Cookbook

Last update: December 1, 2011

Contents

1 Algebraic Algorithms **2**

1.1 Exponentiation by squaring 2

1.2 Linear homogeneous recurrence relations with constant coefficients 2

2 Combinatorial Algorithms **4**

2.1 Integer Partitions 4

Chapter 1

Algebraic Algorithms

1.1 Exponentiation by squaring

Problem 1.1.1.
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 1.1.1. *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 $c_k c_{k-1} \dots c_0$, 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 $[a, a^2, \dots, a^{2^i}, \dots$. 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. $\square$

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

1.2 Linear homogeneous recurrence relations with constant coefficients

Definition 1.2.1 (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. $fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$. We want to generalize it.

1.2.1 Lazy sequence

Problem 1.2.1. *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.

1.2.2 Determine nth element in the index

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

1.2.3 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]

Chapter 2

Combinatorial Algorithms

2.1 Integer Partitions

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.

```
integerPartitions :: Integral a => a -> [a]
integerPartitions n = part n n
  where part 0 _ = [[]]
        part n k = [(i : is) | i <- [1 .. min k n], is <- part (n - i) i]
```