İkbal Kazar
Yusuf Hakan Kalaycı

İçindekiler

Sayılar Teorisi

Faktoriyel için bölenin üssünü bulmak

n ve k tam sayıları verilmektedir. Hesaplanması gereken n! i bölen en büyük k üssünü bulmaktır diğer bir deyişle k^x böler n! şeklinde olan en büyük x değerinin bulunması gerekmektedir.

Asal k değerleri için

İlk olarak k sayısının asal olduğu durumları değerlendirelim.

Faktoriyel işlemini açık olarak yazalım:

$$n! = 1\ 2\ 3\ \ldots\ (n-1)\ n$$

Bu dizinin başından itibaren her k. sayısı (k'nın katı olan) n!'in bölenidir. Ve bunların sayısını da $\lfloor n/k \rfloor$ formülüyle bulabiliriz.

Ayrıca, her k^2. sayıyı bulmak için verilen diziyi k^2 alt parçaya ayırabiliriz ve her biri için sadece bir tane bölen mevcuttur. Yani $\lfloor n/k^2 \rfloor$ tane k^2 sayısına bölünen sayı mevcuttur belirttiğimiz faktoriyel açılımında.

Ve bu şekilde devam ederek, her k^i. sayı aranan x değerini 1 artıracaktır. Ve onların sayısı da yine $\lfloor n/k^i \rfloor$ şeklindedir.

Bu yüzden cevabın büyüklüğü şu kadar olacaktır:

$$\frac{n}{k} + \frac{n}{k^2} + \ldots + \frac{n}{k^i} + \ldots$$

Yukarıda verilen toplam kesinlikle sonsuz değildir. Çünkü sadece ilk $\log_k n$ sayı için cevap 0'dan farksızdır kalan hepsi için 0'dır. Sonuç olarak hesaplamak için gereken süre $O(\log_k n)$ olacaktır.

Kodlama:

```
int fact_pow (int n, int k) {
      int res = 0;
      while (n) {
            n /= k;
            res += n;
```

```
        }
        return res;
}
```

Asal olmayan k değerleri için

Aynı fikir burada direk işe yaramamaktadır.

Ancak biz k sayısını asal çarpanlarına ayırabiliriz ve asal çarpanları için elde edilebilecek en küçük değer aranan k sayısı için de en küçük değer olacaktır.

Daha açık bir ifadeyle: k_i sayısı k sayısının i. asal böleni olsun ve üssü p_i şeklinde olsun. k_i değeri için yukarıda belirttiğimiz gibi çözülecektir. Ardından elde edilen sayıyı Ans_i olarak kabul edelim. Bu şartlar altında cevap k sayısı için aranan değer Ans_i / p_i sayılarının minimumu olacaktır.

Çarpanlarına ayırma işleminin gerçekleşmesi için gereken süre $O(\sqrt{k})$'dır. Bu yüzden de genel çalışma süresi $O(\sqrt{k})$ olacaktır.

Sieve of Eratosthenes

Sieve of Eratosthenes (1,n) aralığındaki asal sayıları bulma işlemini $O(n \log \log n)$ işlemde gerçekleştiren bir algoritmadır.

Bu algoritma şu şekilde çalışmaktadır: 2….n aralığındaki tüm sayılar için ilk başta 2 hariç 2'nin böldüğü sayıları bu listeden çıkaracağız. Arından 3 hariç 3'ün böldüğü tüm sayıları bu listeden çıkaracağız. Bu işlemi n e kadar devam ettirdiğimizde dizide sadece asallar kalmış olacaktır.

Kodlama

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
        if (prime[i])
                if (i * 1ll * i <= n)
                        for (int j=i*i; j<=n; j+=i)
                                prime[j] = false;
```

Bu kod öncelikle 0 ve 1 hariç diğer tüm sayıları kontrol eder ve eğer bulunulan sayı asal ise asal olmayan ,bu sayının katı olan ve i^2 diğer tüm sayıları gezerek onları asal değil diye işaretler. Kod üzerindeki önemli bir ayrıntı i^2 değeri int değerinden daha büyük olabileceği için long long int'e cast edilmiştir.

Görüldüğü gibi kod $O(n)$ hafıza gerektirmektedir ayrıca çalışma zamanı $O(n \log \log n)$'dir(bu daha sonra ispatlanacak).

Çalışma zamanı

Şimdi kodun çalışma zamanının $O(n \log \log n)$ olduğunu ispatlayacağız.

Her bir $p \leq n$ asal sayısı için içteki for döngüsü $\frac{n}{p}$ işlem yapmaktadır. Şimdi bizim şu değeri tahmin etmemiz gerekmektedir.

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{n}{p} = n \cdot \sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p}.$$

Şimdi iki bilinen gözlemden bahsedelim: n'den küçük veya eşit asalların sayısı yaklaşık olarak $\frac{n}{\ln n}$'dir. Ayrıca k. asal sayının değeri yaklaşık olarak $k \ln k$ değerine eşittir. Şimdi toplamı şu şekilde yazabiliriz.

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}.$$

Şimdi toplam işlemi yerine integral değerini kullanabiliriz. İntegral değeri yaklaşık olarak verilen toplam işlemine eşittir.

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk.$$

Verilen integralin antiderivative hali $\ln \ln k$'dır.

$$\int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n.$$

Şimdi orijinal toplamın değeri ile hesapladığımız değerin yaklaşık eşitliğini gösterelim.

$$\sum_{\substack{p \le n, \\ p \text{ is prime}}} \frac{n}{p} \approx n \ln \ln n + o(n),$$

Çeşitli Sieve of Eratosthenes Optimizasyonları

Algoritmanın çalışması için gerekli olan $O(n \log \log n)$ zaman ve algoritmanın çalışması için gerekli olan n hafıza zaman zaman kod için büyük sıkıntılar çıkarabilmektedir.

Verilecek olan optimizasyon yöntemleri hem çalışma zamanını hem de gerekli olan hafızayı düşürmektedir.

Köküne kadar bakmak

Dış for döngüsündeki i değerini n'e kadar gezdirmek yerine kök n'e kadar gezdirebiliriz.

Yapılacak değişiklik:

```
for (int i=2; i*i<=n; ++i)
```

Kodun zaman karmaşıklığı açısından bir değişiklik olmayacaktır anca yapılan işlem sayısında büyük oranda azalma olacaktır.

Sadece tek sayılarla ilgilenmek

Bilindiği üzere 2 hariç hiç bir çift tam sayı asal sayı değildir. Biz bu programla sadece tek sayılarla ilgilenirsek çalışma zamanını ve gerekli olan hafızayı yarıya indirmiş oluruz.

Kullanılan hafızayı kısmak

Bool değişkeniyle toplamda 1 byte'lık yer kullanmaktayız ancak bize 1 byte yerine 1 bit'lik yer yeterlidir. Her bir eleman için sadece 1 bit'lik yer kullanarak kullandığımız hafızayı n byte'dan $n/8$ byte'a indirmiş oluruz.

Ancak bu optimizasyonu yapabilmek için bitwise operatörlerini kullanmak gerekmektedir. Ve her bir kontrol işlemi için normalde yaptığımızdan daha fazla işlem yapmış olacağız ve bu da programımızın hızında düşüşe gitmektedir.

Bu tip bir optimizasyonu sadece n değeri çok büyük olduğunda programımızın çalışması için gerekli olan hafıza kısıtı varsa kullanmak mantıklı olmaktadır.

Euler fonksiyonu ve Hesaplanması

Euler fonksiyonu $\phi(n)$ bazen $\varphi(n)$ ve $phi(n)$ de olarak da gösterilebilmektedir. Bu fonksiyon n için 1 ile n arasındaki n ile arasında asal olan sayıların sayısını vermektedir. Diğer bir deyişle 1 ile n arasında EBOB(en büyük ortak bölen) değeri 1 olan sayıların sayısıdır.

Euler fonksiyonun ilk bir kaç değeri:

$$\phi(1) = 1,$$

$$\phi(2) = 1,$$

$$\phi(3) = 2,$$

$$\phi(4) = 2,$$

$$\phi(5) = 4.$$

Özellikler

3 basit özelliği verilecektir. Bu özellikler nasıl hesaplanacağını anlamaya yeterlidir:

- Eğer p asal sayı ise $\phi(p) = p - 1$ olacaktır.
    - Bu bariz bir şekilde görülmektedir ki p hariç 1….p arasındaki bütün sayılarla arasında asaldır.
- Eğer p asal ve a doğal sayı ise $\phi(p^a) = p^a - p^{a-1}$.
    - p^a ile aralarında asal olmayan sayılar şu formdadır: $pk\ (k \in \mathcal{N})\ p^a/p = p^{a-1}$
- Eğer a ve b aralında asal sayılar ise $\phi(ab) = \phi(a)\phi(b)$.
    - Bu gözlem çin kalan teoreminden gelmektedir. Mesela elimize bir z sayısı alalım öyle ki $z \leq ab$. Ayrıca x ve y, z nin a ve b modundaki değerleri olsun. Bu durumda z ve ab aralarında asaldır ancak ve ancak z, a ve b ile ayrı ayrı aralarında asala ise. Çin kalan teoremini de kullanarak, şunu görebiliriz ki: herhangi x ve y öyle ki $(x \leq a,\ y \leq b)$ için $(z \leq ab)$ değeri oluşacaktır.

Son durumda Euler fonksiyonunun değerini sadece çarpanlarına ayırarak bulabiliriz.

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$$

Tüm p_i'ler asaldır.

$$\phi(n) = \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \cdot \ldots \cdot \phi(p_k^{a_k}) =$$

$$= (p_1^{a_1} - p_1^{a_1-1}) \cdot (p_2^{a_2} - p_2^{a_2-1}) \cdot \ldots \cdot (p_k^{a_k} - p_k^{a_k-1}) =$$

$$= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_k}\right).$$

Kodlama

Euler fonksiyonunun en kolay hesaplanması aşağıda gösterilmiştir. Ayrıca çalışma zamanı $O(\sqrt{n})$ dir.

```
int phi (int n) {
      int result = n;
      for (int i=2; i*i<=n; ++i)
            if (n % i == 0) {
                  while (n % i == 0)
                        n /= i;
                  result -= result / i;
            }
      if (n > 1)
            result -= result / n;
      return result;
}
```

Ayrıca bu fonksiyonun daha hızlı bir şekilde de kodlanması mümkündür.

Euler fonksiyonu'nun kullanımları

Bu fonksiyonun en ünlü ve en önemli kullanıldığı yer **Euler Teoremi**'dir.

$a^{\phi(m)} \equiv 1 \pmod{m}$, öyle ki a ve m aralarında asal ise.

Fermat Teoremi de diğer bir kullanımıdır:

Eğer m asal sayı ise $a^{m-1} \equiv 1 \pmod{m}$.

Çözülebilecek Sorular

- **UVA # 10179 "Irreducible Basic Fractions" [Difficulty: Low]**
  - http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1120
- **UVA # 10299 "Relatives" [Difficulty: Low]**
  - http://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1240
- **UVA # 11327 "Enumerating Rational Numbers" [Difficulty: Medium]**
  - http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2302
- **TIMUS # 1673 "Admission to the exam" [Difficulty: High]**

- http://acm.timus.ru/problem.aspx?space=1&num=1673

Euclid'in GCD(EBOB) algoritması

Verilen iki adet negatif olmayan a ve b tam sayıları için en büyük ortak böleni bulmaya yarayan algorimadır. EBOB ingilizce'de "greatest common divisor" şeklinde adlandırılmaktadır.

$$\gcd(a,b) = \max_{k=1...\infty \ : \ k|a \ \& \ k|b} k$$

Buradaki "$|$" sembolü bölen manasını gelmektedir.

Verilen sayılardan bir tanesi 0 ise gcd diğer sayı olacaktır. Eğer verilen sayıların ikisi de 0 ise gcd tanımlı değildir.

Euclid'in algoritması: Aşağıda bahsedilecek algoritma verilen iki tam sayının gcd'sini bulma işlemini a ve b değerleri için $O(\log \min(a,b))$ işlemde bulacaktır.

Algoritma

Algoritma uygulanması bakımından aşırı derecede basittir şimdi matematiksel gösterimine bakalım:

$$\gcd(a,b) = \begin{cases} a, & \text{if b=0} \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Kodlama

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

Koşullu operatör kullanarak bu kodu çok daha kısa halde yazabiliriz.

```
int gcd (int a, int b) {
      return b ? gcd (b, a % b) : a;
}
```

Son olarak recursive olmayan halini verelim:

```
int gcd (int a, int b) {
      while (b) {
```

```
            a %= b;
            swap (a, b);
        }
        return a;
}
```

Doğruluğunun ispatı

Öncelikle, her işlem sırasında metodun 2. paramateresi kesinlikle azalmaktadır. ve ilerleyen adımlarda metodun sonlanacağı kesindir.

İspatı yapabilmek için şimdi şunu göstermemiz gerekmektedir: $\gcd(a, b) = \gcd(b, a \bmod b)$ öyle ki her $a \geq 0, b > 0$ değerleri için.

Sağdaki parametrenin soldaki sayıya bölümünden kalanın ve soldaki sayının sağdakine bölümünden kalanın ayrı ayrı gcd(a,b) ile bölündüğünü göstermemiz gerekmektedir. Ancak her ikisi için de ispat aynı olacağı için biz sadece bir defa yapacağız.

$d = \gcd(a, b)$ kabul edelim. Tanım gereği $d|a$ ve $d|b$'dir.

Şimdi a'nın b'ye bölümünden kalanını daha açık bir ifadeyle yazalım.

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Diğer adım için ise $d \mid (a \bmod b)$ bu da doğru olmalıdır.

Şimdi bulduğumuz yeni durumu $d|b$ ile birlikte değerlendirelim(Zaten bir sonraki adımda parametre olarak b ve a mod b değişkenleri gönderilecekti.)

$$\begin{cases} d \mid b, \\ d \mid (a \bmod b) \end{cases}$$

Şimdi basit bir gözlem yapacağız. Herhangi p, q, r tam sayıları için $p|q$ ve $p|r$ ise $p \mid \gcd(q, r)$ 'dir. Buradan şunu elde edeceğiz:

$d \mid \gcd(b, a \bmod b)$ son durumda temel olarak şunu bulacağız:
$\gcd(a, b) \mid \gcd(b, a \bmod b)$.

Çalışma Zamanı

Algoritmanın çalışma zamanı Lame Teorem ile değerlendirilmiştir ve Euclid Algoritması ile Fibonacci sayıları arasında bir ilişki keşfedilmiştir.

Eğer $a > b \geq 1$ ve $b < F_n$ bir n değeri için, $n-2$ adet recursive işlemden fazla yapılmayacaktır.

Kısaca $a = F_n, b = F_{n-1}$ ise toplamda $n-2$ recursive işlem yapılacaktır. Diğer bir deyişle Euclid Algoritması için en kötü durum ardışık Fibonacci değerleridir.

Fibonacci sayıları exponantional büyümektedir(yani 2'nin üstel büyümesiyle yaklaşık aynı hıza sahiptir). Buradan da anlaşılacağı üzere bu algoritmanın çalışma süresi yaklaşık olarak $O(\log \min(a,b))$ işlemdir.

LCM(Least common multiple/En küçük ortak kat)

LCM'nin en kolay hesaplanma yöntemi GCD ile ilişkilidir.

$$\operatorname{lcm}(a,b) = \frac{a \cdot b}{\gcd(a,b)}$$

Bunu da yine Euclid'in Algoritmasıyla hesaplanabilir demektir.

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

Geliştirilmiş Euclid Algoritması

Euclid'in normal GCD algoritması kolayca verilen a ve b sayılarının gcd'sini bulmak için kullanılırken, Geliştirilmiş Euclid Algoritması normal olana ek olarak x, y kat sayıları da bulmaktadır öyle ki:

$$a \cdot x + b \cdot y = \gcd(a,b).$$

Bu şekilde a ve b sayılarının gcd' sini kendileri cinsinden gösterecek bir sonuc bulunmuş olacaktır.

Algoritma

Euclid algoritmasının hesaplanması (a,b) çiftini (b%a,a) şekline çevirerek hesaplayacak kadar kolaydır.

Şimdi (a%b) için x_1 ve y_1 ikilisi bulalım öyle ki:

$$(b\%a) \cdot x_1 + a \cdot y_1 = g,$$

Ve biz x ve y ikilisini bulmak istiyoruz verilen (a,b) ikilisi için

$$a \cdot x + b \cdot y = g.$$

Bunu yapabilmek için b%a değerini değiştirelim:

$$b\%a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a.$$

Şimdi bu değeri ilk denkleme yerleştirelim ve :

$$g = (b\%a) \cdot x_1 + a \cdot y_1 = \left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right) \cdot x_1 + a \cdot y_1,$$

Eğer kat sayıları düzenlersek şunu elde ederiz:

$$g = b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right).$$

Bunları da aradığımız x ve y kat sayılarıyla eşleştirirsek şu açılımı elde ederiz:

$$\begin{cases} x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1, \\ y = x_1. \end{cases}$$

Kodlama

```
int gcd (int a, int b, int & x, int & y) {
      if (a == 0) {
            x = 0; y = 1;
            return b;
      }
      int x1, y1;
      int d = gcd (b%a, a, x1, y1);
      x = y1 - (b / a) * x1;
      y = x1;
      return d;
}
```

Bu parametre olarak verilen x ve y ikilisini ayarlarken bir yandan da gcd'yi bulan recursive bir fonksiyondur.

Base case olarak a=0 belirlenmelidir. Base durumda gcd b olacaktır ve kat sayıları da x=0 ve y=1 olarak belirleyebiliriz. Diğer bütün durumlarda belirtilen çözüm geçerli olacaktır. Ve kat sayılar yenilerini oluşturmak için kullanılacaktır.

Yukarıda verilen kod parçası aynı zamanda negatif sayılar için de çalışmaktadır.

Çin Kalan Teoremi

p_i'ler aralarında asal olmak üzere, $p = p_1 \cdot p_2 \cdot \ldots \cdot p_k$.

Herhangi bir $(0 \leq a < p)$ sayısı ile $(a_1, \ldots, a_k)$ dizisi arasında ilişkiyi inceleyelim:

$a_i \equiv a \pmod{p_i}$ Her i için bu denklik sağlanıyorsa a ile $(a_1, \ldots, a_k)$ dizisi arasında bire bir ilişki var demektir ve bu durum aşağıdaki şekide gösterilir:

$$a \Longleftrightarrow (a_1, \ldots, a_k).$$

Bunun bir sonucu a üzerine uygulanan bağımsız değişikliklerin aynısı diziye uygulandığında denklik sağlanmaya devam eder.

Daha matematiksel olarak ifade edecek olursak.

$$a \Longleftrightarrow \Big(a_1, \ldots, a_k\Big),$$

$$b \Longleftrightarrow \Big(b_1, \ldots, b_k\Big),$$

Eğer yukarıdaki iki ifade doğru ise aşağıdakiler de doğrudur.

$$(a + b) \pmod{p} \Longleftrightarrow \Big((a_1 + b_1) \pmod{p_1}, \ldots, (a_k + b_k) \pmod{p_k}\Big),$$

$$(a - b) \pmod{p} \Longleftrightarrow \Big((a_1 - b_1) \pmod{p_1}, \ldots, (a_k - b_k) \pmod{p_k}\Big),$$

$$(a \cdot b) \pmod{p} \Longleftrightarrow \Big((a_1 \cdot b_1) \pmod{p_1}, \ldots, (a_k \cdot b_k) \pmod{p_k}\Big).$$

Sonuç 1

Aşağıdaki modüler denklikler sisteminin p modunda tam olarak 1 tane çözümü vardır:

$$\begin{cases} x \equiv a_1 \pmod{p_1}, \\ \ldots, \\ x \equiv a_k \pmod{p_k} \end{cases}$$

Sonuç 2

$$x \equiv a \pmod{p}$$

Yukarıdaki denklik aşağıdaki denklikler sistemine eşdeğerdir.

$$\begin{cases} x \equiv a \pmod{p_1}, \\ \dots, \\ x \equiv a \pmod{p_k} \end{cases}$$

Algoritma

$$a_i \equiv a \pmod{p_i}.$$

a'yı aşağıdaki şekilde ifade edebiliriz.

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1},$$

Her ikili için birbirlerine göre modda terslerini önceden hesaplamalıyız.

$$r_{ij} = (p_i)^{-1} \pmod{p_j}.$$

İlk terim hariç bütün terimlerin kat sayılarında p1 çarpanı olduğu için.

$$a_1 \equiv x_1.$$

x1 kullanarak x2 aşağıdaki gibi bulunabilir.

$$a_2 \equiv x_1 + x_2 \cdot p_1 \pmod{p_2}.$$

$$a_2 - x_1 \equiv x_2 \cdot p_1 \pmod{p_2};$$

$$(a_2 - x_1) \cdot r_{12} \equiv x_2 \pmod{p_2};$$

$$x_2 \equiv (a_2 - x_1) \cdot r_{12} \pmod{p_2}.$$

Benzer bir şekilde x3, x1 ve x2 kullanılarak hesaplanır.

$$a_3 \equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3};$$

$$(a_3 - x_1) \cdot r_{13} \equiv x_2 + x_3 \cdot p_2 \pmod{p_3};$$

$$((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \equiv x_3 \pmod{p_3};$$

$$x_3 \equiv ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \pmod{p_3}.$$

Bu hesaplamalar genellenerek kolayca aşağıdaki kod haline getirilebilir.

```
for (int i=0; i<k; ++i) {
      x[i] = a[i];
      for (int j=0; j<i; ++j) {
            x[i] = r[j][i] * (x[i] - x[j]);

            x[i] = x[i] % p[i];
            if (x[i] < 0)  x[i] += p[i];
      }
}
```

Bütün x değerlerini O(k^2) de bulmayı öğrendik. x leri aşağıda yerlerine koyarak a'yı hesaplayabiliriz.

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1}.$$

x'ler hesaplandıktan sonra yukarıdaki işlem için 64 bit integer yeterli olmayabilir.

Cebir

Matrisler

m tane satır ve **n** tane sütun oluşturacak biçimde dizilmiş **m*n** tane sayının oluşturduğu tabloya bir **m** × **n** matris denir. Bir **m** × **n** matris A genellikle aşağıdaki gibi gösterilir.

$$A=\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \dots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \quad \text{veya} \quad A=[a_{ij}]\ ,\ 1 \le i \le m\ ;\ 1 \le j \le n.$$

A matrisini oluşturan sayılardan her birine A nın bir girdisi(entry) denir. A matrisinin mn tane girdisi, m satır ve n sütun oluşturacak biçimde düzenlenmiştir.

A matrisinin i–inci satırında ve j–inci sütununda bulunan a_{ij} girdisine A nın i-j girdisi denir. m × n ifadesine A matrisinin büyüklüğü, m ve n sayılarına da A matrisinin boyutları denir.

Sadece bir satırdan oluşan bir matrise satır matrisi , sadece bir sütundan oluşan bir matrise sütun matrisi denir.

Bir matrisin her bir satırı bir satır matrisi, her bir sütunu da bir sütun matrisi olarak düşünülebilir.

İki Matrisin Toplamı

Büyüklükleri aynı olan iki matrisin karşılıklı girdileri toplanarak elde edilen aynı büyüklükteki matrise bu iki matrisin toplamı denir.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}+\begin{bmatrix} x & y \\ z & t \end{bmatrix}=\begin{bmatrix} a+x & b+y \\ c+z & d+t \end{bmatrix}$$

Büyüklükleri farklı olan iki matrisin toplamı tanımsızdır.

Matris toplamının birleşme ve değişme özellikleri vardır: A, B ve C büyüklükleri aynı olan matrisler ise

A+(B+C) = (A+B) + C ve A + B = B + A dır.

Bir Matrisle bir Sayıyla Çarpım

Bir matris ile bir sayıyla çarpımı, matrisin her girdisinin o sayıyla çarpılmasıyla elde edilen matristir.

$$s\begin{bmatrix} x & y \\ z & t \end{bmatrix} = \begin{bmatrix} sx & sy \\ sz & st \end{bmatrix}$$

İki Matrisin Çarpımı

Önce bir satır matrisi ile bir sütun matrisinin çarpımını tanımlayacağız. Aynı sayıda girdiye sahip olan bir satır ile bir sütunun çarpımı şöyle tanımlanır:

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a_1b_1 + a_2b_2 + \cdots + a_nb_n .$$

Sözle ifade edilirse, aynı sayıda girdiye sahip olan bir satır ile bir sütunun çarpımı, o satır ve sütunun karşılıklı girdileri çarpılarak elde edilen çarpımların toplamı olan sayıdır.

İki matrisin çarpımını tanımlarken, bir satır ile bir sütunun çarpımı tanımından yararlanacağız. A bir m×p matris ve B bir p×n matris (A nın sütun sayısı ile B nin satır sayısı aynı) ise, A ile B nin çarpımı; i-j girdisi A nın i-inci satırı ile B nin j-inci sütununun çarpımı olan m×n matristir. Bu çarpım AB ile gösterilir.

$$A = [a_{ij}] \ , \ 1 \le i \le m \ ; \ 1 \le j \le p \quad \text{ve} \quad B = [b_{ij}] \ , \ 1 \le i \le p \ ; \ 1 \le j \le n \ \text{ise,}$$

$$AB = [c_{ij}] \ , \ c_{ij} = a_{i1}\,b_{1j} + a_{i2}\,b_{2j} + \cdots + a_{ip}\,b_{pj} \ , 1 \le i \le m \ ; \ 1 \le j \le n \ .$$

Başka bir anlatımla;

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ & & \dots & \\ a_{i1} & a_{i2} & \dots & a_{ip} \\ & & \dots & \\ a_{m1} & a_{m2} & \dots & a_{mp} \end{bmatrix} \quad \text{ve} \quad B = \begin{bmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1n} \\ & \dots & & \dots & \\ b_{i1} & \dots & b_{ij} & \dots & b_{in} \\ & & \dots & & \\ b_{p1} & \dots & b_{pj} & \dots & b_{pn} \end{bmatrix}$$

Matrisleri için:

$$AB = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ & & \dots & \\ a_{i1} & a_{i2} & \dots & a_{ip} \\ & & \dots & \\ a_{m1} & a_{m2} & \dots & a_{mp} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1n} \\ & \dots & & \dots & \\ b_{i1} & \dots & b_{ij} & \dots & b_{in} \\ & & \dots & & \\ b_{p1} & \dots & b_{pj} & \dots & b_{pn} \end{bmatrix} = \begin{bmatrix} c_{11} & \dots & c_{1j} & \dots & c_{1n} \\ & \dots & & \dots & \\ c_{i1} & \dots & c_{ij} & \dots & c_{in} \\ & & \dots & & \\ c_{m1} & \dots & c_{mj} & \dots & c_{mn} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ . \ . \ . \ + a_{ip}b_{pj} \quad , 1 \le i \le m \ ; \ 1 \le j \le n \ .$$

A ve B matrislerinin çarpımı AB nin tanımlı olması için A nın sütun sayısı ile B nin satır sayısının aynı olması gerekir. Çünkü, AB çarpımının i-j girdisi, A nın i-inci satırı ile B nin j-inci sütununun çarpımı olarak tanımlanmaktadır. A nın i-inci satırında tam A nın sütun sayısı kadar, B nin j-inci sütununda da tam B nin satır sayısı kadar girdi bulunduğundan bu sayılar eşit olmalıdır ki çarpım mümkün olsun.

Binary Exponantiation

Bu metot temel olarak verilen bir sayının n. dereceden üssünü $O(\log n)$ işlemde hesaplamak için kullanılan bir yöntemdir.

Bu metodun kullanılması için gerekli olan şey çarpmanın değişme özelliğidir.Diğer bir deyişle $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ olmalıdır. Bu durum tam sayılar için zaten geçerlidir ayrıca birazdan bahsedeceğimiz matrisler için de geçerlidir.

Algoritma

Verilen bir a tam sayısı ve n çift sayısı için hesaplama işlemini şu şekilde gerçekleştirebiliriz. $a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$. Bu binary exponantiation için temel olarak kullanan yöntemdir.

Diğer bir durumda ise verilen bir a tam sayısı ve n tek sayısı için hesaplama şu şekilde olacaktır. $a^n = a^{n-1} \cdot a$

Açık bir şekilde görülmektedir ki bu işlemlerin toplam sayısı $2 \log n$'i geçmeyecektir.

Kodlama

Basit recursive bir metod:

```
int binpow (int a, int n) {
      if (n == 0)
            return 1;
```

```
        if (n % 2 == 1)
                return binpow (a, n-1) * a;
        else {
                int b = binpow (a, n/2);
                return b * b;
        }
}
```

Recursive olmayan bir metot(Bölme işlemleri bitwise operatörleriyle yapılmıştır):

```
int binpow (int a, int n) {
        int res = 1;
        while (n)
                if (n & 1) {
                        res *= a;
                        --n;
                }
                else {
                        a *= a;
                        n >>= 1;
                }
        return res;
}
```

Bu işlem daha kolay bir şekilde de kodlanabilir.

```
int binpow (int a, int n) {
        int res = 1;
        while (n) {
                if (n & 1)
                        res *= a;
                a *= a;
                n >>= 1;
        }
        return res;
}
```

Son olarak Java dilinde bu işlemler BigInteger için kodlanmıştır.

Problem ve çözüm örnekleri

Fibonacci sayılarını hesaplamanın mantıklı bir yöntemi

Soru: Verilen bir n sayısı için F_n değerinin hesaplanmasını isteniyor öyleki F_i-Fibonacci dizisi'nin i. elemanı.

Çözüm: Bu sorunun çözümü detaylı olarak Fibonacci dizisi adlı konuda anlatılmıştır. Burada çözümü özet olarak vereceğiz.

Fibonacci dizisinin hesaplanması temel olarak önceki iki elemanın değerleri ile hesaplanmaktadır. Her fibonaaci elemanı önceki iki elemanın toplamıdır. Bu demek oluyor ki (2×2)'lik bir matris oluşturabiliriz ve bu matrisi kullanarak önceki iki elemanın toplamını yeni elemana yazacak bir dönüşüm oluşturabiliriz. Oluşan matris ile F_0 , F_1 değerlerini n defa dönüşüm uygulayarak F_n ve F_{n+1} değerlerini elde edebiliriz. Yani bizim yapmamız gereken oluşturduğumuz matrisin n. dereceden üzerini bulmamızdır. Ve bunu da az önce anlattığımız yöntem ile $O(\log n)$ zamanda yapabiliriz.

Permütasyonlar'ın k. dereceden karıştırılması

Soru: Verilen bir n elemanlı p permütasyonu için birim permütasyonun(1,2,3….n) k defa yeniden düzenlenmesinin ardından son permütasyonun ne olacağı bulunmalıdır. Yeniden düzenlenme işlemi şu şekilde tanımlanmaktadır: Bir permütasyon için i. indiste bulunan eleman bir sonraki adımda P_i pozisyonunda yer alacaktır.

Çözüm: Basit olarak yine değişimi temsil edecek bir matris hazırlanabilir ve bu matrisin üzerini k defa alarak yine aynı şekilde hesaplayabiliriz. Son durumda çalışma zamanı $O(n \log k)$.

Ayrıca bu sorunun çözümü linear zamanda da yapılabilir ancak bunlar için permütasyonda bulunan cycle'lar belirlenmeli ve üzerinde işlemler yapılmalıdır.

Geometrik işlemlerin hızlı bir şekilde yapılması

Soru: Verilen n adet p_i şeklinde adlandırılmış nokta için m adet dönüşümün ardından noktaların yeni yerleri bulunmak istenmektedir. Dönüşümler; bir vektör boyunda kaydırma, ölçekleme(yani belli bir oranda noktaların hepsini orijine yaklaştırma ya da uzaklaştırma), belirli bir açıyla belirlenen bir eksen etrafında döndürme işlemlerinden bir tanesidir. Bu işlemler sırayla yapıldıktan sonra bu m işlemin dairesel olarak tekrar tekrar yapılmasının ardından noktaların yeni konumlarının belirlenmesini $O(n \cdot length)$ zamandan daha kısa sürede bulmak istiyoruz. Burada belirtilen $length$ toplamda yapılacak dönüşüm sayısıdır.

Çözüm: Her bir dönüşümün noktaların bileşenleri üzerindeki değişikliklerinin nasıl olduklarını inceleyelim:

Öteleme işlemi: Tüm bileşenlere belirli bir sabit ekleme işlemidir.

Ölçekleme işlemi: Bütün bileşenleri belirli bir sayı ile çarpmamız gerekmektedir.

Döndürme işlemi: Bileşenlerin yeni değerleri eskilerinin linear kombinasyonları şeklinde yazılmasıyla hesaplanmaktadır.

Açık bir şekilde görebiliriz ki tüm bu dönüşüm işlemleri linear denklemlerin yeniden hesaplanması yöntemidir. Yani tüm bu dönüşümler 4×4'lük bir matris ile temsil edilebilir.

$$\begin{pmatrix} a_1 1 & a_{12} & a_{13} & a_{14} \\ a_2 1 & a_{22} & a_{23} & a_{24} \\ a_3 1 & a_{32} & a_{33} & a_{34} \\ a_4 1 & a_{42} & a_{43} & a_{44} \end{pmatrix},$$

Noktaların eski koordinatlarının yazılı olduğu bir vektör ile bu matrisin çarpımı bize noktaların yeni koordinatlarının yazılı olduğu vektörü verecektir.

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} x' & y' & z' & 1 \end{pmatrix}.$$

Neden 4. boyutu temsil edecek değer için sadece 1'i kullanıyoruz? Bu değişken olmaksızın öteleme işlemini gerçekleştiremeyiz çünkü bütün koordinatlara belirlenen bir sabiti eklemeliyiz yani bu değer 1'in bir katıdır.

Artık çözüm kolaylaştı, çözüm artık bu matrislerin çarpımlarının kullanılmasından ibaret. Bütün işlemlerin tekrarlanmaları bitiminde noktaların gidecekleri yeni koordinatların hesaplanmasına olanak sağlayacak matrisi kolayca bulabiliriz. Ve bu işlemin hesaplanması için gereken süre $O(m \cdot \log repetition)$. Çarpma işlemi bittikten sonra sorgu işlemine ise sadece $O(n)$ zamanda cevap verebiliriz.

Belirlenen uzunluğa sahip farklı yolların sayısı

Soru: Verilen n node'dan oluşan G graph'ı için her i, j yol ikilileri için tam olarak k adet yoldan geçerek kaç farklı şekilde gidilebileceğini bulunuz.

Çözüm: Sorunun detaylı çözümü başka bir konuda anlatılmıştır bu yüzden burada çok az değineceğiz. Yapılacak yöntem komşuluk matrisinin k defa tekrar hesaplanmasına dayalı bir yöntemdir. Ancak bu hesaplamayı yine binary exponantiation ile hesaplayacağız. Toplam çalışma zamanı $O(n^3 \log k)$ olacaktır.

Verilen iki sayının belirli bir sayı modundaki çarpımı

Bize iki adet sayı verilmiş olsun ve bu sayıların çarpımlarının m modundaki değeri isteniyor olsun. Yani: $a \cdot b \pmod m$ bu değer isteniyor olsun. Ancak a, b ve m değerleri bilgisayardaki çarpma işlemlerine müsaade etmeyecek kadar büyük sayılar olsunlar örneğin 64bit tam sayı değerine çarpımları sığmayacak şekilde olsun. Bu durumda hesaplamayı çarpma işlemi yapmak sızın gerçekleştirebiliriz.

Çarpma işlemi yerine toplama işlemini kullanabiliriz. Bizden b defa a sayısının toplamı istenmektedir ve bu işlemi gerçekleştirirken tıpkı ilk kullandığımız binary exponantiation yöntemini kullanarak basit bir şekilde $O(\log m)$ zamanda gerçekleştirebiliriz.

Çözülebilecek sorular

- SGU # 265 **"Wizards"** [Difficulty: Medium]
  - http://acm.sgu.ru/problem.php?contest=0&problem=265

Fibonacci Sayıları

Tanımlama

Fibonacci sayıları şu şekilde tanımlanmıştır:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

İlk bir kaç fibonacci sayısı şu şekildedir:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ ,\ 55,\ 89,\ \ldots$$

Özellikleri

Fibonacci sayıları bir çok özelliğe sahiptir iste onlardan bazıları:

- Value Cassini:
  - $F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$
- Toplama kuralı:
  - $F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$
- Bir önceki kurallı kullanarak k=n için:
  - $F_{2n} = F_n(F_{n+1} + F_{n-1}).$
- Önceki eşiltikleri kullanarak tümevarım ile F_n in her zaman F_{kn} i böldüğünü gösterebiliriz.
- Ayrıca bir önceki kuralın tersi de geçerlidir yani F_n F_m'i bölüyorsa n de m i bölmektedir.
- GCD eşitliği:

- $\gcd(F_m, F_n) = F_{\gcd(m,n)}$.

Fibonacci Kodlama

Zeckendorf Theorem'i belirtmektedir ki: herhangi bir positive n tam sayısı fibonacci sayılarının toplamı şeklinde unique olarak yazılabilmektedir. Diğer bir deyişle:

$N = F_{k_1} + F_{k_2} + \ldots + F_{k_r}$

öyle ki; $k_1 \geq k_2 + 2$, $k_2 \geq k_3 + 2$, $\ldots$, $k_r \geq 2$

Bu da demektir ki herhangi bir pozitif tam sayı Fibonacci tabanında unique olarak yazılabilmektedir. Örneğin:

$9 = 8 + 1 = F_6 + F_1 = (10001)_F$,

$6 = 5 + 1 = F_5 + F_1 = (1001)_F$,

$19 = 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F$,

Greedy bir yöntemle fibonacci tabanındaki sayıyı bulacak bir algoritma geliştirebiliriz. Örneğin F_k sayısı n den küçük en büyük fibonacci sayısı olsun oluşturulacak sayının k. basamağı 1 olacaktır. Ardından n-F_k için hesaplanmaya devam edilecektir.

N. Fibonacci sayısı için genel formul

Fransız matematikçi Binet'in geliştirdiği bir formul mevcuttur bu konuda:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

Bu formul tümevarımla kolayca ispatlanabilmektedir.

Fibonacci sayıların matrix ile hesaplanma

Aşağıda verilen matrix formulunun doğruluğunun ispatı çok kolaydır.

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

Şu şekilde bir P matrixi tanımlayalım:

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

Ardından aşağıdaki eşitliği elde edebiliriz.

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}.$$

Bu yüzden n. fibonacci sayısının hesaplanması için P^n matrisinin hesaplanması gerekmektedir.

Ayrıca daha önce de gösterdiğimiz gibi bir matrisin n. dereceden üssünü $O(\log n)$ zamanda hesaplayabilmekteyiz(Binary Exponantiation). Bu da demektir ki n. fibonacci sayısı $O(\log n)$ zamanda sadece basit aritmetik işlemler kullanarak bulunabilmektedir.

Fibonacci sayılarının modüler olarak frekansı

Her F_i fibonacci sayısını P modunda inceleyelim. Şimdi bu sayıların P modunda periyodik olduğunu gösterelim.

Bunu çelişki yöntemiyle ispatlayacağız. İlk P^2+1 Fibonacci sayısını P modunda incelediğimizi düşünelim:

$$(F_1, F_2),\ (F_2, F_3),\ \ldots,\ (F_{p^2+1}, F_{p^2+2}).$$

Ancak P modunda en fazla P^2 farklı ikili olabilmektedir. Bu da demektir ki yukarıda verilen dizide en az iki tane aynı değerli ikili mevcuttur. Bu da dizinin periyodik olduğunu göstermektedir.

Şimdi bu aynı olan ikililer içinden en küçük indisli ikiliyi seçelim ve ikili sırasıyla şöyle olsun: (F_a, F_{a+1}) ve (F_b, F_{b+1}). Bu şartlar altında $a = 1$.

Kombinatorik

Binom Katsayıları

Binom katsayısı C_n^k n adet elemanlı bir kümeden hangi sırada seçildiğinden bağımsız olmak üzere kac farklı şekilde k farklı eleman seçilebileceğini belirtir.

Ayrıca binom kat sayıları $(a+b)^n$'nin açılımında bulunur:

$$(a+b)^n = C_n^0 a^n + C_n^1 a^{n-1} b + C_n^2 a^{n-2} b^2 + \ldots + C_n^k a^{n-k} b^k + \ldots + C_n^n b^n$$

Calculation

Analitik olarak hesaplanması

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Bu formülü permutasyon(n, k)'nı kullanarak kolayca cıkarabilirsiniz.

Özyinelemeli Formül

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

Özellikler

Binom katsayıları bir çok özelliğe sahip, biz sadece basit birkaç tanesine değineceğiz:

- $C_n^k = C_n^{n-k}$:
- $C_n^k = \frac{n}{k} C_{n-1}^{k-1}$
- $\sum_{k=0}^{n} C_n^k = 2^n$
- $\sum_{m=0}^{n} C_m^k = C_{n+1}^{k+1}$
- $\sum_{k=0}^{m} C_{n+k}^k = C_{n+m+1}^m$
- $(C_n^0)^2 + (C_n^1)^2 + \ldots + (C_n^n)^2 = C_{2n}^n$

- $1C_n^1 + 2C_n^2 + \ldots + nC_n^n = n2^{n-1}$
- $C_n^0 + C_{n-1}^1 + \ldots + C_{n-k}^k + \ldots + C_0^n = F_{n+1}$

Program ile Hesaplanması

Doğrudan analitik formülün hesaplanması

Bu yöntemle hesaplama yaparken overflow olma ihtimali yüksek olduğu için bu yöntem genellikle daha küçük n ve k degerleri için tercih edilir.

```
int C (int n, int k) {
      int res = 1;
      for (int i=n-k+1; i<=n; ++i)
            res *= i;
      for (int i=2; i<=k; ++i)
            res /= i;
}
```

Geliştirilmiş Kod

Yukarıdaki yöntemde ilk döngüde res degişkeni sürekli büyüdüğü için overflow ihtimali daha da artıyor. Bunun yerine aşagıdaki gibi k tane kesirin çarpımını kullanarak overflow ihtimalini düşürebiriz.

```
int C (int n, int k) {
      double res = 1;
      for (int i=1; i<=k; ++i)
            res = res * (n-k+i) / i;
      return (int) (res + 0.01);
}
```

Cevaba 0.01 eklenmesinin sebebi double'dan kaynaklanabilecek hatanın önüne geçmek.

Paskal Üçgeni

Özyinelemeli formül kullanılarak istenen sınırlardaki bütün binom katsayıları bulunabilir.

```
const int maxn = ...;
int C[maxn+1][maxn+1];
for (int n=0; n<=maxn; ++n) {
```

```
        C[n][0] = C[n][n] = 1;
        for (int k=1; k<n; ++k)
             C[n][k] = C[n-1][k-1] + C[n-1][k];
}
```

Bütün değerlere ulaşmak gerekli değilse sadece son 2 satır kullanılarak da hesaplanabilir.

Sabit Zamanda Hesaplama

Faktoriel degerleri önceden hesaplanıp, analitik formül O(1) zaman karmaşıklığında hesaplanabilir.

İçerme Dışarma Prensibi

İçerme dışarma prensibi herhangi bir kümenin eleman sayısını hesaplamaya yarayan önemli bir tekniktir.

Birden fazla kümenin birleşiminin eleman sayısını hesaplamaya çalışalım. Bunun için öncelikle her kümenin eleman sayısını toplarız, ikili kesişimleri çıkartırız, üçlü kesişimleri ekleriz ….

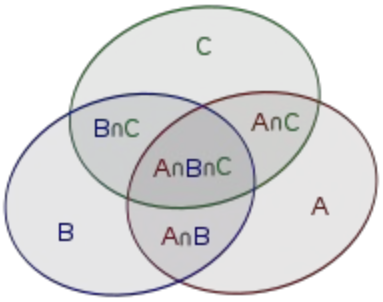
Matematiksel bir şekilde ifade edersek:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{\substack{i,j:\\ 1\le i<j\le n}} |A_i \cap A_j| + \sum_{\substack{i,j,k:\\ 1\le i<j<k\le n}} |A_i \cap A_j \cap A_k| - \ldots + (-1)^{n-1} |A_1 \cap \ldots \cap A_n|.$$

Daha kapalı bir şekilde yazacak olursak, B A_i'lerden oluşan bir küme olmak üzere.

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{C \subseteq B} (-1)^{size(C)-1} \left|\bigcap_{e \in C} e\right|.$$

Venn Diyagramı ile Gösterimi



$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C).$$

Olasılık Teorisine Yönelik İfade Şekli

$\mathcal{P}(A_i)$ A_i'durumunun ihtimalini belirtir.

$$\mathcal{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} \mathcal{P}(A_i) - \sum_{\substack{i,j: \\ 1 \le i < j \le n}} \mathcal{P}(A_i \cap A_j) + \sum_{\substack{i,j,k: \\ 1 \le i < j < k \le n}} \mathcal{P}(A_i \cap A_j \cap A_k) - \ldots + (-1)^{n-1} \mathcal{P}(A_1 \cap \ldots \cap A_n).$$

İçerme Dışarmanın İspatı

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{C \subseteq B} (-1)^{size(C)-1} \left|\bigcap_{e \in C} e\right|,$$

Her elemanın tam olarak bir defa sayıldığını ispatlamamız lazım.

Her hangi bir x elemanını ele alalım öyle ki tam olarak k tane kümede bulunsun.

- $size(C) = 1$ ise x tam olarak k defa pozitif kat sayı ile sayılır;
- ise defa negatif kat sayı ile sayılır;
- ise defa pozitif kat sayı ile sayılır;
-
- ise defa, kat sayısı ile sayılır;

Aşağıdaki toplamı hesaplamamız gerekiyor:

nın açılımı şeklinde yazalım:

için 'ye eşitir. Dolayısı ile.

Örnekler

Basit Permütasyon Sorusu

10 elemanlı kaç farklı permütasyon vardır öyle ki ilk elemanı 1 den büyük ve son elemanı 8 ‘den küçük olsun.

Bunun yerine ilk elemanı 1 den küçük eşit ve son elemanı 8 den büyük eşit olanları sayıp bütün durumlardan çıkaralım.

X ilk elemanı 1 den küçük eşit ve Y son elemanı 8 den büyük eşit olan permütasyonların sayısı olmak üzere aşağıdaki hesaplamayı yapmalıyız.

Bunu 10! den çıkarıp sonucu elde edebiliriz.

Verilen denklemin tam sayı çözümleri

Verilen denklem;

ayrıca şu koşullar geçerlidir:

,

Bu denklem için kaç farklı çözüm olduğu bulunmak istenmektedir.

İlk olarak, sayılar için belirlenmiş olan kısıtları boş verelim ve böyle bir denklem verildiğinde kaç farklı çözüm elde edilebileceğini bulalım. Bu kolay bir şekilde binom kat sayılarıyla yapılabilir. Bunu 20 topu 6 kutuya atmak olarak nitelendirebiliriz. Ve şu şekilde hesaplanır:

Şimdi içerme dışarma prensibini kullanarak saydığımız ama saymamamız gereken çözümleri çıkaralım. Yani herhangi bir x_i değeri arıyoruz en az 9 olacak şekilde.

Şimdi bu durumları saymak için bir sayının değerini 9 yaptığımızı ve eşitliğin sağ tarafını 20 yerine 11 olarak değerlendirelim. Şimdi yine üstteki gibi verilen denkleme eşitliğin sağ tarafı 11 olacak şekilde çözümü bulalım ve bu çözüm olsun.

olacaktır.

Ve şimdi de iki farklı sayıyı bu şekilde değerlendirdiğimizi düşünelim yani iki farklı sayının 9 veya üzeri olduğunu. Bu şartlar altında bizim bulmaya çalıştığımız şey elimizdeki ve durumları için kesişimlerini bulmaktır.

Bu kesişimlerin sayısı en fazla 2 olabilmektedir çünkü üç veya daha fazla olma şartları altında eşitliğin sağ tarafının 20 den büyük olması gerekmektedir.

Bu durumların hepsini içerme dışarma prensibi altında birleştirirsek şu durumu elde ederiz:

Örnek Sorular

- UVA # 10325 **"The Lottery"** [Difficulty: Low]
- UVA # 11806 **"Cheerleaders"** [Difficulty: Low]
- TopCoder SRM 477 **"CarelessSecretary"** [Difficulty: Low]
- TopCoder TCHS 16 **"Divisibility"** [Difficulty: Low]
- SPOJ # 6285 NGM2 **"Another Game With Numbers"** [Difficulty: Low]
- TopCoder SRM 382 **"CharmingTicketsEasy"** [Difficulty: Medium]
- TopCoder SRM 390 **"SetOfPatterns"** [Difficulty: Medium]
- TopCoder SRM 176 **"Deranged"** [Difficulty: Medium]
- TopCoder SRM 457 **"TheHexagonsDivOne"** [Difficulty: Medium]
- SPOJ # 4191 MSKYCODE **"Sky Code"** [Difficulty: Medium]
- SPOJ # 4168 SQFREE **"Square-free integers"** [Difficulty: Medium]
- CodeChef **"Count Relations"** [Difficulty: Medium]

Catalan Sayıları

Catalan sayıları - belirli bir forma sahip bir sayı dizisizdir. Ancak onların şaşırtan yanı çok fazla sayıda kombinatorik probleminde karşımıza çıkıyor olmasıdır.

Bu sayılara bu isim 19. yüzyılda yaşamış Belçikalı matematikçi Eugène Charles Catalana'dan sonra verilmiştir. Ayrıca böyle bir isme ve tarihe sahip olmasının yanında Euler de bu sayıları biliyordu.

Not: Burada Catalan sayılarının ispatından bahsetmeyeceğiz ancak ne tarz problemlerde karşımıza çıktığından ve nasıl hesaplandığından bahsedeceğiz.

Dizi

İlk bir kaç C_n(n=0'dan başlayarak) catalan sayısı:

Catalan sayıları çok fazla kombinatorik probleminin çözümü olarak karşımıza çıkmaktadır. n. Catalan sayısı aynı zamanda eşittir:

- n açma ve n kapama paranteziyle yapılabilecek doğru parantezleme sayısı.
- n+1 yapraktan oluşan binary(ikili) ağaç sayısı(root yaprak olarak sayılmaz.).
- n+2 kenarlı bir konveks çokgenin, köşegenler aracılığıyla kaç üçgene bölünebileceğini,
- Birbirinin izomorfiği olmayacak şekilde n iç node'a sahip binary tree lerin sayısı.
- n x n'lik matriste (0,0) noktasından (n,n) noktasına ana diyagonalin üzerine çıkmadan sağa ve üste giderek kaç farklı şekilde gidilebileceğinin sayısı.
- 1'den n'e kadar olan sayıların, sıralı üçlü oluşturmamak koşuluyla kaç farklı şekilde dizilebileceğini,
- n basamaklı bir merdivenin, n tane karoyla kaç farklı şekilde kaplanabileceğini,
- {1,…,n} kümesinin kesişmeyen kısımlarının sayısını,

Hesaplama

İki farklı hesaplama metodu geliştirilmiştir catalan sayıları için ilki recursive, ikincisi de analitik formüldür. Öte yandan biz yukarıdaki bütün problemlerin birbiriyle aynı olduğunu biliyoruz.

Recursive Formül

Bu formül doğru parantezleme probleminden türetilebilir.

Analitik Formül

Bu formülde n'in k'lı kombinasyonu demektir.

Bu formül n x n matriste diyagonalin üstüne çıkmadan kaç farklı şekilde gidilebileceğini hesaplama probleminden türetilebilmektedir.

Genel Matematik

Binary Search

Binary search monotonik(sürekli artan veya sürekli azalan) fonksiyonlar üzerinde O(logN) işlemde arama yapmak için kullanılır.

2 kişi(A ve B) arasında oynanan şöyle bir oyun tanımlayalım: A kişisi N'den küçük bir sayı tutarak oyuna başlıyor. Daha sonra her adımda B kişisi tahminde bulunuyor ve A kişisi B'nin tahmin ettiği sayının aklında tutuğu sayıya eşit, küçük veya büyük olduğunu söyleyerek cevap veriyor.

B oyuncusunun yerinde olsaydınız aranan sayının bulunması mümkün olan aralığı her adımda ikiye bölerek O(logN) işlemde oyunun biteceğini garanti edebilirdiniz.

Bu oyun stratejisinin bir örneğini inceleyelim:

N = 10, A'nın aklından tutuğu sayı 7 olsun.

1. B'nin tahmini = 5, A'nın cevabı = küçük. Mümkün aralık = [1, 10]
2. B'nin tahmini = 8, A'nın cevabı = büyük. Mümkün aralık = [6, 10]
3. B'nin tahmini = 6, A'nın cevabı = küçük. Mümkün aralık = [6, 7]
4. B'nin tahmini = 7, A'nın cevabı = eşit. Mümkün aralık = [7, 7]

Gördüğünüz üzere B oyuncusu aralığı sürekli ikiye bölerek log(N) işlemde oyunun bitmesini sağladı. B oyuncusunun kullandığı bu yönteme *binary search* denir.

İmplementation

```
   // vec'in monoton olduğu garanti edilmek üzere
int binary_search(vector<int> vec, int start, int end, int key){
   // Aralıkta cevap olması mümkün olan hiçbir sayı yoksa
   if(start > end)
      return -1;
   // vector'ün ortadaki elemanını bulup ikiye bölelim
   int middle = start + ((end - start) / 2);
```

```
    // cevabı bulduk
    if(vec[middle] == key)
        return middle;
    // ortanın gerisinde tarafında aramaya devam etmeliyiz
    else if(vec[middle] > key)
        return binary_search(vec, start, middle - 1, key);
    // ortanın ilerisinde tarafında aramaya devam etmeliyiz
    return binary_search(vec, middle + 1, end, key);
}
```

Ternary Search(üçlü arama)

Problemin Tanımı

f(x) [l,r] aralığında tanımlanmış unimodal bir fonksiyondur. Unimodal fonksiyonlar şu iki şekilden biri şeklinde olacaktır. İlki, başlangıçta sadece artan bir fonksiyon gibi davranıp maksimum noktaya ulaştıktan sonra sadece azalan fonksiyon gibi davranacaktır. İkinci, ilk durumun simetriğidir yani ilk olarak azalan bir fonksiyon gibi davranıp minimum noktaya vardıktan sonra sadece artan fonksiyon gibi davranacaktır.

Bulmamız gereken ise verilen f(x) fonksiyonu için tanımlı olduğu aralıkta max ya da min noktayı bulmaktır.

Algoritma

İlk olarak verilen f fonksiyonu yukarıda tanımlanan ilk fonksiyon gibi olsun.

Varsayalım m_1 ve m_2 belirlenen aralıkta seçilmiş iki nokta olsun şartını sağlayan. Şimdi $f(m_1)$ ve $f(m_2)$ değerlerini hesaplayalım. Ardından elimizde 3 seçenek oluşacaktır:

- Eğer $f(m_1) < f(m_2)$ ise bu şartlar altında aranan maksimum nokta sol tarafta yer alamaz yani [l,m1] aralığında. Bu da demektir ki bulmak istediğimiz maksimum nokta sadece (m1,r] aralığında bulunabilir. Aramaya sadece bu aralık ile devam etmek en mantıklısıdır.
- Eğer $f(m_1) > f(m_2)$ ise bu şartlar altında aranan maksimum nokta sağ tarafta yer alamaz yani [m2,r] aralığında. Bu da demektir ki bulmak istediğimiz maksimum nokta sadece [l,m2) aralığında bulunabilir. Aramaya sadece bu aralık ile devam etmek en mantıklısıdır.
- Eğer $f(m_1) = f(m_2)$ ise aranan maksimum nokta [m1,m2] kapalı aralığındadır çünkü böyle bir durumu ancak m_1 artan kolda ve m_2 azalan kolda olduğunda elde edebiliriz. Bu durum için de herhangi özel bir şey yapmaksızın ilk seçeneğin içinde ya da ikinci seçeneğin içinde değerlendirebiliriz.

Buradan çıkaracağımız sonuç ile bulunulan her [l,r] aralığı için daima daha dar bir [l',r'] aralığı elde edebilmekteyiz. Bu işlemi aranan maksimum nokta bulunana kadar arama aralığını daraltarak tekrarlayabiliriz.

Bu arama işleminde aranan aralığın uzunluğu işlem devam ettikçe başta soruda belirtilen bir hassaslık değerine ulaşılana kadar devam ettirilir. Böyle bir hassaslık değerine ulaştıktan sonra aranan sonuca varılmış demektir.

Kısaca yapmak istediğimiz şey verilen bir [l,r] aralığını m_1 , m_2 sayılarıyla birlikte üç eşit parçaya böleceğiz ve ardından aradığımız sayının bulunmayacağı garanti olan bir parçayı bunların içinden ayırıp kalan aralıklar için arama işlemine devam edeceğiz.

m_1 ve m_2 sayılarının belirlenmesi:

Tam sayılar için arama

Bu durumda da yine algoritmada bir değişiklik olmayacaktır. Verilen bir [l,r] aralığı için yine aralığı üç eşit parçaya bölmeye çalışacağız. Bunun için yine yukarıda kullandığımız şekilde m_1 ve m_2 sayıları seçmeye çalışacağız ve bu sayıları aralığı neredeyse üçe bölecek şekilde seçmemiz gerekmektedir.

İkinci farklı nokta ise, bitirme durumu: çalıştıracağımız algoritmanın bitirme koşulu olacaktır. Çünkü bu durumda daha fazla m_1 ve m_2 sayılarını seçmek mümkün olmayacaktır. Ayrıca bu durumun değerlendirilmemesi sonsuz döngülere sokabilir algoritmayı. koşulunda algoritma sonlandığında kontrol edilmemiş sayılarını tek tek kontrol ederek bunların içinden maksimum olanı seçebiliriz.

Kodlama

Verilen kod reel sayılar için çalışmaktadır:

```
double l = ..., r = ..., EPS = ...; // ilk atamalar
while (r - l > EPS) {
   double m1 = l + (r - l) / 3,
      m2 = r - (r - l) / 3;
   if (f (m1) < f (m2))
      l = m1;
   else
      r = m2;
}
```

Burada verilen EPS değeri mutlak hata değerini temsil etmektedir. Ve bu değer soruda verilen hassalık değerine göre belirlenmektedir. Bunun yerine biz işlemi belirli bir sayıda da gerçekleştirebiliriz:

```
for (int it=0; it<iterations; ++it)
```

Bu durumda da iterations sayısı kadar işlem yapıldığı takdirde elde edilmek istenen hassaslık değerini elde ediyor olmayı garantilememiz gerekmektedir.