Article can be accessed online at http: //www.publishingindia.com

A Method for Derivating an Algorithm of Power Computation of Cyclic Permutation

Bo Yang*

Abstract

Permutation algorithm is widely used in numerous information technology domains such as cryptology. Any permutation can be viewed as a combination of several cyclic permutations which have no shared element. This article continues the work discussed in existing documents, reforms its part of formal derivation using dynamic programming in order to reveal essence of this algorithm. On the basisof ring representation for cyclic permutation, this article uses dynamic programming to get a bottom-up recurrence relation through a series of formal derivations. And then gets a concise middle algorithm. After that, through correlation between ring and abstract array containing a cyclic permutation, it uses an equally simple coordinate transformation to yield the final algorithm for computing positive power of cyclic permutation stored as a function in an abstract array.

Keyword : Cyclic Permutation, Dynamic Programming, Recurrence Relation, Abstract Array

Introduction

After many years of deep research on essence laws of algorithm and program design, our research group puts forward a series of problems and solves them, and forms Dynamic programming and its support platforms. Dynamic programming (http://docs.oasis-open.org) supports software development formalization (Baldy *et al.*, 1992; Ostroff and Wonham, 1985) and automation (Gardner, 2009; Fidge, 1989), manages to convert creative labours into uncreative labours as much as possible in software development and algorithm design (Netzer and Miller, 1992). Dynamic programming is a simple and practical formal method, which can be used in algorithm design and correctness proof (Christiaens, 2001). Through function specification transformation, it looks for recurrence relations on the base of inductive hypothesis (Gardner, 2009), and can reveal the intrinsic characteristics of many creative algorithms (Baldy *et al.*, 1992). Dynamic programming mainly uses common algorithm design technologies such as partition, recurrence, quantifier transformation and so on (Christiaens, 2001; Lamport, 1978). These are all common technologies used in some typical problems. Dynamic programming integrates them, then expands and reforms them (Netzer and Miller, 1992). Dynamic programming can ensure the correctness of algorithms, can help designer to understand the formation processes of algorithms, and can heighten the designer's abilities of algorithm design. This article continues work discussed in research by Gardner (2009), reforms its part of formal derivation using Dynamic programming in order to reveal essence of this algorithm.

Permutation algorithm is widely used in numerous information technology domains such as cryptology (Shang *et al.*, 2010). Any permutation can be viewed as a combination of several cyclic permutations which have no shared element (Lamport, 1978). Power of permutation is equal to combination of powers of these cyclic permutations. There are many factors which influence the reliability of software related with permutation, and obviously, a very important factor is the reliability of algorithm computing power of cyclic permutation (Cui, 2009). Formal method is the important path to heighten

* School of Information Technology, Jiangxi University of Finance and Economics, Nanchang, China.
E-mail : jxncyangbo2002@163.com

the correctness and reliability of algorithms (Fidge, 1989). On the basisof ring representation for cyclic permutation, this article uses Dynamic programming to transform function specification, gets a bottom-up recurrence relation through a series of formal derivations (Sheng, Iiang and Ping, 2008; Shaodong and Lei, 2010). And then gets a concise algorithm for computing positive power of cyclic permutation stored as a ring. After that, through correlation between ring and abstract array containing a cyclic permutation, uses an equally simple coordinate transformation to yield the final algorithm for computing positive power of cyclic permutation stored as a function in an abstract array.

Permutation and Cyclic Permutation

Considering p to be a permutation of a finite set $s = \{a_1, a_2, \ldots, a_n\}$, p can be viewed as a one-to-one function on the set s."$\cdot$" is used to represent the function application. The following is a permutation p on the set {a, b, c, d, e} :

$p = \begin{pmatrix} a & b & c & d & e \\ c & a & b & e & d \end{pmatrix}$. the np.a = c, p.b = a, p.c = b, p.d = e, p.e = d

if p and q are permutations on the same set, p*q can be defined as another permutation on the set : (p*q). x = p.(q.x). Further p^0 is the identity permutation, p^r is the r power of permutation p, that is, for $r \geq 0$, $p^{r+1} = p*p^r = p^r*p$.

Given a permutation p on set s, if x, y$\in$ s(x, y are any two elements of the set), equations $y = p^i.x$ and $x = p^j.y(i, j \geq 0)$ hold, then the p is cyclic permutation (also called rotation). Any permutation can be viewed as a combination of its cyclic components which have no shared element. For example, $p = \begin{pmatrix} a & b & c & d & e \\ c & a & b & e & d \end{pmatrix}$ can be viewed as the combination of $p_1 = \begin{pmatrix} a & b & c \\ c & a & b \end{pmatrix}$ and $p_2 = \begin{pmatrix} d & e \\ e & d \end{pmatrix}$

Thus, we will concentrate on the cyclic permutation, and consider that p is the cyclic permutation on set s.

Given an abstract array A containing cyclic permutation p, that is, A.x = p.x for all x in domain of p. Array A can also be viewed as a function on set s. Our problem is that how to get the algorithm for computing the positive power of cyclic permutation stored as a function in abstract array A. Thus, the specification is $\{A = p \wedge r > 0\}S\{A = p^r \wedge r > 0\}$.

It is difficult to compute p^ron the abstract array A directly. We will look for another representation of the p in A. Based on it, we derive a middle algorithm in order to reveal the law of computing p^r, after that, an equally simple coordinate transformation is used to yield the final algorithm.

Cyclic permutation p can be represented by a sequence. This sequence consists of elements of the domain of p. In this sequence, any element's follower is p.x(the follower of the last element is the first element). This is the ring representation of cyclic permutation. p^r can be gotten by rotating every element r position to the left in the ring (Gardner, 2009). This is an important property of ring representation of cyclic permutation.Given two rings which are correspondent up and down, the up ring is the sequence k, which is in the original state, the down ring is the sequence h, which is in the result state, that is, p = h : k.At first, $h = k = p^0$; at the end of algorithm execution, every element in h is the result of every element in k rotating r position to left, that is, $h = p^r \wedge k = p^0$. This is the ring representation of cyclic permutation.

Use Dynamic Programming to Formally Derive p^r in Terms of Ring Representation

We use two arrays to store ring, for convenience.Consider that the elements type is char temporarily. Define k, h : array [0.. n – 1] of char; array k is used to store up ring; h is used to store down ring. Initially, h[0] = k[0], h[1] = k[1], …, h[n – 1] = k[n – 1].

Construct the Formal Specification of Computing p^r

AQ : Given two arrays k = h, and p = h : k is the ring representation of p.

Let pcy(p[0 : n –1], r) denote that every element in h is the result of element in k rotating r position to left, that is pcy(p[0 : n – 1], r) $\equiv \forall i : 0 \leq i < n : h[i] = k[(i + r) \bmod n]$. Thus,

AR : pcy(p[0 : n – 1], r)

Analyse the Problem

Change p into two parts p[0 : m – 1] and p[m : n – 1]. pcy(p[0 : m – 1], r) denotes that h[0..m – 1] is the result that interior elements in k[0..m – 1] rotate r position to left within k[0..m – 1], and h[m..n – 1] remains in the original state, that is ∀j : m ≤ j < n : h[j] = k[j].Thus, we define

pcy(p[0 : m – 1], r) ≡ (∀i : 0 ≤ i < m : h[i] = k [(i + r)mod m])∧(∀j : m ≤ j < n : h[j] = k[j])

In Dynamic programming, the aim of this stage is to partition the problem into a couple of sub-problems, each of which has the same structure with the original problem. Thus, we partition computing pcy(p[0 : m – 1], r) into computing pcy(p[0 : m – 2], r) with p[0 : m – 1], that is to search for recurrence F such that :

pcy(p[0 : m – 1], r) = F(pcy(p[0 : m – 2], p[0 : m – 1])) (m > r)

Derive Relation

The relation is F between pcy(p[0 : m – 1], r) and pcy(p[0 : m – 2]. Now, we begin to derive it.

pcy(p[0 : m – 1], r)

⇔ (∀i : 0 ≤ i < m : h[i] = k[(i + + r)mod m])∧(∀j : m ≤ j < n : h[j] = k[j])

∵ ∀i : 0 ≤ i < m : h[i] = k[(i + r)mod m]

⇔ (∀i : 0 ≤ i < m – r : h[i] = k[i + r])∧(∀j : m – r ≤ j < m : h[j] = k[j + r – m]) [Range Splitting]

⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(∀j : m – r + 1 ≤ j < m : h[j] = k[j + r – m])

∧(h[m – 1 – r] = k[m – 1])∧(h[m – r] = k[0]) [Range Splitting and Singleton Range]

∴ pcy(p[0 : m – 1], r)

⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(h[m – 1 – r] = k[m – 1])∧(h[m – r] = k[0])

∧(∀j : m – r + 1 ≤ j < m : h[j] = k[j + r – m]) ∧(∀j : m ≤ j < n : h[j] = k[j])

⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(∀j : m – r ≤ j < m – 1 : h[j] = k[j + r – m])∧(h[m – 1] = k[r – 1])∧(∀j : m ≤ j < n : h[j] = k[j])

[Range Splitting and Singleton Range]

pcy(p[0 : m – 2], r)

⇔ (∀i : 0 ≤ i < m – 1 : h[i] = k[(i + r)mod (m – 1)])∧(∀j : m – 1 ≤ j < n : h[j] = k[j])

⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(∀j : m – 1 – r ≤ j < m – 1 : h[j] = k[j + r – m + 1])

∧(h[m – 1] = k[m – 1])∧(∀j : m ≤ j < n : h[j] = k[j]) [Range Splitting and Singleton Range]

⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(h[m – 1 – r] = k[0])∧(h[m – r] = k[1])

∧(∀j : m – r + 1 ≤ j < m – 1 : h[j] = k[j + r – m + 1])∧(h[m – 1] = k[m – 1])∧(∀j : m ≤ j < n : h[j] = k[j])

[Range Splitting and Singleton Range]

Compare pcy(p[0 : m – 1], r) with pcy(p[0 : m – 2], r), respectively draw the different parts and same part between them.

the same part : fix ⇔ (∀i : 0 ≤ i < m – 1 – r : h[i] = k[i + r])∧(∀j : m ≤ j < n : h[j] = k[j])

the different parts :

rotor1 ⇔ (h[m – 1 – r] = k[m – 1])∧(h[m – r] = k[0])

∧(∀j : m – r + 1 ≤ j < m – 1 : h[j] = k[j + r – m]) ∧(h[m – 1] = k[r – 1])

rotor2 ⇔ (h[m – 1 – r] = k[0])∧(h[m – r] = k[1])

∧(∀j : m – r + 1 ≤ j < m – 1 : h[j] = k[j + r – m + 1]) ∧(h[m – 1] = k[m – 1])

hencepcy(p[0 : m – 1], r) ⇔ fix∧rotor1

pcy(p[0 : m – 2], r) ⇔ fix∧rotor2

∴ pcy(p[0 : m – 2], r)∧○rotor1∧ rotor2∧(fix→○fix) ⇔

fix∧(fix→○fix)∧rotor2∧○rotor1

⇔ ○fix∧○rotor1 ⇔ ○(fix∧rotor1) ⇔ ○pcy(p[0 : m – 1], r) (1)

["○" is next operator in temporal logic]

Define operator "⊙", ⊙h[i] represents the value of h[i] in next state. h[i] represents the value of h[i] in current state.

hence ○rotor1 ⇔

(⊙h[m – 1 – r] = ⊙k[m – 1])∧(⊙h[m – r] = ⊙k[0])

∧(∀j : m – r + 1 ≤ j < m – 1 : ⊙h[j] = ⊙k[j + r – m]) ∧(⊙h[m – 1] = ⊙k[r – 1])

[according to the definition of rotor1]

rotor2 ⇔

(h[m – 1 – r] = k[0])∧(h[m – r] = k[1])

∧(∀j : m – r + 1 ≤ j < m – 1 : h[j] = k[j + r – m + 1]) ∧(h[m – 1] = k[m – 1])

[according to the definition of rotor2]

⇔ (h[m – 1 – r] = ⊙k[0])∧(h[m – r] = ⊙k[1])

∧(∀j : m – r + 1 ≤ j < m – 1 : h[j] = ⊙k[j + r – m + 1]) ∧(h[m – 1] = ⊙k[m – 1])

[the up ring of ring is unchanged]

⇔ (⊙k[m – 1] = h[m – 1])∧(⊙k[0] = h[m – 1 – r])

∧(∀j : m – r + 1 ≤ j < m – 1 : ⊙k[j + r – m] = h[j – 1]) ∧(⊙k[r – 1] = h[m – 2])

∴ ○rotor1∧rotor2 ⇔

(⊙h[m – 1 – r] = ⊙k[m – 1] = h[m – 1])∧(⊙h[m – r] = ⊙k[0] = h[m – 1 – r])

∧(∀j : m – r + 1 ≤ j < m – 1 : ⊙h[j] = ⊙k[j + r – m] = h[j – 1]) ∧(⊙h[m – 1] = ⊙k[r – 1] = h[m – 2])

⇔ (⊙h[m – 1 – r] = h[m – 1])∧(⊙h[m – r] = h[m – 1 – r])

∧(∀j : m – r + 1 ≤ j < m – 1 : ⊙h[j] = h[j – 1]) ∧(⊙h[m – 1] = h[m – 2])∧rotor2

⇔ (⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1])∧rotor2 (2)

⊙pcy(p[0 : m – 2], r)∧

(fix→○fix)∧(⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1])

⇔ pcy(p[0 : m – 2], r)∧(fix→○fix)

∧(⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1]) ∧rotor2

[according to pcy(p[0 : m – 2], r) ⇔ fix∧rotor2]

⇔ pcy(p[0 : m – 2], r)∧○rotor1∧ rotor2∧(fix→○fix) [according to equation (2)]

⇔ ○pcy(p[0 : m – 1], r) [according to equation (1)]

Therefore, we get relation as following lemma.

Lemma

pcy(p[0 : m – 2], r)∧

(fix→○fix)∧(⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1])

⇔ ○pcy(p[0 : m – 1], r) r < m ≤ n

Clearly, (fix→○fix)∧(⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1]) denotes that internally rotating h[m – 1 – r], …, h[m – 1] these r + 1 elements one position to right, the rotation range is limited within h[m – 1 – r], …, h[m – 1], and other elements in array h are unchanged.

Above recurrence relation is derived from formal specification. The recurrence relation implicates the thoughts of designing algorithm for computing p^r.The thought is that: consider preceding m – 1 elements in ring h is of the preceding m – 1 elements in ring k(that is k[0], …, k[m – 2]) rotating r position to left within these elements, in this case, we should note that the rotation is limited to the range of k[0], …, k[m – 2] regardless of other elements. Based on it, the next iteration of recurrence rotates h[m – 1 – r], …, h[m – 1] these r + 1 elements one position to right, and it brings about that the preceding m elements in ring h complete the r time interior rotations limited within k[0], …, k[m – 1], that is pcy (p[0 : m – 1], r).When m = n, the final result can be derived. Dynamic programming supposes the former sub-problems have been solved. On the basis of this hypothesis, recurrence relations are derived from formal specification.This method can reduce the range of thought, and can help to formally derive correct and reliable algorithm.

Construct Algorithm

Obviously, in Lemma, the expression

(fix→○fix)∧(⊙h[m – 1 – r] = h[m – 1])∧(∀j : m – r ≤ j ≤ m – 1 : ⊙h[j] = h[j – 1])

is correspondent to following assignment statements:

h[m – 1 – r], h[m – r], …, h[m – 1] : = h[m – 1], h[m – 1 – r], …, h[m – 2]

We describe the middle algorithm PCY1 as following:

Prequistite. m = r + 1

ALGORITHM : PCY1

|[h, k : arrar(0 : n – 1, char);m, n, r : integer]|

{AQ∧AR}

A-I : pcy(p[0 : m – 2], r)

BEGIN : m = r + 1 + + 1

END : m = n + 1

RELATIONSHIP: h[m – 1 – r], h[m – r], …, h[m – 1] : = h[m – 1], h[m – 1 – r], …, h[m – 2]

END

According to recurrence relation, the loop invariant is pcy(p[0 : m – 2], r).

Coordinate Transformation to an Algorithm in Terms of A

In above middle algorithm PCY1, ring is stored in arrays, up ring is unchanged, elements in down ring shift by reference to the up ring. At last, the down ring is the result of computing p^r. In this case, the precondition of algorithm is that the ring representation of p should be given in advance, and any element's follower is gotten by increasing the index of array. In order to eliminate this precondition, we need to coordinately transform middle algorithm into an algorithm in terms of abstract array A.

We can view A as a function with two arrays.We can get the follower of arbitrary element in domain through "." which denotes the application of function, instead of through increasing the index of array. For example, if a = k[i], then A.a = k[(i + 1)mod n]. This is the correlation between ring and abstract array.

The initial state:

k ring	k[0]	k[1]	k[2]	k[3]	k[4]
h ring	h[0] = k[0]	h[1] = k[1]	h[2] = k[2]	h[3] = k[3]	h[4] = k[4]

The initial state of abstract array A:

array A	k	k[0]	k[1]	k[2]	k[3]	k[4]
	h	A.k[0]	A.k[1]	A.k[2]	A.k[3]	A.k[4]

Because A.k[i] = k[(i + 1)mod n], thus, initiallyin abstract array A, array h is the one time cyclic permutation of array k. In algorithm PCY1, the elements in ring h are rotated r position to left.Correspondently, in order to transform it into algorithm in terms of A, the elements in array h in A should be rotated r – 1 position to left. The middle algorithm PCY2 is gotten from algorithm 1 by replacing r with r – 1. In this algorithm, every element in h is rotated r – 1 position to left.

Prerequistite. m = r

ALGORITHM : PCY2

|[h, k : arrar(0 : n – 1, char);m, n, r : integer]|

BEGIN : m = r + + 1

END : m = n + 1

RELATIONSHIP : h[m – r], h[m – r + 1], …, h[m – 1] : = h[m – 1], h[m – r], …, h[m – 2]

END

Consider x is any element in permutation p, the follower of x can be gotten through A. x rather than through increasing index of array. Therefore, the algorithm need not rely on the index of array, the act of variable m, n is only to control the time of loop. From this, the above algorithm PCY2 can be transformed into final algorithm in terms of A as following:

Prerequistite. m = r, $x_0 = x$, $x_1 = A.x_0$, …, $x_{r-1} = A.x_{r-2}$

ALGORITHM : PCY

|[x_0, x_1, …, x_r : char;m, n, r : integer]|

BEGIN : m = r + + 1

END : m = n + 1

RELATIONSHIP : $A.x_0$, $A.x_1$, ..., $A.x_{r-1}$: = $A.x_{r-1}$, $A.x_0$, ..., $A.x_{r-2}$

x_0, x_1, …x_{r-2}, x_{r-1} : = x_1, x_2, …x_{r-1}, $A.x_0$

END

In this algorithm, it is unnecessary to give ring representation for cyclic permutation in advance. The algorithm execution time is the linear function of number of elements in the permutation. r + 1 extra storage units are used in this algorithm.

On the basis of this algorithm, we use a function on two arrays to implement the abstract array A, and add input and output statements, and then convert the algorithm into executive program.

Concluding Remarks

This article uses dynamic programming to solve the problem about computing the positive power of cyclic permutation. It is difficult to compute p^ron the abstract array A directly. We give another representation of the p in A. Based on it, we derive a middle algorithm in order to reveal the law of computing p^r. An equally simple coordinate transformation is used to yield the final algorithm.Cyclic permutation can be represented by a sequence. This sequence consists of elements of the domain of p. Any element's follower is p.x. This is the ring representation of cyclic permutation. Anp^r can be gotten by rotating every element r position to the left in the ring. The recurrence relation implicates the thoughts of designing algorithm for computing p^r. We use two arrays to store ring. The main ideas and ingenuity of the algorithms that are represented by lemma are revealed by formula deduction.

These are useful in understanding how these algorithms are designed. After formally deriving recurrence relation with dynamic programming, we can get correct, reliable, efficient algorithmic program. Algorithmic program design is a skillful and creative labour. The designer of algorithm using dynamic programming can partly avoid the difficulty in making choice among various existed design methods. The level of software automation can be heightened.

References

Baldy, P., Dicky, H., Medina, R., Morvan, M. & Vilarem, J. F. (1992). Efficient Reconstruction of the Causal Relationship in Distributed Computations. Technical Report 92-013, France.

Christiaens, M. (2001). *TRaDe, A Topological Approach to On-the-Fly Race Detection in Java Programs*. Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, 1, 1-15.

Cui, J. (2009). *DRD4BPEL: A Tool for Data Race Detection of BPEL Process*. World Congress on Software Engineering, (pp. 200-205).

Fidge, C. J. (1989). *Dynamic Analysis of Event Orderings in Message-Passing Systems* (Doctoral Dissertation) Australian Nat'l University.

Gardner, M. (2009). *Mathematical Circus: More Puzzles, Games, Paradoxes and Other Mathematical Entertainments From Scientific American*. New York: The Mathematical Association of America, (pp. 111-122).

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* , 21(7) 558-565.

Netzer, R. H. & Miller, B. P. (1992). What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), 74-88 .

Ostroff, J. S. & Wonham, W. M. (1985). *A Temporal Logic Approach to Real Time Control*. In 24th Conference Proceedings of Decision Control, 24, 656-657.

Shang, C., Liu, Q. & Yang, Z. (2010). Implementing transaction management in cross engine business process recursive composition by BPEL Extension. *Journal of Computational Information Systems*, 6(8), 2603-2607.

Shaodong, L. & Lei, X. (2010). A static data race detecting method for BPEL based on happens-before and Lockset. *Computer and Didital Engineering*, 250(8), 6-9.

Sheng, C., Iiang, B. & Ping, C. (2008). Study of algorithm on data race and deadlock detection for BPEL process. *Journal of Xidian University*, 35(6), 1056 -1063.

Web Services Business Process Execution Language Version 2.0. Retrieved from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.htm. (accessed on April 11, 2007).