Algorithmic Recipes in Haskell

Chao Xu

Last update: December 29, 2011

Contents

Chapter 1

Exact Numerical Algorithms

1.1 Continued Fraction Representation

Continued fractions can be used to represent real numbers in a much more natrual way than decimal notations. In fact, the arithmetic operations are not difficult for continued fractions.

A continued fraction can be implemented as a list of integers. There is one operation that works for all arithmetics of continued fraction. All other ones can be derived.

1.1.1 The main arithmetic operation

The algorithm is derived by Bill Gosper. The algorithm here is directly modeled after is Mark Jason Dominus's talk.

Link of current implementation.

Chapter 2

Algebraic Algorithms

2.1 Exponentiation by squaring

Problem 2.1.1.
Input
Given the operator $\cdot$, element a and positive integer n. Where a is an element of a semigroup under $\cdot$.
Output
Find a^n, where $a^n = a \cdot a^{n-1}$.

The general method to solve the problem is exponentiation by squaring. It is originally used for integer exponentiation, but any associate operator can be used in it's place. Here is a theorem stated in algebraic flavor.

Theorem 2.1.1. *For any semigroup $(S, \cdot)$, $x \in S$ and $n \in \mathbb{N}$, x^n can be computed with $O(\log n)$ applications of $\cdot$.*

Proof. Express n as binary $c_k c_{k-1} \ldots c_0$, where $c_i \in \{0, 1\}$. We make sure $0 \cdot a$ is the treated as the identity, and $1 \cdot a = a$ for all a. The following observations are crucial.

$$a^n = c_0 a^{2^0} \cdot c_1 a^{2^1} \cdot \ldots \cdot c_k a^{2^k}$$
$$a^{2^{i+1}} = a^{2^i} \cdot a^{2^i}$$

The code that compute a^n from the above two equalities.

```
import Data.Digits
exponentiationBySquaring :: Integral a ⇒ (b → b → b) → b → a → b
exponentiationBySquaring op a n = foldr1 op $ [y | (x, y) ← (zip binary twoPow), x ≢ 0]
  where twoPow = a : zipWith op twoPow twoPow
    binary = digitsRev 2 n
```

One can analyze the number of times the operator is used. The *twoPow* is the infinite list $[a, a^2, \ldots, a^{2^i}, \ldots$. It takes k operations to generate the first $k+1$ elements. At most k additional operations are required to combine the result with the operator. Therefore the operator is used $O(\log n)$ times. □

This result can of course be extended to monoid and groups, so it work for all non-negative and integer exponents, respectively.

2.2 Linear homogeneous recurrence relations with constant coefficients

Definition 2.2.1 (Linear homogeneous recurrence relations with constant coefficients)**.** A linear homogeneous recurrence relations in ring R with constant coefficients of order k is a sequence with the following recursive relation

$$a_n = \sum_{i=1}^{k} c_i a_{n-i}$$

, where c_i are constants.

We use linear recurrence relation to abbreviate.

The most common example is the Fibonacci sequence. $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ in the ring $\mathbb{Z}$. The Fibonacci sequence have a simple implementation. $\mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail\ fibs})$. We want to generalize it.

2.2.1 Lazy sequence

Problem 2.2.1.
Input

1. A list of coefficients $[c_1, c_2, \ldots, c_n]$ of a linear recurrence relation.
2. A list of base cases $[a_0, a_1, \ldots, a_{n-1}]$ of a linear recurrence relation.

Output
The sequence of values of the linear recurrence relation as a infinite list $[a_0, a_1, \ldots$.

Here is a specific implementation where we are working in the ring $\mathbb{Z}$.

```
import Data.List
linearRecurrence :: Integral a => [a] -> [a] -> [a]
linearRecurrence coef base = a
  where a = base ++ map (sum . (zipWith (*) coef)) (map (take n) (tails a))
        n = (length coef)
```

One can generalize it easily to any ring.

Having a infinite list allows simple manipulations. However, finding the nth element in the sequence cost $O(nk)$ time. It becomes unreasonable if a person only need to know the nth element.

2.2.2 Determine nth element in the index

If n is very large, a more common technique would be solve for a_n using matrix multiplication.

2.2.3 Linear Recurrence in Finite Ring

Linear recurrence is perodic in finite rings. Therefore one might want to produce only the periodic part of the ring. [INSERT MORE ON THIS SUBJECT]

2.3 A particular kind of recurrence

A common recurrence has the form

$$a_n = \sum_{i=0}^{\infty} b_i a_{n-m_i}$$

, where m_i and b_i are both *infinite* sequences. $m_i \in \mathbb{N}$. $a_{-i} = 0$ for all positive i. This is well defined as long as b_i, a_i are in the same ring.

Problem 2.3.1.
Input
Infinite sequence b_i and m_i, $m_i \in \mathbb{N}$. Finite sequence $c_0, \ldots, c_k$
Output
The infinite sequence defined as

$$a_n = \begin{cases} \sum_{i=0}^{\infty} b_i a_{n-m_i} & \text{if } n > k \\ c_n & \text{if } n \leqslant k \end{cases}$$

One can use a balanced binary tree to store the entire infinite list, and the time to generate the nth element is $O(d(n)\log n)$, where d is the density function of $\{m_i\}$.

Using an array would make it $O(d(n))$, but it is too imperative for our taste, how about we only use list and achieve $O(d(n))$ time, elegantly?

The idea is that we are summing the first item of infinite many stacks. However we don't have to really sum the infinite stacks, we only sum the stack we require.

```
import Data.List
rec :: Num a => [a] -> [a] -> [Int] -> [a]
rec c b m = a
  where a = c ++ rest
    rest = next [] 0 m
    next xs k (m : ms)
      | k == m = next (a : xs) k ms
      | otherwise = val ++ next (map tail xs) (k + 1) (m : ms)
      where val = if (k < length c) then [] else [sum $ zipWith (*) (reverse (map head xs)) b]
```

This kind of problem can be thought of moving the pointer of to a_n from a_{n-1}, how does the pointers that originally pointing to all the elements a_{n-1} requires to sum should move to now? This is a extra complication required when an array is not accessable.

2.4 Canonical forms of a boolean function

One can always describe a boolean function f over n variables to a list of 2^n boolean values, by mapping it into a function g.

$$f(x_n, \ldots, x_1) = g(\sum_{i=1}^{n} 2^{i-1} x_i)$$

2.4.1 Sum of minterms

Problem 2.4.1.
Input
A list of values of $\{0, 1\}$, of length 2^n.
Output
Find a function of the following form that also generate the same list.

$$\bigvee_{y \in Y} (\bigwedge_{x \in y} x)$$

Where each x is either x_i or $\neg x_i$ for some i. Y can be described by a list of lists. Use i to denote x_i and $-i$ to denote x_i.

2.4.2 Product of maxterms

Problem 2.4.2.
Input
Input the same as the sum of minterms.
Output
Find a function of the following form that also generate the same list.

$$\bigwedge_{y \in Y} (\bigvee_{x \in y} x)$$

Output Y.

2.4.3 Implementation

```
import Data.Digits
import Data.List
import Data.List.Utils

sumOfMinterms = snd ∘ booleanCanonicalForm
productOfMaxterms = fst ∘ booleanCanonicalForm

booleanCanonicalForm :: Integral a ⇒ [a] → ([[a]], [[a]])
booleanCanonicalForm values = (snd $ unzip pos, snd $ unzip sop)
  where (pos, sop) = partition (λ(x, y). x ≡ 0) (zip values power)
    power   = map (terms ∘ (replace [0] [−1]) ∘ pad ∘ digitsRev 2) [0 ..]
    terms d = zipWith (∗) d [1 ..]
    pad a   = a ++ replicate (n − (length a)) 0
    n       = floor $ logBase 2 (fromIntegral (length values))
```

Chapter 3

Combinatorial Algorithms

3.1 List of Lattice Points

Problem 3.1.1.
Input
Positive integer k.
Output
A infinite list that contain all nonnegative lattice points in k-th dimension.

$$
\begin{aligned}
&\mathit{nonNegativeLatticePoints}\ k = \mathit{concat}\ \$\ \mathit{map}\ (\mathit{sumToN}\ k)\ [0\mathinner{..}]\\
&\quad \textbf{where}\ \mathit{sumToN}\ k\ n\\
&\qquad \mid k \equiv 1 = [[n]]\\
&\qquad \mid \mathit{otherwise} = \mathit{concat}\ [(\mathit{map}\ (i{:})\ (\mathit{sumToN}\ (k-1)\ (n-i))) \mid i \leftarrow [0\mathinner{..}n]]
\end{aligned}
$$

Problem 3.1.2.
Input
Positive integer k.
Output
A infinite list that contain all lattice points in k-th dimension.

To show an example, here is list of integers.

$$
\begin{aligned}
&\mathit{integers} :: [\mathit{Integer}]\\
&\mathit{integers} = (0{:})\ \$\ \mathit{concat}\ \$\ \mathit{zipWith}\ (\lambda x\ y.\,[x,y])\ [1\mathinner{..}]\ (\mathit{map}\ \mathit{negate}\ [1\mathinner{..}])
\end{aligned}
$$

One want a way to be able to list all elements in the k-th dimension.

3.2 Integer Partitions

Definition 3.2.1 (Integer Partition)**.** A integer partition of n is a multiset $\{a_1, \ldots, a_k\}$, such that $\sum_{i=1}^{k} a_i = n$.

Definition 3.2.2 (Partition Numbers)**.** The sequence of partition numbers $\{p(n)\}$ is the number of integer partitions for n.

Problem 3.2.1.
Input
Integer n.
Output
List of partitions of n.

To find all possible partition of a integer, we proceed with a simple recursive formula.

Let $p(n,k)$ be the list of ways to partition integer n using integers less or equal to k. $p(n,n)$ is the solution to our problem. It is implemented as *part* in the code.

$$
\begin{aligned}
&\mathit{integerPartitions} :: \mathit{Integral}\ a \Rightarrow a \rightarrow [\,a\,] \\
&\mathit{integerPartitions}\ n = \mathit{part}\ n\ n \\
&\quad \textbf{where}\ \mathit{part}\ 0\ \_ = [[\,]] \\
&\qquad \mathit{part}\ n\ k = [(i : \mathit{is}) \mid i \leftarrow [1\mathinner{..}\mathit{min}\ k\ n], \mathit{is} \leftarrow \mathit{part}\ (n-i)\ i]
\end{aligned}
$$

Problem 3.2.2.
Input
None
Output
The infinite list of partition numbers.

Naively, $0 : \mathit{map}\ (\mathit{length} \circ \mathit{integerPartitions})\ [1\mathinner{..}]$ works well, except the time complexity is $O(np(n))$, and $p(n)$ is exponential. A more well known approach, that only cost $O(\sqrt{n})$ additional operations to generate the nth number, will be given instead.

Extend the definition of the partition number, such that $p(0) = 1$ and $p(-n) = 0$ for all positive integer n. The partition number $p(n)$ has the relation

$$p(n) = \sum_{k=0}^{\infty} (-1)^k (p(n - p_{2k+1}) + p(n - p_{2k+2}))$$

where p_n is the sequence of generalized pentagonal number.

We have already developed the tools to work with this kind of recurrence in section 2.3.

$$
\begin{aligned}
&\mathit{generalizedPentagonalNumbers} :: [\,\mathit{Integer}\,] \\
&\mathit{generalizedPentagonalNumbers} = [(3 * n \uparrow 2 - n)\ \text{`}\mathit{div}\text{`}\ 2 \mid n \leftarrow \mathit{integers}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{partitionNumbers} :: [\,\mathit{Integer}\,] \\
&\mathit{partitionNumbers} = \mathit{rec}\ [1]\ (\mathit{cycle}\ [1, 1, -1, -1])\ (\mathit{tail}\ \mathit{generalizedPentagonalNumbers})
\end{aligned}
$$

3.3 Find the primitive word in a free monoid

Problem 3.3.1.

Input
A word w in a free monid.
Output
A primitive word p, such that $p^n = w$ for some integer n.

A word p is primitive if $p = w^k$ implies $k = 1$. This will use the algorithm in [1]. [Nah, just KMP...]

3.4 Period of a eventually periodic sequence

A sequence is eventually periodic if it is a concatination of a finite sequence and a periodic sequence.

3.4.1 Knows the upper bound of the period and a certain condition

Problem 3.4.1.
Input

1. A integer of the upper bound u of the period.
2. A infinite list that represent a eventually periodic sequence, such that if two finite sequence of length u are equal and the starting index is less than u apart, then they must be inside the periodic part of the sequence.

Output
A pair of the initial sequence and the periodic part.

The naive algorithm, for each finite sequence of length u, see if the second condition in the input holds, does pretty well if u is small. In fact $O((n+u)u^2)$ where n is the length of the aperiodic part.

```
import Data.List
import Data.Maybe
eventuallyPeriodic :: Eq a => [a] -> Int -> ([a], [a])
eventuallyPeriodic sequence bound = (ini, take period rep)
  where table = map (take (bound + 1)) (tails (map (take bound) (tails sequence)))
        exist      = map (\x. elemIndex (head x) (tail x)) table
        period     = 1 + (fromJust $ head just)
        (no, just) = span isNothing exist
        (ini, rep) = splitAt (length no) sequence
```

Of course it can be improved to $O((n+u)u)$ easily by using a smarter string search algorithm like KMP. Under a simple observation $O(n+u)$ is the possible.

The algorithm can be abstracted as another sequence. Given a sequence a in the problem, and a u, we can define another sequence b, such that b_i is true if and only if a_i meets condition two. $b_i = Fasle, \ldots, False, True, \ldots$, and the $False$ correspond to the finite part. In this sense, it become obvious a binary search would suffice, and one can construct a solution in $O(n+u+u\log(n+u))$ time.

To make it truly $O(n+u)$, we need to get $u\log(n+u) = O(n+u)$. How so, when we don't even know what n is? Only when n is very small would $u\log(n+u) > n$. Consider the following hackish algorithm:

$O(n+u)$ is clearly the lowerbound, one must read to the $n+u$th position in the sequence to be able to decide the periodic part.

Check if the condition is true for b_{ku}, where k is a integer. After $(n+u)/u+1$ tests are required figure out which u positions can be the start of the periodic sequence. We know this can be done in $O((n+u)/u \times u) = O(n+u)$ time.

This the problem really reduce to can we find a the first substring of length u that appears twice in a string of length $2u$ in $O(u)$ time.

A variation of the problem could be the upper bound for length of the non-periodic part of the sequence is known.

3.4.2 Upper bound of the length of the non-periodic part and upper bound of the period are known

Problem 3.4.2.
Input

1. A infinite list that represent a eventually periodic sequence.
2. A integer of the upper bound u of the period.
3. A integer n represent the upper bound on the length of the non-periodic part of the sequence.

Output
A pair of the initial sequence and the periodic part.

Bibliography

[1] Artur Czumaj and Leszek Gasieniec. On the complexity of determining the period of a string, 2000.