Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/070425

A Comparative Study on the Algorithms for a Generalized Josephus Problem

Lei Wang[1] *and Xiaodong Wang*[2,3,*]

[1]Microsoft AdCenter, Bellevue, WA 98004, USA
[2]Faculty of Mathematics and Computer Science, Fuzhou University, 350002 Fuzhou , China
[3]Faculty of Computer Science, Quanzhou Normal University, 362000 Quanzhou, China

Received: 3 Dec. 2012, Revised: 14 Jan. 2013, Accepted: 12 Mar. 2013
Published online: 1 Jul. 2013

Abstract: The classic Josephus problem can be described as follows: There are n objects, consecutively numbered from 1 through n, arranged in a circle. We are given a positive integer k. Beginning with a designated first object, we proceed around the circle, removing every kth object. After each object is removed, counting continues around the circle that remains. This process continues until all n objects have been removed. In a generalized Josephus problem, a number of lives l is introduced for the problem. Each object has l lives. The object is removed only when it has been selected l times. In this paper we present a fast algorithm for generating the Josephus permutation for the generalized Josephus problem. Our new algorithm can also be applied to a more general case of the generalized Josephus problem where the lives for all objects can be different. The time and space complexities of new algorithms are O(lnlogk) and O(n) respectively. The computational experiments demonstrate that the achieved results are not only of theoretical interest, but also that the techniques developed may actually lead to considerably faster algorithms.

Keywords: Josephus permutation, hit sequence, full binary tree

1. Introduction

In this paper we study a new variant of the Josephus problem [1,2,3].

The classic Josephus problem is defined as follows. Suppose that n objects, consecutively numbered from 1 through n, are arranged in a circle and that we are given a positive integer k. Beginning with a designated first object, we proceed around the circle, removing every kth object. After each object is removed, counting continues around the circle that remains. This process continues until all n objects have been removed. The order in which the objects are removed from the circle defines a particular permutation of the integers $1,2,\cdots,n$. This permutation is usually called the (n,k) Josephus permutation. For example, the (6,4) Josephus permutation is (4 2 1 3 6 5).

There are also various generalizations on the classic Josephus problem in the literature [4,6]. In a recent research [5], the classic Josephus problem was generalized as follows. A uniform number of lives l is introduced for the problem. Each object has l lives. The object is removed only when it has been selected for l times. This new variant of Josephus problem was called Feline Josephus problem. In this generalized Josephus problem, corresponding to the order in which the objects are removed from the circle, there is also a particular permutation of the integers $1,2,\cdots,n$. This permutation is called the (n,k,l) Josephus permutation. For example, the (6,4,2) Josephus permutation is (4 2 1 6 5 3). The classic Josephus problem is a special case of the generalized Josephus problem when $l=1$.

We are interested in algorithms which, given integers n, k and l, generate the corresponding (n,k,l) Josephus permutation. An $O(n^2)$ time algorithm using $O(n^2)$ space to compute the last removed object of a given generalized Josephus problem with n objects is sketched in [5]. The time and space complexities of the algorithm are independent of the problem parameters k and l. The algorithm is conceptual feasible. Due to the $O(n^2)$ time and space costs of the algorithm, the algorithm is only of theoretical interest. It is not practical.

In this paper we present a practical algorithm for generating the general (n,k,l) Josephus permutation. Our

* Corresponding author e-mail: wangxiaodong@qztc.edu.cn

new algorithm can also be applied to a more general case of the generalized Josephus problem where the lives for all objects can be different. The time and space complexities of the new algorithm is $O(ln\log k)$ and $O(n)$ respectively. Throughout the paper, we assume that a real RAM with a uniform cost criterion is the computational model. In particular, this means that each arithmetic operation requires constant time.

The organization of the paper is as follows. In the following 4 sections we describe the algorithms and our computational experience with the algorithms for generating the general (n,k,l) Josephus permutation. In section 2 we describe a new variant tree based algorithm for generating the hit sequences of the generalized Josephus problem and their correctness and complexities. Based on the algorithm in section 2, an improved algorithm for generating the (n,k,l) Josephus permutations is proposed in section 3. In section 4 we give a computational study of the presented algorithms which demonstrates that the achieved results are not only of theoretical interest, but also that the techniques developed may actually lead to practical algorithms. Some concluding remarks are in section 5.

2. A Fast Algorithm

In this section we describe a fast algorithm for generating the (n,k,l) Josephus permutation. The algorithm is a variant of a tree based algorithm for classic Josephus problem proposed by Lloyd [3]. The new variant of the algorithm uses a heap like data structure and modified to solve the generalized Josephus problem.

2.1. The Data Structures of the Algorithm

The parameters to the generalized Josephus problem are the number of objects n, the skip factor k, and the number of lives l. We make no assumptions about these parameters. In particular, k or l or both, could be larger than n. When the parameters are given, the following notations are introduced for the generalized Josephus problem in [5].

Let h_i be the ith selected object.

The sequence $h_1,h_2,\cdots,h_{l\cdot n}$ is called the hit sequence.
Let x_i be the ith removed object.
The sequence $x_1,x_2,\cdots,x_n$ is called the kill sequence.

It is obvious that the kill sequence forms the (n,k,l) Josephus permutation of $(1,2,\cdots,n)$. In the following sections, an object is called *active* before it is removed, and *dead* after it is removed.

For the given generalized Josephus problem, there are n objects and the skip factor is k. In the following sections, we consider a more general Josephus problem, where the numbers of lives for each object are not necessary consistent. Assume at the beginning, the number of lives for the ith object be $l_i, 1 \le i \le n$. We use an array l to store the number of lives for each object. When the ith object is selected, the corresponding value of $l[i]$ is decreased by 1. If the value of $l[i]$ becomes 0, the ith object is removed.

Let $m = 2^{\lceil \log k \rceil}$ and $b = \lceil n/m \rceil$. The n objects $\{1,\cdots,n\}$ are divided evenly into b groups as follows.

$$\{1,\cdots,m\};\{m+1,\cdots,2m\};\cdots;\{(b-1)m+1,\cdots,n\}$$

Each group has m objects, except the last one.

For each group $j, 1 \le j \le b$, we use an array $a[j-1]$ of size $2m-1$ to store a full binary tree T_j, with integers in group j stored at its leaves $a[j-1][m-1], a[j-1][m],\cdots,a[j-1][2m-2]$. All of the b full binary trees $T_j, 1 \le j \le b$ constitute a two dimensional array a. The ith active object in T_j is the ith active integer encountered in a left-to-right traversal of the leaves of T_j. Also, a sequence of objects $y_1,\cdots,y_p$ is in leaf order with respect to the tree T_j, if for $1 \le i \le p$, y_i is the ith active integer in tree T_j. When the object in a leaf is removed, the corresponding leaf is *dead*, otherwise it is an *active* leaf.

The full binary tree T_j is represented as an array indexed binary tree $a[j-1]$. For each node $a[j-1][i], 0 \le i \le m-2$, its left child node is $a[j-1][2*i+1]$ and its right child node is $a[j-1][2*i+2]$. For each node $a[j-1][i]$, its parent node is $a[j-1][\lfloor (i-1)/2 \rfloor]$. Every node of each tree $a[j-1]$ contains a count of the number of active leaves in the subtree rooted at that node. This is the size of the node. When an object is removed, the size of each subtree containing the object is decreased since that object is now dead.

Notice that the meanings of the values stored in the leaf nodes and the non-leaf nodes of each tree T_j are different. It is obvious that the number of active leaves in the subtree rooted at a leaf is 1 if the leaf is active, and 0 if the leaf is dead. In order to distinguish the node type of each tree T_j, we change the sign of integer e stored in a leaf to $-e$. Whenever we read a negative integer $-e$, we know that it is a leaf node representing the object e.

In addition, the trees are doubly linked together in a circular fashion, with tree T_j linked after the tree T_{j-1} and before the tree T_{j+1}. Two arrays *prev* and *next* are used to represent the backward and the forward links respectively.

2.2. An Algorithm for Generating Hit Sequences

The algorithm consists of two parts: the initialization procedure and the main program. In the initialization

procedure, the data structure for the whole algorithm is initialized as follows.

Algorithm 2.1: INIT$(n,k,b,next,prev)$

$m \leftarrow 2^{\lceil \log k \rceil}, b \leftarrow \lceil n/m \rceil$
for $i \leftarrow 0$ **to** $b-1$
do {
  $s \leftarrow i*m, t \leftarrow (i+1)*m$
  for $j \leftarrow 0$ **to** $m-1$
    do $u[j] \leftarrow 0$
  for $j \leftarrow s$ **to** $\min(t-1, n-1)$
    do $u[j-s] \leftarrow -(j+1)$
  BUILD$(u, a[i])$
  $t \leftarrow (i+1)$ **mod** b
  $next[i] \leftarrow t, prev[t] \leftarrow i$

In the algorithm described above, the sub-algorithm BUILD(u,v) is used to build the full binary tree v described in the last subsection with its leaves given by the array u in the left-to-right order. A recursive function COUNT(v,i) is used to count the number of active leaves in the subtree rooted at node $v[i]$ in a bottom-up fashion. A leaf node with size 1 is recognized by the sign of its value.

Algorithm 2.2: BUILD(u,v)

procedure COUNT(v,i)
  if $m < i+2$
    then {
      if $v[i] < 0$
        then return (1)
        else return (0)
    else {
      $v[i] \leftarrow$ COUNT$(v, 2*i+1)$
      $v[i] \leftarrow v[i] +$ COUNT$(v, 2*i+2)$
      return $(v[i])$

main
  for $i \leftarrow 0$ **to** $m-1$
    do $v[m+i-1] \leftarrow u[i]$
  COUNT$(v,0)$

It is clear that the algorithm COUNT(v,i) requires $O(m) = O(k)$ time and so the algorithm COUNT(v,i) for each binary tree requires $O(k)$ time. Thus, the initialization procedure requires $O(n)$ total time.

By the set of b trees represented by the array a, the main program to generate the hit sequences consists of the following:

We will show later that the total cost of these operations to generate a hit object is $O(\log k)$. When two consecutive trees become small, we then combine the two trees into a new tree of height $\lceil \log k \rceil$. Such a combining operation is used to guarantee that the movement between trees when generating a hit object is limited to one or two. By amortizing the cost of the operations among the dead objects in the two original trees, we can maintain the $O(\log k)$ cost for each hit object.

Algorithm 2.3: ALGORITHM-L(n,b)

$i \leftarrow 0, r \leftarrow 0, t \leftarrow b-1$
while $i < n$
do {
  TREE(r,t)
  $j \leftarrow$ LEAF(r,t)
  HIT(i,j,r,t)

In the algorithm ALGORITHM-L(n,b), three subalgorithms are called. The sub-algorithm TREE(r,t) is used to locate the tree containing the next active object to be hit. The sub-algorithm LEAF(r,t) is used to find the next active leaf to be hit. The sub-algorithm HIT(i,j,r,t) is used to output the next hit object and adjust the data structure. In these three sub-algorithms, the variable r plays an important role. The variable r records the number of active objects to the right of currently hit leaf node in the full binary tree containing it. This variable is very useful to find the position of the next hit object in our data structures. With the changing of the hit sequences, the variable r changes accordingly. The variable t is the index of the full binary tree containing the active leaf to be hit.

Algorithm 2.4: TREE(r,t)

if $t = next[t]$ **and** $r < k$
  then $r \leftarrow k + a[t][0] - 1 - (k-r-1)$ **mod** $a[t][0]$
  else {
    while $r < k$
      do $t \leftarrow next[t], r \leftarrow r + a[t][0]$

Algorithm 2.5: LEAF(r,t)

$j \leftarrow 0, nxt \leftarrow r-k$
while $j+1 < m$
do {
  $rs \leftarrow$ SIZE$(a[t][2*j+2])$
  if $r - rs < k$
    then $j \leftarrow 2*j+2$
    else $j \leftarrow 2*j+1, r \leftarrow r-rs$
$r \leftarrow nxt$
return (j)

In the algorithms above, the function SIZE$(a[t][j])$ returns the number of active leaves in the subtree rooted at the node $a[t][j]$.

Algorithm 2.6: HIT(i,j,r,t)

$h \leftarrow -a[t][j]$
output (h)
if $l[h-1] > 1$
  then $l[h-1] \leftarrow l[h-1] - 1$
  else KILL(i,j,r,t)

The sub-algorithm KILL(i,j,r,t) is the heart of the whole algorithm which removes the dead leaf from the current binary tree and adjusts the current data structures.

Algorithm 2.7: KILL(i,j,r,t)

$a[t][j] \leftarrow 0$
while $j > 0$
  do $j \leftarrow \lfloor (j-1)/2 \rfloor, a[t][j] \leftarrow a[t][j]-1$
if $t \neq next[t]$
then
  if $a[t][0]+a[next[t]][0] = m$
    then $r \leftarrow r + a[next[t]][0]$; COMBINE$(t)$
  else if $a[t][0]+a[prev[t]][0] = m$
    then $t \leftarrow prev[t]$; COMBINE(t)

The algorithm COMBINE(t) will combine the two small trees $a[t]$ and $a[next[t]]$ into a new binary tree $a[t]$, and the tree $a[next[t]]$ will no longer be used.

Algorithm 2.8: COMBINE(t)

$x \leftarrow next[t], i \leftarrow 0$
for $j \leftarrow 0$ **to** $m-1$
  do **if** $a[t][m+j-1] < 0$
    then $u[i] \leftarrow a[t][m+j-1], i \leftarrow i+1$
for $j \leftarrow 0$ **to** $m-1$
  do **if** $a[x][m+j-1] < 0$
    then $u[i] \leftarrow a[x][m+j-1], i \leftarrow i+1$
BUILD$(u, a[t])$
$next[t] \leftarrow next[next[t]], prev[next[t]] \leftarrow t$

Now we consider the time complexity for the whole algorithm.

From the tests performed in the algorithm KILL(i,j,r,t), we know that at the beginning of each execution of TREE(r,t), either there is exactly one tree with active leaves, or for each tree $a[t]$ with active leaves we have,

$$a[t][0]+a[next[t]][0] \geq m \geq k.$$

It follows that there must be two trees with active leaves for the while-loop body of TREE(r,t) to be executed. Therefore, the while-loop body of each call of TREE(r,t) is executed at most twice.

In the leaf search algorithm LEAF(r,t), the search follows a path from the root of the tree $a[t]$ to a leaf $a[t][j]$. Since each tree created by the algorithm has a height of $\log k$, the cost per iteration of the while-loop in LEAF(r,t) is $O(\log k)$.

Next we consider the algorithm KILL(i,j,r,t). The time cost of the algorithm consists of two parts. The first part is a series of node value changes from the leaf $a[t][j]$ to the root $a[t][0]$. Since the height of the tree is $\log k$, the time of the first part is $O(\log k)$.

The second part of the time cost is incurred by the algorithm COMBINE(t). The cost of the algorithm is dominated by the execution time of BUILD(u,a[t]) to build the new full binary tree $a[t]$. We already know the time cost for this task is $O(k)$. This part of cost can amortized to each leaf of the tree as follows. In the algorithm COMBINE(t), trees $a[t]$ and $a[next[t]]$ are combined. When these trees are created, each has m active leaves. When the algorithm COMBINE(t) is performed, these two trees have m active leaves and m dead leaves. The $O(m)$ cost of the algorithm can thus be divided evenly among the objects stored at the m dead leaves. Each such object is thus assigned a cost of $O(1)$. Since each object is charged at most once, the total cost of the algorithm COMBINE(t) performed in the whole algorithm is $O(n)$.

Denote $L = \sum_{i=1}^{n} l_i$. Since there are total L iterations for the whole algorithm and each iteration requires $O(\log k)$ time. The time complexity of the whole algorithm is thus $O(L \log k)$.

The space used by the algorithm is obviously $O(n)$.

Theorem 1 *The algorithm* ALGORITHM-L(n,b) *for solving the generalized Josephus problem requires* $O(L \log k)$ *time and* $O(n)$ *space.*

3. An Improved Algorithm

By exploring the following property of the hit sequences for a given generalized Josephus problem, we can improved the algorithm described in the last section further. A similar property was mentioned in [5] with an informal proof. The fact mentioned there seems right, but the proof is not strict and incorrect. We will express this useful property by a theorem and give a strict proof here.

We divide the hit sequence into n segments by the successively removed objects as follows

$$r_1, r_2, \cdots, r_n.$$

The segment r_i is called round i hit sequence. The last object of r_i is the ith removed object.

Theorem 2 *For each fixed index* $i, 1 \leq i \leq n$, *the hit sequences* r_i *can be uniquely formulated as* $r_i = \alpha^s \beta$, *where* s *is a positive integer, and both of the sequences* α *and* β *are sequences of unique elements of* $\{1, 2, \cdots, n\}$.

*Proof.*We consider the hit sequence r_i for fixed index i. There has been $i-1$ objects removed before r_i. The current $n-i+1$ objects remained can be written in decreasing order by $p_1, p_2, \cdots, p_{n-i+1}$. The position p_d is the starting position where its predecessor x_{i-1} was just removed. Then the next object hit will be $h_1 = p_{1+(d+k-1) \bmod n-i+1}$. In general, suppose $|r_i| = t$, and for each $1 \leq j \leq t$, $d_j = 1+(d+jk-1) \bmod n-i+1$, $h_j = p_{d_j}$, then the hit sequence r_i can be written as $h_1, h_2, \cdots, h_t$.

If all of the objects $h_1, h_2, \cdots, h_t$ are distinct, then we know the lives of the object h_t must be 1, and it is the next removed object. This is the case of $\alpha = \emptyset$ and $\beta = \{h_1, h_2, \cdots, h_t\}$.

Otherwise, the lives of each object in $\{h_1, h_2, \cdots, h_t\}$ must be greater than 1. In this case, there must be a cycle in $\{h_1, h_2, \cdots, h_t\}$. Suppose the cycle be $\{h_u, h_{u+1}, \cdots, h_v\}$, $1 \le u \le v \le t$. This means that all of the objects $h_1, h_2, \cdots, h_v$ are distinct, and $h_{v+1} = h_u$ is the first object repeatedly hit. We can show that $u = 1$, since otherwise $h_{u-1} = h_v$ has been hit repeatedly before h_u. Therefore, we know that $\{h_1, h_2, \cdots, h_v\}$ forms a cycle of the hit sequence.

Suppose that $l^* = \min\{l[h_j] \mid 1 \le j \le v\}$, and $j^* = \min\{j \mid l[h_j] = l^*, 1 \le j \le v\}$. It is readily seen that h_{j^*} will be the next object to be removed. That is $h_{j^*} = h_t$. If we denote $\alpha = \{h_1, h_2, \cdots, h_v\}$, $\beta = \{h_1, h_2, \cdots, h_{j^*}\}$, and $s = l^* - 1$, then the hit sequence r_i can be written as $r_i = \alpha^s\beta$. We can easily seen that $s > 0$, since $l^* > 1$. In the special case of $j^* = v$, we have $r_i = \alpha^{s+1}$. □

From Theorem 2, we know that the lives of the current objects can be easily updated from the information in determining $r_i = \alpha^s\beta$.

First, if $p_j \notin \{h_1, h_2, \cdots, h_t\}$, then the lives of p_j remains unchanged.

From the proof of Theorem 2 we know that the last object of β is the removed object x_i of round r_i. So, if $p_j \in \beta$, then $l[p_j] = l[p_j] - s - 1$. If $p_j \in \alpha$ and $p_j \notin \beta$, then $l[p_j] = l[p_j] - s$.

According to the Theorem 2, we can improve the algorithm for solving the generalized Josephus problem to a new algorithm ALGORITHM-N(n, b) as follows.

The sub-algorithm HIT(i, j, r, t) of the algorithm ALGORITHM-L(n, b) is changed to follows.

Algorithm 3.1: HIT(i, j, r, t)

$h \leftarrow -a[t][j]$
$q[g] \leftarrow h, g \leftarrow g + 1$
if $g = 1$ **or** $l[h-1] < l[q[min]-1]$
  then $\begin{cases} min \leftarrow g - 1 \\ p[0] \leftarrow t, p[1] \leftarrow j, p[2] \leftarrow r \end{cases}$
if $l[q[min]-1] = 1$ **or** $g > 1$ **and** $h = q[0]$
  then $\begin{cases} \text{CHANGE}(t, j, r, g, min) \\ \text{KILL}(i, j, r, t) \end{cases}$

In the algorithm described above, we use an array q to record the hit sequence α of round r_i as stated in Theorem 2. A variable *min* is used to record the index of α with minimal lives. When an object with only one life is found or a cycle of the hit sequence is found, we then have determined $r_i = \alpha^s\beta$. With the minimal lives in α recorded by *min*, we can easily update the lives of the current objects according to Theorem 2 by algorithm CHANGE(t, j, r, g, min) as follows.

Algorithm 3.2: CHANGE(t, j, r, g, min)

$d \leftarrow l[q[min]-1]$
for $i \leftarrow 0$ **to** min
  do $l[q[i]-1] \leftarrow l[q[i]-1] - d$
for $i \leftarrow min + 1$ **to** $g - 2$
  do $l[q[i]-1] \leftarrow l[q[i]-1] - d + 1$
$g \leftarrow 0, t \leftarrow p[0], j \leftarrow p[1], r \leftarrow p[2]$

In the two sub-algorithms above, another array p is used to record the information t, j, r on the tree containing the next removed active object.

Other parts of the algorithm ALGORITHM-L(n, b) remains unchanged.

Suppose for each hit sequence r_i, the corresponding cyclic hit sequence stated in Theorem 2 be α_i, and the length of α_i be denoted as t_i, $1 \le i \le n$.

Denote $T = \sum_{i=1}^{n} t_i$.

Since there are totally T iterations for the whole modified algorithm and each iteration requires $O(\log k)$ time. The time complexity of the improved algorithm is thus $O(T \log k)$.

The space used by the algorithm is obviously $O(n)$.

Theorem 3 *The improved algorithm* ALGORITHM-N(n, b) *for solving the generalized Josephus problem requires* $O(T \log k)$ *time and* $O(n)$ *space.*

The modified algorithm is indeed an improvement on the algorithm described in the last section, since $T = \sum_{i=1}^{n} t_i$ is much smaller than $L = \sum_{i=1}^{n} l_i$. For example, if n and k are relatively prime, and each object has a uniform number of lives l, then the hit sequence will have the form $\alpha^{l-1}\beta$, where α and β are permutations of $\{1, 2, \cdots, n\}$ and β is the Josephus permutation for $l = 1$ [5]. So, in this case, the length of α equals to n, and $T = \sum_{i=1}^{n} t_i = n$, $L = \sum_{i=1}^{n} l_i = l * n$. The original algorithm requires $O(l * n \log k)$ time, while the improved algorithm requires $O(n \log k)$ time even though the number of lives l may tend to infinite.

4. Computational Experiments

In this section, we give some computational experiments on the performance of the algorithms for the generalized Josephus problem.

Table 1: Comparing algorithms for small k and l: Running times in seconds

n	k	l	*Alg*-R	*Alg*-L	*Alg*-N
10^2	4	6	0	0	0
10^3	8	9	0.031	0	0
10^4	16	12	1.81	0.109	0.047
10^5	32	15	177.029	1.342	0.421
10^6	64	18	*****	18.876	4.571
10^7	128	21	*****	259.974	50.653

Table 2: Comparing algorithms for small l and increasing k: Running times in seconds

n	k	l	*Alg*-R	*Alg*-L	*Alg*-N
10^5	10^2	15	175.048	1.685	1.951
10^5	10^3	15	174.969	2.201	2.591
10^5	10^4	15	175.172	2.652	2.824
10^5	10^5	15	174.829	2.793	0.639
10^5	10^6	15	174.096	2.808	0.671
10^5	10^7	15	174.268	2.901	0.641
10^5	10^8	15	174.845	2.917	0.671
10^5	10^9	15	174.564	2.917	0.655

Our computational experiments were carried out on a personal computer with Pentium(R) Dual Core CPU 2.10 GHz and 2.0 Gb RAM. The word size of the processor is $w = 32$.

We compare the three algorithms, the $O(n^2)$ time algorithm in [5], denoted by ALGORITHM-R(), the algorithm ALGORITHM-L() described in section 2 and the improved algorithm ALGORITHM-N() described in section 3 for 5 different test data sets.

In order to compare the algorithms with ALGORITHM-R(), the data sets are designed with a uniform number of lives for each test case.

The running times in seconds of the 3 algorithms for computing the generalized Josephus permutations with small parameters k and l are compared in Table 1. The times reported for each value of n indicate the results of running repeated 10 times to mitigate the effects of random fluctuations in system overhead.

In the following tables, the entries marked with symbol ***** could not be solved due to insufficient memory or time.

As expected, in our experiments for small k and l, the improved algorithm performs best with the number of objects n increasing. For this data set the Algorithm-R exhibits an $\Omega(n^2)$ time bound in the worst case.

Table 2 gives the running times in seconds of the 3 algorithms for computing the generalized Josephus permutations with small parameters l and increasing k.

For this data set, the time costs of the 3 algorithms have only small changes as the parameter k increasing.

Table 3: Comparing algorithms for n and k fixed and increasing l: Running times in seconds

n	k	l	*Alg*-R	*Alg*-L	*Alg*-N
10^5	10^7	10^2	174.721	17.893	0.655
10^5	10^7	10^3	174.252	176.997	0.687
10^5	10^7	10^4	174.252	1767.79	0.641
10^5	10^7	10^5	174.221	*****	0.655
10^5	10^7	10^6	796.755	*****	0.655
10^5	10^7	10^7	904.894	*****	0.641
10^5	10^7	10^8	910.572	*****	0.761
10^5	10^7	10^9	915.517	*****	0.686

Table 4: Comparing algorithms for n fixed and increasing k and l: Running times in seconds

n	k	l	*Alg*-R	*Alg*-L	*Alg*-N
10^5	3723359	10^2	173.955	19.921	0.655
10^5	41912239	10^3	173.784	198.588	0.671
10^5	126590963	10^4	173.987	1962.31	0.671
10^5	152585351	10^5	173.797	*****	0.671
10^5	198676927	10^6	174.268	*****	0.671
10^5	271830677	10^7	176.274	*****	0.702
10^5	273068023	10^8	200.711	*****	0.687
10^5	273068023	10^9	432.151	*****	0.702

Table 3 gives the running times in seconds of the 3 algorithms for computing the generalized Josephus permutations with parameters n and k fixed and increasing l.

As expected, in our experiments for increasing l, the time cost of the algorithm ALGORITHM-L() is increasing with the parameter l increasing.

While the algorithm ALGORITHM-R() and the improved algorithm ALGORITHM-N() are not affected by the increasing of the parameter l.

The improved algorithm ALGORITHM-N() performs best. For this data set the ALGORITHM-R() exhibits an $\Omega(n^2)$ time bound too.

Table 4 gives the running times in seconds of the 3 algorithms for computing the generalized Josephus permutations with parameters n fixed and increasing k and l.

The effect of increasing k is not significant for the 3 algorithms.

Table 5 gives the running times in seconds of the 3 algorithms for computing the generalized Josephus permutations with the 3 parameters n ,k and l increasing.

As expected, for this data set, the improved algorithm performs best with the 3 parameters n ,k and l increasing. The algorithm ALGORITHM-L() is affected significantly by the parameter l and ALGORITHM-R() exhibits an $\Omega(n^2)$ time bound in the worst case.

Table 5: Comparing algorithms for n, k and l increasing: Running times in seconds

n	k	l	*Alg*-R	*Alg*-L	*Alg*-N
10^2	195	10^2	0	0	0
10^3	1942	10^3	0.046	0.249	0.015
10^4	19420	10^4	2.013	29.702	1.575
10^5	194201	10^5	174.408	3795.01	0.702
10^6	1942007	10^6	*****	*****	8.253

5. Concluding Remarks

We have proposed an efficient algorithm for computing the generalized Josephus permutations. The time cost of the new algorithm is $O(T\log k)$ and $O(n)$ space is used. For some special cases of the generalized Josephus problem, $T=\sum_{i=1}^{n} t_i = O(n)$. So our new algorithm requires $O(n\log k)$ time in these cases. This is a big improvement on the previous algorithms.

For the general Josephus problem, whether the formula $T=\sum_{i=1}^{n} t_i = O(n)$ is always true is an open problem. We will investigate the problem further.

Acknowledgement

The authors acknowledge the financial support of Science and Technology of Fengze under Grant No.2009FZ24 and 2010FZ02 and the Haixi Project of Fujian under Grant No.A099. The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] W.W. Rouse Ball and H.S.M. Coxeter. *Mathematical Recreations and Essays*, Dover Publications (1987).
[2] R.L. Graham, D.E. Knuth and O. Patashnik. *Concrete Mathematics*, Addison Wesley (1994).
[3] E.L. Lloyd. An $O(n\log m)$ algorithm for the Josephus problem. *Journal of Algorithms* **4**(3), (1983) 262–270.
[4] A. M. Odlyzko and H. S. Wilf. Functional Iteration and the Josephus Problem. *Glasgow Math. J.* **33**, (1991) 235–240.
[5] F. Ruskey and A. Williams. The Feline Josephus Problem. *Theory of Computing Systems*, **50**, (2012) 20–34.
[6] N. Theriault. Generalizations of the Josephus problem. *Utilitas Mathamatica* **58**, (2000) 161–173.

Lei Wang, PhD in Computer Science from Georgia Institute of Technology 2011. Applied researcher at Microsoft. Has experience in computer science with emphasis in algorithm design. The areas of interest are approximation and randomized algorithms,mechanism design, market equilibrium computation.

Xiaodong Wang, Professor in Computer Science Department of Quanzhou Normal University and Fuzhou University,China. Has experience in computer science and applied mathematics. The areas of interest are design and analysis of algorithms,exponential-time algorithms for NP-hard problems,strategy game programming.