Упражнение №2 по САА

Прости и съвършени числа.

Простите числа се разглеждат в математиката още от дълбока древност и са свързани с много интересни задачи далеч извън нейните предели. В информатиката те намират приложение в криптирането, архивирането и т.н.

Дефиниция 1.14. Едно естествено число се нарича *просто*, ако няма други делители освен 1 и себе си, като числото 1 не се счита за просто. Ако не е просто, то се нарича *съставно*.

Редицата на простите числа започва така:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, …

Както вече споменахме, едно от приложенията на простите числа е за криптиране на данни, които се предават през “несигурна” мрежа (например финансови транзакции в Интернет). Някои алгоритми за кодиране на предаваната информация (например *RSA* [*RSA-78*]) използват произведение на големи прости числа. За да бъде “разбит” такъв канал за предаване на информация, трябва да бъдат известни *самите* прости числа, а не само тяхното *произведение*. Ако разгледаме числото 55, веднага можем да се досетим как се разлага то — като произведение на 5 и 11. За числото 4853 трудно бихме възстановили на ум простите му делители (211 и 23), но бихме могли да съставим програма, която ги намира. В настоящия момент са известни много големи прости числа — ако умножим например две произволни 100 цифрени прости числа и разполагаме само с полученото произведение, то възстановяването на числата, по който и да е алгоритъм, ще отнеме неприемливо дълго време (например, многократно повече от времето, необходимо за приключването на една банкова транзакция).

Когато разглеждаме редицата на простите числа, възникват някои въпроси:

- Колко прости числа има в даден интервал $[a, b]$?
- Каква част от безкрайността представляват простите числа?
- Съществува ли формула за намиране на n-тото поред просто число?

Ако с $\pi(x)$ означим броя на всички прости числа, ненадминаващи дадено естествено число x, то намирането на точна формула за изчисляването на $\pi(x)$ ще даде отговор на горните три въпроса. За съжаление такава формула все още няма (и е малко вероятно да се намери — *виж 6.2.*), но съществуват някои формули за апроксимиране на $\pi(x)$, например (нека да си припомним, че $\ln x \equiv \log_e x$):

Теорема (*за простите числа*) $\pi(x) \cong x \,/\, ln(x - a)$, където a е произволна положителна константа, по-малка от x.

Най-добро приближение се постига при $a = 1$.

Следствие. *n*-тото просто число е приблизително [*n*.*ln*(*n*)]. По-добро приближение се постига с [*n*(*ln*(*n*) + *ln*(*ln n* – 1))].

Следствие. Вероятността едно число *x* да бъде просто е приблизително 1/*ln*(*x*).

Проверка дали дадено число е просто

Един очевиден алгоритъм, пряко следствие от дефиницията, е следният: проверяваме дали всяко число от интервала [2, p/2 –1] дели p и, ако намерим такова, следва, че p е съставно.

Лесно може да се съобрази, че е излишно да изследваме всички числа до p/2 –1: достатъчно е да проверим за делимост само до $\sqrt{p}$ (включително). Това е така, тъй като винаги, когато p има делител x, x > $\sqrt{p}$, то следва, че p се представя във вида p = x.y, y < $\sqrt{p}$, т.е. има и делител по-малък от $\sqrt{p}$.

```
char isPrime(unsigned n)
/* връща 1, ако е просто, и 0 – при съставно */
{ unsigned i = 2;
  if (n == 2) return 1;
  while (i <= sqrt(n)) {
    if (n % i == 0) return 0;
    i++;
  }
  return 1;
}
```

Можем да разширим още последния резултат: за да установим, че p е просто, е достатъчно да сме сигурни, че не се дели на нито едно друго просто число от интервала [2, $\sqrt{p}$]. Така, ако разполагаме с първите k прости числа (да ги означим с P_i, за i = 1,2,…,k), ще можем да проверяваме дали произволно число от интервала [2, $(P_k)^2$] е просто.

```
/* брой прости числа, с които разполагаме – изчислени
предварително */
#define K 25
unsigned prime[K] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

/* проверяваме дали число е просто, като проверяваме дали има
делители сред числата от масива prime[] */
char checkprime(unsigned n)
{ unsigned i = 0;
  while (i < K && prime[i] * prime[i] <= n) {
  if (n % prime[i] == 0) return 0;
    i++;
  }
  return 1;
}
```

Решето на Ератостен....

Съществува и още по-ефективен алгоритъм за търсене на всички прости числа в интервал. При него не е необходима памет (масив sieve[N]) с големината на интервала, както не е необходимо и на всяка стъпка да се обхожда целият интервал.

При реализацията на търсене с масив `prime[]`, съдържащ първите k прости числа, и с негова помощ проверявахме дали едно число от интервала $[2, (P_k)^2]$ е просто. Сега ще приложим следната модифицирана схема: ще започнем от празен списък, който ще попълваме последователно. Например, поставяйки в него първото просто число 2 можем да намерим всички прости числа в интервала $[3, 2^2]$ – такова е 3 и го добавяме в списъка. По-нататък, разполагайки вече с простите числа до 3, можем да намерим всички прости числа в интервала $[4, 3^2]$ — това са 5 и 7, които също добавяме в списъка. Продължаваме този процес, докато в prime[] се добавят достатъчно прости числа, за да покрият проверката дали всяко число от интервала $[2, n]$ е просто. Следва примерна реализация:

```
#include <stdio.h>
#define MAXN 10000
/* Намира простите числа до n */
const unsigned n = 500;
unsigned primes[MAXN], pN = 0;

char isPrime(unsigned n)
{ unsigned i = 0;
  while (i < pN && primes[i] * primes[i] <= n) {
    if (n % primes[i] == 0) return 0;
      i++;
    }
```

```
  return 1;
}

void findPrimes(unsigned n)
{ unsigned i = 2;
  while (i < n) {
    if (isPrime(i)) {
      primes[pN] = i;
      pN++;
      printf("%5u", i);
    }
    i++;
  }
}

int main(void) {
  findPrimes(n);
  printf("\n");
  return 0;
}
```

Най-голям общ делител, най-малко общо кратно

Най-голям общ делител (НОД)

Дефиниция 1. Дадени са две естествени числа a и b. Казваме, че d е най-голям общ делител (НОД) на a и b, ако е максималното естествено число, което дели едновременно a и b.

Дефиниция 2. Ако *НОД* на две цели положителни числа е равен на 1, то числата се наричат *взаимно прости*.

За най-голям общ делител на повече от две числа е валидна следната формула:

$$НОД(a_1, a_2, ..., a_n) = НОД(НОД(a_1, a_2, ..., a_{n-1}), a_n)$$

Ще разгледаме два алгоритъма за намиране на $НОД(a,b)$, носещи името на Евклид. Първият от тях се нарича *алгоритъм на Евклид с изваждане*:

1) Ако $a > b$, се изпълнява 4), в противен случай се изпълнява 2).

2) Ако $a = b$ следва, че сме намерили $НОД(a,b)$ — това е стойността на b и приключваме. Ако $a \neq b$ изпълняваме 3).

3) Присвояваме b = b – a и се връщаме на стъпка 1).

4) Присвояваме a = a – b и се връщаме на стъпка 1).

Няма да се спираме по-подробно върху този алгоритъм, тъй като следващият, който ще

разгледаме, е доста по-ефективен. Нарича се алгоритъмът на Евклид с деление (за него вече стана дума и в увода на книгата). Основава се на следното:

$$НОД(a, b) = НОД(b, a \% b)$$

Итеративна реализация на алгоритъма:

```
unsigned gcd(unsigned a, unsigned b)
{ unsigned swap;
  while (b > 0) {
    swap = b;
    b = a % b;
    a = swap;
  }
  return a;
}
```

Най-малко общо кратно (НОК)

Дефиниция 3. Дадени са две цели числа a и b. Минималното естествено число d ($d > 0$) такова, че $a|d$ и $b|d$ се нарича *най-малко общо кратно* (*НОК*) на a и b.

Когато търсим *НОК* на повече от две числа е налице аналогична зависимост, както и при *НОД*:

$$НОК(a_1, a_2, ..., a_n) = НОК(НОК(a_1, a_2, ..., a_{n-1}), a_n)$$

Най-малкото общо кратно може да се намери, като се използва съществуваща зависимост между него и *НОД*, а именно:

$$НОК(a, b) = \frac{a.b}{НОД(a, b)}$$

На базата на приведените по-горе свойства ще реализираме *НОК* на n числа. Числата са зададени в масив A[] и се подават, заедно с техния брой, като параметър на рекурсивната функция lcm():

```
#include <stdio.h>
const unsigned n = 4;
const unsigned A[] = { 10, 8, 5, 9 };
unsigned gcd(unsigned a, unsigned b)
{ return (0 == b) ? a : gcd(b, a % b);
}

unsigned lcm(unsigned a[], unsigned n)
{ unsigned b;
  if (2 == n)
    return(a[0] * a[1]) / (gcd(a[0], a[1]));
  else {
    b = lcm(a, n - 1);
```

```
    return(a[n - 1] * b) / (gcd(a[n - 1], b));
  }
}
```

Алчни алгоритми

В голяма част от задачите, които разгледахме в предходните глави, се търсеше оптималното измежду възможните решения. Нека припомним: при метода *търсене с връщане*, за да се на-мери оптималното решение, е необходимо да се намерят решенията на всички подслучаи на задачата (не непременно оптимални). Основен недостатък е, че някои от тях евентуално биват пресмятани многократно. Повторното пресмятане на едни и същи подслучаи може да се избегне, като се приложи динамично оптимиране, но за нещастие последното е свързано с необходимост от достатъчно памет за запазване на резултатите (и с необходимост от оптимална подструктура на решението, *виж 8.1.*). Идеята, която стои зад *евристичните алгоритми*, е следната: евристичният алгоритъм се насочва към *един* от всичките подслучаи на задачата и решава единствено него, с "надеждата", че той ще се окаже правилният. Изборът на този подслучай се извършва въз основата на локален критерий за оптималност. Така например, *алчните алгоритми*, както подсказва и самото име, се насочват винаги към най-добрия за момента избор, погледнато *локално*, като съвсем естествено, на по-късен етап може да се окаже, че този избор не е бил най-подходящият, погледнато *глобално*.

Алчните алгоритми се съставят лесно, съответната реализация на алгоритъма не е сложна и единственият недостатък е, че понякога те не гарантират правилното решение на задачата. Последното обаче не намалява ползата от тях — за евристичните алгоритми (и в частност алчните) е характерно, че бързо успяват да намерят решение, *близко до оптималното*. В много практически задачи е невъзможно да се изследват всички случаи и често съставянето на алгоритъм, който намира с 5% по-лошо решение от оптималното, се счита за успех, сравнено с алтернативата за едно почти безкрайно и безперспективно претърсване за “истинско” оптимално решение.

Ще се върнем на алчните алгоритми и ще илюстрираме как те се прилагат върху някои конкретни примери. Първата задача, която ще разгледаме, е следната: Да се намери начин за получаване на дадена сума m (m е естествено число), като се използват минимален брой банкноти, с номинали от множеството $C = \{a_1, a_2, ..., a_n\}$. Например, за българската национална валута стойностите на банкнотите са 1, 2, 5, 10, 20, 50 лева. Да разгледаме следния алгоритъм:

1) Инициализираме $s = 0$
2) Намираме банкнотата i с максимална стойност a_i ($a_i \in C$), такава че $s + a_i \leq m$.
   2.1) Ако няма банкнота, за която $s + ai \leq m$, следва, че задачата *няма решение*. Край.
   2.2) Иначе, вземаме банкнотата i и увеличаваме s с ai.
      2.2.1) Ако $s = n$ следва, че *задачата е решена*. Край.
      2.2.2) Ако $s < n$, то още не сме получили цялата сума и повтаряме стъпка 2).

Така например, за сумата 298 лева ще бъдат избрани последователно пет банкноти от

по 50 лева, две от 20, една от 5 и по една банкнота от два и един лева (общо 250 + 40 + 5 + 2 + 1 = 298).

Очевидно, описаният алгоритъм отговаря на критериите за алчен алгоритъм: на всяка стъпка той избира максималната по-стойност банкнота, като по този начин се стреми да постигне възможно най-бързо търсената сума. В конкретния пример той води до ефективно решение на за-дачата.

За съжаление не съществуват достатъчно общи схеми за това какви да бъдат номиналите на банкнотите, за които да може да се твърди, че алчният алгоритъм ще работи правилно за всяка сума. Така например, да разгледаме случая, при който възможните стойности за банкноти са 2, 5, 20 и 30 и искаме да получим сума 40. Алчният алгоритъм първо ще избере банкнота от 30 (максималното възможно), след което ще избере две банкноти по 5, т. е. общо 3 банкноти. Очевидно обаче, съществува и по-добро решение: две банкноти по 20. Освен че може да не намери оптималното решение, възможно е алчният алгоритъм изобщо да не намери решение. Така например, ако искаме да получим сума 6: след банкнотата със стойност 5 пред алгоритъма не остава никаква възможност за следващ избор (а сумата все пак може да бъде получена от 3 банкноти със стойност 2).

Все пак нещата не винаги са толкова лоши и съществуват редица задачи, при които може да се покаже, че алчният алгоритъм *винаги* намира решение. Материалът в този параграф обхваща най-известните задачи, които *могат* да се решат лесно и ефективно с помощта на алчни алгоритми.

Египетски дроби

Ще разгледаме една проста задача, подобна на тази за получаване на парична сума с минимален брой банкноти. Древните египтяни са използвали означение само за дробите с числител единица. Всяка друга дроб p/q представяли и записвали като сума от такива дроби (с числител единица). Например, 7/9 може да се представи като сума по някой от следните начини:

$$7/9 = 1/3 + 1/3 + 1/9$$
$$7/9 = 1/2 + 1/4 + 1/36$$
$$7/9 = 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9$$

***Задача*:** Дадени са две естествени числа p и q ($q \neq 0$, $p < q$; $p,q \in N$). Да се намери представяне на дробта p/q във вид на сума:

$$p/q = 1/a_1 + 1/a_2 + ... + 1/a_n,$$

при което знаменателите да бъдат различни ($ai \neq aj$, $1 \leq i, j \leq n$, $i \neq j$, $ai \geq 2$, $aj \geq 2$, $ai, aj \in N$).

Забележка: Възможно е задачата да има повече от едно решение. Например, за дробта 3/7 две възможни решения са:

$$3/7 = 1/3 + 1/11 + 1/231$$

$$3/7 = 1/4 + 1/8 + 1/19 + 1/1064$$

В конкретната задача търсим *произволно* решение, отговарящо на условието знаменателите на намерените дроби да бъдат различни.

Съществува прост алчен алгоритъм за решаване на задачата: На всяка стъпка поредният член в сумата да бъде максималната дроб, която може да се добави към текущата сума така, че резултатът да не надвишава p/q (тъй като числителят е винаги 1, това означава дробта с най-малък знаменател). Например, за $p/q = 7/9$ най-голямата възможна дроб е 1/2. По-нататък трябва да изберем нова дроб $1/a_2$, така че

$$1/2 + 1/a_2 \leq 7/9, \text{ т. е. } 1/a_2 \leq 7/9 - 1/2, \text{ или } 1/a_2 \leq 5/18.$$

Най-голяма дроб, отговаряща на това условие, е 1/4, при което получаваме:

$$1/a_3 \leq 7/9 - 1/2 - 1/4 = 5/18 - 1/4 = 2/72,$$

т. е. максималното a_3 е 1/36, с което сумирането приключва, тъй като 1/2 + 1/4 + 1/36 = 7/9. Изчисленията в последния пример подсказват каква ще бъде схемата за реализация на алгоритъма:

```
while (p > 1) {
  Намира_се_максималната_дроб_1/r_ненадвишаваща_p/q;
  Отпечатва_се_дробта_1/r;
  p/q = p/q - 1/r;
}
```

В горната схема трябва да уточним две неща. Първото е, как да търсим максималната дроб $1/r$, ненадвишаваща p/q ($q \neq 0$), т. е. минималното r, за което е изпълнено $1/r \leq p/q$, $r \geq 2$, $q \geq 2$. Последното неравенство е еквивалентно на $r \geq q/p$. За да намерим r, можем да извършим делението q/p и да вземем най-малкото цяло число, не по-малко от q/p (в езика Си може да се използва функцията `ceil(q/p)`). За да избегнем използването на реални числа (за по-голямо бързодействие, както и за да си спестим грешки при закръгляне), ще намерим r, като използваме само целочислено деление: r = (q+p)/p;.

Разликата $p/q - 1/r$ се пресмята чрез привеждане под общ знаменател. Така, новите стойности за p и q ще бъдат:

```
p = p*r - q;
q = q*r;
```

Възможно е, след пресмятане на разликата да се получи съкратима дроб. Това ще попречи на правилната работа на алгоритъма единствено в случая, когато се получи дроб p/q, която може да се съкрати до $1/x$, т. е. q % p == 0. В този случай, ако не се извърши съкращението, условието $p > 1$ ще продължава да бъде изпълнено и търсенето на дроби ще продължи до безкрайност. (*Защо?*) Следва реализация на алгоритъма, където сме се погрижили за този случай.

```
/* ако q е кратно на p, се извършва съответно съкращение */
```

```
void cancel(unsigned long *p, unsigned long *q) {
     if (0 == *q % *p) {
          *q /= *p;
          *p = 1;
     }
}

void solve(unsigned long p, unsigned long q) {
     printf("%lu/%lu = ", p, q);
     cancel(&p, &q);
     while (p > 1) {
          /* намира максималната дроб 1/r, 1/r<=p/q */
          unsigned long r;
          r = (q + p) / p;
          printf("1/%lu + ", r);
          /* изчислява p/q - 1/r */
          p = p * r - q;
          q = q * r;
          cancel(&p, &q);
     }
     if (p > 0)
          printf("%lu/%lu ", p, q);
     printf("\n");
}

int main(void) {
     solve(3, 7);
     return 0;
}
```