) = a1 * b1 + (a1 * l + k * b1 + m* k * l) * m より a2 * b2 ≡ a1 * b1 mod m が導ける。

ただし合同式の上では、普通の式と全く同じようには、割り算をすることができません。例えば、3 * 4 ≡ 5 * 4(mod 8)ですが 3 ≡ 5(mod 8)ではないということから分かります。
しかし、互いに素という条件があれば割り算をすることができます。

逆元

逆元とは、次のように定義されたものを言います。

> 互いに素な整数 a, m に対して、gcd(a, m) = 1 より、2 章で書いたように ax + my = 1 を満たす整数 x, y が存在することが分かる。この時 ax – 1 = –my より ax – 1 ≡ 0 mod m となる。つまり ax ≡ 1 mod m である。さらに整数 b, c が ab ≡ ac mod m を満たしているならば、両辺に先ほど見つけた x をかけると、b ≡ c mod m が分かる。このように整数 a, m に対して ax ≡ 1 mod m を満たす整数 x を、法 m に関する a の逆元と呼ぶ。

逆元には、以下の様な特徴があります。

> ax + my = 1 の解は無数にあるが、x, x’が法 m に関する a の逆元だとすれば x ≡ x’ mod m であることがわかる。よって、逆元は一意に定まるということが言える。また、逆に法 m に関する a の逆元 x が存在するならば、整数 y を用いて ax = 1 + my ⇔ ax – my = 1 と表せるので gcd(a, m) = 1 が言える。ここから、法 m に関する a の逆元 x が存在することと、a と m が互いに素であることが同値であることが分かる。特に p を素数とすると、a が p で割り切れない時 a と p は互いに素となるので、p で割り切れない任意の整数に対して法 p に関する逆元が存在する。(これを a ^ (–1)と表す)

以上のことより、逆元は拡張ユークリッドの互除法を用いると計算できることが分かります。具体的にコードにすると、以下のようになります。

```
int inv_mod(int a, int m) {
  int x, y;
  if (extgcd(a, m, x, y) == 1)
    return (x + m) % m;
  else
    return 0; // 逆元は存在しない
}
```

そして次は、逆元を違う方法で求められたり、素数判定に使えたりと便利な、合同式に関する有名な定理を紹介します。

フェルマーの小定理

フェルマーの小定理とは、次のようなものです。

> p を素数とし、a を p で割り切れない整数とすると a ^ (p − 1) ≡ 1 mod p が成り立つ。

このフェルマーの小定理について証明するに、補題について証明を行います。

> **補題**
> p を素数とし、a を p で割り切れない整数とすると a, 2 * a, 3 * a, … ,(p − 1) * a (mod p)は数 1, 2, 3,…, (p − 1)(mod p)と順序を除いて一致する。

> **証明**
> a, 2 * a, 3 * a, …(p − 1) * a のうち ja ≡ ka(mod p)となる 2 つの整数 ja, k をとる。このとき、合同式の定義より p は(j − k) * a を割り切るが、p は a を割り切らないので 3 章の定理 3.1 より、p は j − k を割り切る。ここで 1≦ j, k ≦ p − 1 より|j − k| < p − 1 なので j = k となる。よって a, 2a, …, (p − 1)a は p を

法としてすべて異なる。また、これらが p で割り切れないことは明らかであり、p を法として 0 でない数は 1, 2…, p − 1 の p − 1 個なので、示せた。

次に、フェルマーの小定理の証明です。

補題より a, 2 * a, 3 * a, …, (p − 1) * a (mod p)と 1, 2, 3, …, (p − 1) (mod p)の集合は同じなので、それぞれの積は等しく a * (2a) * (3a) … ((p − 1) * a) = 1 * 2 * 3…(p − 1) (mod p)が成り立つことが分かる。これを整理すると、 a ^ (p − 1) * (p − 1)! ≡ (p − 1)! (mod p)が言えるので、(p − 1)!と p は互いに素であることから、両辺から消去して a ^ (p − 1) ≡ 1(mod p)を示せる。

フェルマーの小定理は二項定理を使ったり、剰余系を考えたりしても証明することができます。(上の証明は本質的には剰余系を考えています)
逆元のところで述べたように、素数 p で割り切れない a に対して法 p に関する逆元が存在し、フェルマーの小定理より a ^ (p − 1) ≡ 1 mod p が成り立つので、a ^ (p − 2) ≡ a ^ (− 1) mod p となって逆元を求めることができます。(p − 2 乗を求める際に繰り返し二乗法というアルゴリズムを用いると良いです)

```
typedef long long ll;

ll mod_pow(ll x, ll k, ll mod) {
  ll ret = 1;
  while (k) {
    if (k & 1)
      ret = ret * x % mod;
    x = x * x % mod;
    k >>= 1;
  }
  return ret;
}

ll fermat_inv_mod(ll a, ll p) {
  return mod_pow(a, p - 2, p);
}
```

フェルマーの小定理は、素数判定にも利用することができます。フェルマーの小定理の対偶を考えると、p の互いに素な整数 a について、a ^ (p − 1) ≡ 1 mod p でないならば、p は素数ではない、ということがわかるので、これを利用して確率的素数判定をすることができ、この手法はフェルマーテストと呼ばれています。
n と互いに素な十分な数の相異なる整数 a に対して a ^ (n − 1) ≡ 1 mod n が成り立つ時、n は素数とは限りませんが、「確率的」に素数だと言え、こういった n を擬素数と呼びます。また a ^ (n − 1)が n を法として 1 と等しくならない a が見つかれば、n は絶対に合成数であるので、このような a を n に対する証人と呼びます。
少し調べると、n が合成数の時はほとんどの a の値が証人になっているように思えますが、実はどんな a をとっても擬素数と判断されるような数が存在して、カーマイケル数と呼ばれています。つまり、カーマイケル数は合成数であるにも関わらず、フェルマーテストでは素数だと判定することができないのです。しかしながら、カーマイケル数はかなり限られているので、フェルマーテストはそれなりに有効と言えます。
例えばカーマイケル数には 561, 1105, 1729, 2465, 2821, 6601, 8911 などがあり、無数に存在することが示されています。(実際に 561 = 3 * 11 * 17, 1105 = 5 * 13 * 17, …となり合成数です)しかし、カーマイケル数には、相異なる奇素数の積であるといった性質があり、合成数 n がカーマイケル数である必要十分条件は、n を割り切る全ての奇素数について、p ^ 2 が n を割りきらず、p − 1 が n − 1 を割り切ることである、ということが言えます。この条件を用いて、カーマイケル数かどうかを判定する手法としてコルセルトの判定法というものがあります。
また、カーマイケル数が存在するためにフェルマーテストは不完全と言えますが、もっと良いアルゴリズムとして Solovay-Strassen 素数判定法やミラー・ラビン素数判定法や AKS 素数判定法といったものが知られています。

ここでは、高速で、また手軽であるミラー・ラビン素数判定法について軽く説明しておきます。この素数判定法は、次に述べる素数の性質を利用しています。

p を奇素数とし、p − 1 = (2 ^ k) * q となる素数 q をとる。a を p で割り切れない任意の整数とすると、
(1)　a ^ q は p を法として 1 に合同
(2)　a ^ q, a ^ 2q, a ^ 4q, … , a ^ (2 ^ (k − 1) * q)の内 1 つは p を法として−1 に合同
のどちらかが成り立つ。

証明
まず(2 ^ k) * q = p − 1 よりフェルマーの小定理から、a ^ ((2 ^ k) * q) = a ^ (p − 1) ≡ 1 (mod p)が成り立つ。また x ^ 2 ≡ 1 (mod p) ⇔ (x − 1) * (x + 1) ≡ 1 (mod p) ⇔ x ≡ 1, −1 (mod p)が言え、かつ(2)の各数は直前の数の平方になっている。よって(2)の各数を、大きい順、つまり a ^ (2 ^ (k − 1) * q), …, a ^ 4q, a ^ 2q, a ^ q の順に見ていくと、p を法として−1 と合同な数が現れれば(2)を満たし、逆に−1 と合同でなければ 1 と合同であるので、 (1)を満たす。

これの対偶を用いると、素数判定をすることができ、その手法をミラー・ラビン素数判定法と言います。具体的には、n を奇数とし、 n − 1 = (2 ^ k) * q となる奇数 q をとった時に、

(a)　a ^ q ≡ 1 (mod n)でない
(b)　i = 0, 1, …, k − 1 全てに対して a ^ ((2 ^ i) * q) ≡ −1(mod n)でない

の両方が成り立つと、n は合成数であるということです。
これは a ^ q (mod n)を計算した後、平方を繰り返すことで簡単に計算できます。フェルマーテストと同じで、任意の整数 a に対しこの判定法を用いると、n が合成数であると示すラビン・ミラー

証人になるか、もしくは n が素数かもしれないことを示唆します。この判定法では、フェルマーテストと違い、カーマイケル数のようなうまく判定できない数は存在しません。よって、繰り返す回数を k とすると、合成数を素数と誤判定する確率は最大で(1 / 4) ^ k となることがわかり、十分な回数繰り返せばほぼ確実に素数判定ができるのです。
これを実装すると次のようになります。

```
typedef long long ll;

bool miller_rabin(ll n, int k) {
  int s = 0;
  ll d = n - 1;
  srand((unsigned)time(NULL));
  while (d % 2 == 0) {
    d /= 2;
    s++;
  }
  for (int i = 0; i < k; i++) {
    ll a = rand() % (n - 1) + 1;
    ll y = mod_pow(a, d, n);
    if (y == 1)
      continue;
    for (int j = 0; j < s; j++) {
      if (y == n - 1)
        break;
      y = y * y % n;
    }
    if (y == n - 1)
      continue;
    return false;
  }
  return true;
}
```

実際に試してみると、k(繰り返す回数)を 10 として、10 ～ 1000000 の数全てを判定するのに 0.997 秒程度で済みました。
また、フェルマーの小定理は p を合成数とした場合には成り立ちませんが、p が合成数の場合に拡張した定理としてオイラーの公式が知られています。

オイラーの定理

オイラーの定理とは以下のようなものです。

> $\phi(m)$: 1 以上 m 以下の整数のうち、m と互いに素なものの個数(オイラーのトーシェント関数、ϕ 関数ともいう)として、a と m が互いに素な時、 $a^{\phi(m)} \equiv 1 \bmod m$ が成り立つ。

これもフェルマーの小定理と似たような方法で示すことができます。

> **補題**
> $1 \leqq b1 < b2 < \cdots < b\phi(m) < m$ を 0 と m の間にある m と互いに素な $\phi(m)$個の数とする。a と m が互いに素である時、$b1 * a, b2 * a, b3 * a, \cdots b\phi(m) * a \bmod m$ は $b1, b2, \cdots b\phi(m) \bmod m$ と順序を除いて一致する。
>
> **証明**
> ある数 b が m と互いに素であるならば $a * b$ も m と互いに素なので、$b1 * a, b2 * a, b3 * a, \cdots b\phi(m) * a$ に含まれる数はそれぞれ、$b1, b2, \cdots b\phi(m)$ のうちの 1 つと m を法として合同である。前者から 2 つの数 $bj * a$ と $bk * a$ をとり、$bj * a \equiv bk * a \pmod m$だとすると、m は$(bj - bk) * a$ を割り切るが、m と a は互いに素であることから、m は $bj - bk$ を割り切ることがわかる。しかし、bj, bk は共に 1 以上 m 以下なので$|bj - bk| \leqq m - 1$ である。よって $bj - bk = 0$ つまり $bj = bk$ が言える。よって $b1 * a, b2 * a, \cdots b\phi(m) * a \bmod m$ は m を法としてすべて異なる。

この補題を用いて、オイラーの定理を証明します。

> **証明**

補題より、(b1 * a) * (b2 * a) * (b3 * a) … (bφ(m) * a) ≡ b1 * b2 * … * bφ(m) mod m が成り立つ。整理すると、(a ^ φ(m)) * (b1 * b2 * … * bφ(m)) ≡ b1 * b2 * … * bφ(m)mod m となり、b1, b2, … , bφ(m)は m と互いに素であることから、a ^ φ(m) ≡ 1 mod m が示される。

また、素数かどうかを判定するよりも素因数分解をすることのほうが難しいことがわかっていて、このことが RSA 公開鍵暗号方式などに利用されています。
オイラーの定理を活用するためにはφ関数の値を知る必要がありますが、φ関数は、

① p が素数、k を自然数とするとφ(p^k) = p^k − p^(k − 1)
② m, n を互いに素な自然数とするとφ(mn) = φ(m) * φ(n)

という性質を持っています。これを用いると、

n = (p1^e1) * (p2^e2) * … * (pd^ed)と素因数分解される時、φ(n) = φ(p1^e1) * φ(p2^e2) * … * φ(pd^ed) = (p1^e1 − p1^(e1 − 1)) * (p2^e2 − p2^(e2 − 1)) * … * (pd^ed − pd^(ed − 1)) = n * (1 − 1/p1) * (1 − 1/p2) * … (1 − 1/pd)

というように表すことができ、φ関数を簡単に求めることができます。コードにすると、以下のようになります。

```
int euler_phi(int n) {
  if (n == 0)
    return 0;
  int ret = n;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      ret -= ret / i;
      while (n % i == 0)
        n /= i;
```

```
    }
  }
  if (n != 1)
    ret -= ret / n;
  return ret;
}
```

また、複数回φ関数の値を求めたい時は、

```
const int sz = 1000000;
int totient[sz];

void _euler_phi() {
  for (int i = 0; i < sz; i++)
    totient[i] = i;
  for (int i = 2; i < sz; i++)
    if (totient[i] == i)
      for (int j = i; j < sz; j += i)
        totient[j] -= totient[j] / i;
}
```

のようにすればまとめて表を作っておくことができます。
また、a ^ φ(m) ≡ 1 mod m は成り立ちますが、このφ(m)が a ^ p ≡ 1 を満たす最小の整数であるとは限らないので、この最小の整数を与える関数としてカーマイケルのλ関数が知られています。λ(n)は、n と互いに素な整数 a に対して a ^ m ≡ 1(mod n)となる m を表します。λ関数は帰納的に定義されていて、

(1) k ≧ 3 の時
λ(2 ^ k) = 2 ^ (k − 2), λ(2 ^ 1) = 1, λ(2 ^ 2) = 2

(2) n が奇素数 p を使って p ^ k と表せる時
λ(p ^ k) = (p − 1) * p ^ (e − 1)

(3) n が(p1 ^ k1) * (p2 ^ k2)…(pq ^ kq)と素因数分解できる時
λ((p1 ^ k1) * (p2 ^ k2)…(pq ^ kq)) = lcm(λ(p1 ^ k1), λ(p2 ^ k2), …, λ(pq ^ kq))

のように計算できます。また、以下のようにすると、奇素数と 2 を区別する手間を省くことができます。

```
int carmichael_lambda(int n) {
  int ret = 1;
  if (n % 8 == 0)
    n /= 2;
  for (int i = 2; i <= n; i++) {
    if (n % i != 0)
      continue;
    int sub = i - 1;
    n /= i;
    while (n % i == 0) {
       n /= i;
       sub *= i;
    }
    ret = lcm(ret, sub);
  }
  return ret;
}
```

またλ(n)がn－1を割り切る時、nはフェルマーテストのところで紹介したカーマイケル数になっていることが知られています。

5. 練習

色々紹介してきたので、初めに紹介した、Project Eulerの問題をいくつか解いてみせたいと思います。Project Eulerは実行時間に制限が無いので、計算量はあまり気にしなくても良いです。(とはいっても現実的な時間で実行できないといけませんが)
初めの方はとても簡単です。

Problem 1 Multiples of 3 and 5

3または5で割り切れる、1000未満の自然数の和を答えよ

範囲が小さいので探索しても等差数列の和で求めても良いです。以下のソースコードでは、3の倍数と5の倍数の和を求めてから15の倍数を引いています。

```
#include <cstdio>
using namespace std;

int n;
```

```
int getsum(int k) {
  int res = 0;
  for (int i = 0; i < n; i++)
    if (i % k == 0)
      res += i;
  return res;
}

int main() {
  scanf("%d", &n);
  printf("%d¥n", getsum(3) + getsum(5) - getsum(15));
  return 0;
}
```

Problem 2 Even Fibonacci numbers

1, 2, 3, 5, 8…というような 4000000 未満のフィボナッチ数列のうち、偶数である項の和を求めよ。

実際にフィボナッチ数列を作っていって、偶数だったら足すという処理をします。メモリを節約するために配列を使い回しています。

```
#include <cstdio>
using namespace std;

int dp[2];

int main() {
  dp[0] = 1;
  dp[1] = 2;
  int f = 1;
  int ret = 2;
  while (dp[0] < 4000000 && dp[1] < 4000000) {
    dp[f ^ 1] = dp[0] + dp[1];
    if (!(dp[f ^ 1] & 1))
      ret += dp[f ^ 1];
    f ^= 1;
  }
  printf("%d¥n", ret);
  return 0;
}
```

Problem 3 Largest prime factor

600851475143 の最大の素因数を答えよ。

前に紹介したように、実際に素因数分解します。C++だと int に収まらないことに注意しましょう。

```
#include <cstdio>
#include <map>
using namespace std;

#define FOR(i,a,b) for(int i=(a);i<(b);++i)
#define rep(i,n)  FOR(i,0,n)
#define foreach(c,itr) for(__typeof((c).begin())
itr=(c).begin();itr!=(c).end();itr++)

typedef long long ll;

map<ll, ll> solve(ll n) {
  map<ll, ll> res;
  for (ll i = 2; i <= n; i++) {
    ll ret = 0;
    while (n % i == 0) {
      ret++;
      n /= i;
    }
    if (ret)
      res[i] = ret;
  }
  if (n != 1)
    res[n] = 1;
  return res;
}

int main() {
  map<ll, ll> ans = solve(600851475143LL);
  foreach(ans, itr)
    printf("%lld %lld¥n", itr->first, itr->second);
  return 0;
}
```

Problem 4 Largest palindrome product

2 つの 3 桁の整数の積で表せる最大の回文数を答えよ。

候補がそんなに多くないので素直に全探索しています。回文数は、文字列に変換して、左右逆転させたものと一致すれば OK というように判定しています。

```
int ma = 0;

bool ok(int k) {
  string tmp = "";
  while (k > 0) {
    char c = k % 10 + '0';
    tmp.push_back(c);
    k /= 10;
  }
  string rev = tmp;
  reverse(rev.begin(), rev.end());
  if (tmp == rev)
    return true;
  else
    return false;
}

int main() {
  for (int i = 999; i >= 100; --i) {
    for (int j = 999; j >= 100; --j){
      int pro = i * j;
      if (ok(pro))
        ma = max(ma, pro);
    }
  }
  printf("%d¥n", ma);
  return 0;
}
```

Problem 5 Smallest multiple

1 から 20 の整数全てで割り切れる最小の自然数を答えよ。

問題文通り、1 から 20 までの最小公倍数を計算すればいいだけです。

```
ll ans = 1;

ll lcm(ll a, ll b) {
  return a / __gcd(a, b) * b;
}

int main() {
  for (int i = 1; i <= 20; i++)
    ans = lcm(ans, (ll)i);
  printf("%lld¥n", ans);
  return 0;
}
```

6. まとめ

今までは、競技プログラミングに挑戦する上で基本となる簡単な知識や問題を説明してきました。
初めは簡単なものから解いていけばだんだん難しい問題が解けるようになっていって楽しいと思うので、是非こういう問題もやってみてください。
読んでくださってありがとうございました。

参考文献:
『プログラミングコンテストチャレンジブック 第２版』
秋葉拓哉・岩田陽一・北川宣稔著
『はじめての数論　原著第３版』
ジョセフ・H・シルヴァーマン著　鈴木治郎訳