Limbajul Algoritmic: Exerciţii

Ştefan Ciobâcă, Dorel Lucanu
Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania

PA 2015/2016

1 Introducere

Scopul seminarului este de familiarizare cu limbajul Alk, scrierea de algorimi simli în Alk şi de testare a acestora cu `K Tool`, exersarea calculului pentru funcţia timp.

2 Exerciţii rezolvate

Exerciţiul 1. Să se proiecteze un algoritm care determină primele n numere prime pentru un n dat. Ce se poate spune despre timpii de execuţie uniform şi logaritmic?

Soluţie. Să reamintim mai întâi definiţia pentru numere prime[1]:

> A *prime number* (or prime integer, often simply called a "prime" for short) is a positive integer $p > 1$ that has no positive integer divisors other than 1 and p itself. More concisely, a prime number p is a positive integer having exactly one positive divisor other than 1, meaning it is a number that cannot be factored. For example, the only divisors of 13 are 1 and 13, making 13 a prime number, while the number 24 has divisors $1, 2, 3, 4, 6, 8, 12$, and 24 (corresponding to the factorization $24 = 2^3 \cdot 3$), making 24 not a prime number. Positive integers other than 1 which are not prime are called *composite numbers*.

Pe baza acestei definiţii ne putem construi şi primele teste:

[1] Sursa: `http://mathworld.wolfram.com/PrimeNumber.html`

n	primele n numere prime
0	
1	2
2	2, 3
3	2, 3, 5
4	2, 3, 5, 7

Structurile de date utilizate: cele n numere prime vor fi memorate într-o listă. Vom scrie o funcţie după următorul şablon (schemă):

```
firstNPrimes(n) {
  calculează primele $n$ numere prime şi le memorează în lista  l
  return l;
}
```

Şablonul de mai sus poate fi rafinat prin schema de construcţie a unei liste element cu element:

```
firstNPrimes(n) {
  l = emptyList;
  while ( l.size() < n) {
    calculează următorul număr prim  x
    l.pushBack(x);
  }
  return l;
}
```

Calcularea următorului număr prim poate fi făcută prin parcurgerea numerelor naturale > 1 şi testarea lor dacă sunt prime:

```
firstNPrimes(n) {
  x = 2;
  l = emptyList;
  while ( l.size() < n) {
    if (isPrime(x)) {
      l.pushBack(x);
    }
    ++ x;
  }
  return l;
}
```

Algoritmul `isPrime(x)` poate fi scris printr-o schemă iterativă care testează dacă există un număr întreg pozitiv $n \neq 1, n$ care divide pe x; dacă da, atunci algoritmul întoarce *false*, altfel întoarce *true*. Numerele întregi pozitive $n \neq 1, n$ care ar putea divide pe x sunt $2, 3, \ldots, x/2$. Rezultă că algoritmul `isPrime(x)` poate fi scris simplu după cum urmează:

```
isPrime(x) {
  if (x < 2) return false;
  for (i= 2; i <= x / 2; ++i)
    if (x % i == 0) return false;
  return true;
}
```

Acum scriem cei doi algorimti, împreună cu instrcţiunea de testare, într-un fişier `prime.alk`:

```
isPrime(x) {
  if (x < 2) return false;
  for (i= 2; i <= x / 2; ++i)
    if (x % i == 0) return false;
  return true;
}

firstNPrimes(n) {
  x = 2;
  l = emptyList;
  while ( l.size() < n) {
    if (isPrime(x)) {
      l.pushBack(x);
    }
    ++ x;
  }
  return l;
}

a = firstNPrimes(3);
```

Algoritmul poate fi testat prin următoarea linie de comandă (se presupune că limbajul Alk a fost deja compilat):

```
$ ~/k-3.6/bin/krun tests/prime.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    a |-> < 2, 3, 5 >
    firstNPrimes |-> lambda ( n ) . ({ ((x = 2) ;) ((l = emptyList) ;) (
      while ( l . size ( ) < n ) { (if ( isPrime ( x ) ) { l . pushBack ( x
      ) ; }) ((++ x) ;) }) (return l ;) })
    isPrime |-> lambda ( x ) . ({ (if ( x < 2 ) return false ;) (for ( i = 2
       ; i <= x / 2 ; ++ i ) if ( x % i == 0 ) return false ;) (return true
      ;) })
</state>
```

Opţiunea `-cINIT=".Map"` precizează starea iniţială. Dacă instrucţiunea de test se schimbă cu `a = firstNPrimes(n);`, atunci starea iniţială trebuie să includă valoarea lui n:

```
$ ~/k-3.6/bin/krun tests/prime.alk -cINIT="n |-> 3"
<k>
    .K
</k>
<state>
    a |-> < 2, 3, 5 >
    firstNPrimes |-> lambda ( n ) . ({ ((x = 2) ;) ((l = emptyList) ;) (
      while ( l . size ( ) < n ) { (if ( isPrime ( x ) ) { l . pushBack ( x
      ) ; }) ((++ x) ;) }) (return l ;) })
    isPrime |-> lambda ( x ) . ({ (if ( x < 2 ) return false ;) (for ( i = 2
       ; i <= x / 2 ; ++ i ) if ( x % i == 0 ) return false ;) (return true
      ;) })
    n |-> 3
</state>
```

sfsol

3 Exerciţii propuse

Exerciţiul 2. Să se acrie un algoritm care generează prime cu ciurul lui Eratosthenes (`http://mathworld.wolfram.com/SieveofEratosthenes.html`).

Exerciţiul 3. Limbajul Alk include şi operaţii peste mulţimi (`tests/sets.alk`):

```
s1 = { 1 .. 5 };
s2 = { 2, 4, 6, 7 };
a = s1 U s2 ;
b = s1 ^ s2;
c = s1 \ s2;
x = 0;
forall y in s2 x = x + y;
d = emptySet;
forall y in { 1 .. 6 }
  if (y belongsTo s2) d = d U { y };
```

Ce se poate spune despre timpii de execuţie uniform şi logaritmic ai fiecării operaţii?