ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **12**, Number 1, 2009, 13–24

Usage of advanced data structure for improving efficiency for large (n, m) permutations inspired from the Josephus problem

Mirel COŞULSCHI, Mihai GABROVEANU, Nicolae CONSTANTINESCU

Faculty of Mathematics and Computer Science,
University of Craiova,Romania

E-mail: `{mirelc, mihaiug, nikyc}@central.ucv.ro`

Abstract. Choosing the right data structure has been proved many times to have a major role toward design of an optimal algorithm. In this paper, we will present two classical algorithms (together with their associated classical data structures, array and linked list) for finding the (n, m)-Josephus permutations, our contribution being materialized in the third algorithm and the usage of an interesting data structure, *binary indexed tree*, which combines the ideas from binary tree traversal algorithms with the idea of binary representation of an index.

Key words: Josephus problem, data structures, binary indexed tree, algorithm complexity.

1. Introduction

An interesting theoretical computer science and mathematical problem is the Josephus problem (or *Roman roulette*) [1] which has as starting point the legend ([1], [9]) about the famous Jewish historian from the first century, Titus Flavius Josephus [2], known also as apologist of priestly and royal ancestry:

---

[1] http://en.wikipedia.org/wiki/Josephus_problem
[2] http://en.wikipedia.org/wiki/Josephus

> In the Jewish revolt against Rome, Josephus and 39 of his comrades were holding out against the Romans in a cave. With defeat imminent, they resolved that, like the rebels at Masada, they would rather die than be slaves to the Romans. They decided to arrange themselves in a circle. One man was designated as number one, and they proceeded clockwise killing every seventh man. Josephus (according to the story) was among other things an accomplished mathematician; so he instantly figured out where he ought to sit in order to be the last to go. But when the time came, instead of killing himself he joined the Roman side.

In other words, there are n places arranged in a circle, numbered clockwise $1, 2, \ldots,$ n and occupied. Every m-th element is removed from the set (we must notice that the counting started at number 1). The first element eliminated is the m-th element; in order to exclude another element we have to count again m places, without taking into consideration the vacancy places (a vacancy place results after removal of some element at a previous step).

The first question is if someone would like to be the last survivor, then what place should him occupy initially? We will name that the basic Josephus problem, and the answer of the question will be given by the function $J_m(n)$. Another interesting problem is to find the order in which the initial elements were removed from the set ((n, m)-Josephus permutation). The number of the ith element excluded from the circular arrangement will be denoted as $J(n, m, i)$ $(n \geq 1, m \geq 1, 1 \leq i \leq n)$ [7].

Besides these two problems, there are also interesting related questions that were investigated in the literature ([10], [16], [11], [12], [15]).

2. Previous work

Knuth describes one of the best algorithms known so far ([6], [14]) for basic Josephus problem, whose complexity is $O(\log_{\frac{m}{m-1}}(mn-n))$. This elegant solution has the following ideas [6]:

a) Define a sequence of numbers $D_n^{(m)}$, where

$$D_n^{(m)} = \lceil \frac{m}{m-1} D_{n-1}^{(m)} \rceil, n \geq 1, D_0^{(m)} = 1.$$

b) Determine the least integer k such that $D_k^{(m)} > (q-1) \cdot n$.

c) Compute $J_m(n) = m \cdot n + 1 - D_k^{(m)}$.

In [5], the author present an improvement to this method, resulting an algorithm with $O(m + \log_{\frac{m}{m-1}}(\frac{n}{m}))$.

Another algorithm, whose complexity is $O(n \cdot \log(m))$, for finding the values of the function $J_m(n)$ was developed by Lloyd in 1983 ([13]).

In [14] there are studied some interesting mathematical properties regarding Knuth's method sketched above.

Augenstein et al [2] presents a systematic approach of the (n, m)-Josephus permutation (see also [16] and the rank tree usage [3]). For generating the sequence of numbers resulted from the elimination process, they have chosen de simulation approach. In their article it is presented the idea of using an *almost completely strictly binary tree* (ACSB).

Definition 1. An almost *completely strictly binary tree* is characterized by the following properties:

1. every node in the tree has either 0 or 2 sons;
2. for some K, every leaf in the tree is at level K or level $K + 1$;
3. if a node in the tree has a right descendant at level $K + 1$, then all its left descendants which are leafs are at level $K + 1$.

The algorithm depicted there has $O(n \cdot log(n))$ complexity given by the insert, delete and select operations into the ACSB tree.

3. Obtaining the (n, m)-Josephus permutations

In this section we will present three algorithms whose goal is getting the (n, m)-Josephus permutations. Two of them, the first ones, are well-known methods while the third is our contribution. The algorithms are introduced gradually in decreasing order of their associated time complexity, toward improving the efficiency of the generating process.

3.1. Algorithm 1: using a static data structure

The procedure *ExtractTable* (see Algorithm 1) uses for storage the array *table*, where $table_i = 1$ means that the i-th element is present while $table_i = 0$ means this element was cast out. Lines 2-4 initialize the values from array with 1 stating that, at the beginning, all the elements is present in the initial set. The inner *while* (lines 10-15) counts p elements, which are still present in the set. As an optimization, if $p >$ number of elements remained in the set, then it will be counted only $p \mod (n - count + 1)$ elements. The first *while* (lines 7-19) loops n times in order to extract all elements from the set.

The overall time complexity of Algorithm 1 is $O(n^2)$ while space complexity is $O(n)$. We will work toward improving this time complexity.

3.2. Algorithm 2: using a dynamic data structure

In the second algorithm (procedure *ExtractList*), we will use as an abstract data type (ADT) [3] a list implemented through a dynamically linked list instead of a static array. The extraction of an element from the set will materialize through the removal

[3]http://jupiter.kaist.ac.kr/~otfried/cs206/notes/ranktrees.pdf

of the element from the list, thus not being visited again at the next loop. In order to get a better overview, the actions related to the ADT list is only outlined, the details of implementation being skipped:

1. *CreateNode(k)* - allocate space for a node whose information is the value of parameter k;
2. *AddNode(node)* - appends the node at the end of the list;
3. *Locate(node, p, n; previous)* - starting from the current node, counts p elements, in order to locate the p-th node, and returns a reference to this and also to the previous node;
4. *RemoveNext(node)* - removes the next element of the current node;
5. *MoveNext(node; newNode)* - moves to the right, with a position, the current node, returning the value as *newNode*.

Algorithm 1 (n, m)-Josephus permutation variant 1

Input: n - the number of initial elements
k - the index of the first element removed
p - the step

Output: perm - the array with the extracted elements and their order

```
1: procedure ExtractTable(n, k, p; Perm)
2:     for i ← 1, n do
3:         table_i ← 1
4:     end for
5:     count ← 1, j ← k
6:     table_j ← 0, perm_count ← j
7:     while (count < n) do
8:         steps ← 0
9:         pr ← p mod (n − count + 1)
10:        while (steps < pr) do
11:            j ← next(j, n)
12:            if (table_j = 1) then
13:                steps ← steps + 1
14:            end if
15:        end while
16:        count ← count + 1
17:        table_j ← 0
18:        perm_count ← j
19:    end while
20: end procedure
```

The initialization step is performed now within lines 2-7, while the extraction step is fulfilled by lines 14-21. Due to the usage of ADT list, the time complexity of Algorithm 2 will be $O(n \cdot m')$, where $m' = m \mod n$.

Algorithm 2 (n, m)-Josephus permutation variant 2

```
Input: n - the number of initial elements
       k - the index of the first element removed
       p - the step
Output: perm - the array with the extracted elements and their order
 1: procedure ExtractList(n, k, p; perm)
 2:     for i ← 1, n do
 3:         tmp ← CreateNode(i)
 4:         if (tmp ≠ NULL) then
 5:             call AddNode(tmp)
 6:         end if
 7:     end for
 8:     tmp ← Locate(head, p, n; previous)
 9:     count ← 1
10:     perm_count ← tmp.value
11:     call RemoveNext(previous)
12:     call MoveNext(previous, tmp)
13:     n ← n − 1
14:     while (n > 0) do
15:         tmp ← Locate(tmp, p, n; previous)
16:         count ← count + 1
17:         perm_count ← tmp.value
18:         call RemoveNext(previous)
19:         call MoveNext(previous, tmp)
20:         n ← n − 1
21:     end while
22: end procedure
```

3.3. Algorithm 3: improving the generation algorithm

An improvement of the two previous presented methods that are already known and makes use of classical data structures, array and linked list respectively, would be to directly access the mth element and remove it instead of counting through m integers one by one.

We will make use of a data structure who, although have been introduced more than a decade ago, is still not widely known [4]. This data structure allows the update and query operations to be performed in logarithmic time.

The seminal ides of the method has its inception from the observation that every integer can be decomposed into a sum of appropriate powers of two - in the same way the sum of a sequence of elements can be computed from the sum of some subsequences subtotal.

Let's suppose we have a sequence of elements $v_i, i = 1, \ldots, n$, where v_i can be interpreted as the frequency of a value with index i, while c_i keeps the cumulative frequency for index i: $c_i = v_1 + v_2 + \ldots + v_i$.

The structure is named *binary indexed tree* (BIT), and is implemented with the help of an array a, where each element a_i will be equal with the sum of the elements from the sequence $i - 2^k + 1, \ldots, i$ (k is the number of trailing 0's from the binary representation of number i).

Table 1.

	1	2	3	4	5	6	7	8	9	10	11	12
v	1	0	4	1	2	0	1	2	1	2	4	1
c	1	1	5	6	8	8	9	11	12	4	18	19
a	1	1	4	6	2	2	1	11	1	3	4	8

Table 2. Table with subsequences corresponding to each element from a

	1	2	3	4	5	6	7	8	9	10	11	12
a	v_1	$v_{1\ldots2}$	v_3	$v_{1\ldots4}$	v_5	$v_{5\ldots6}$	v_7	$v_{1\ldots8}$	v_9	$v_{9\ldots10}$	v_{11}	$v_{9\ldots12}$

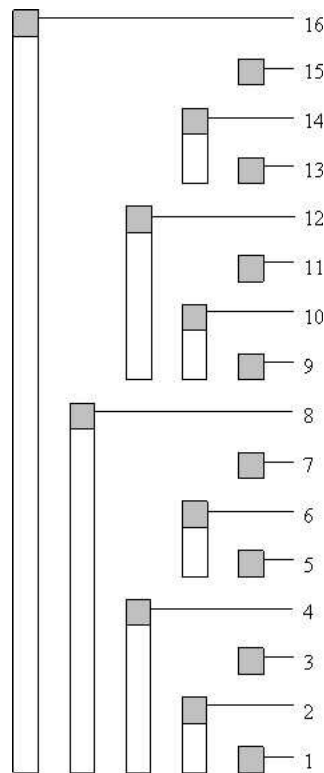

Fig. 1. The tree showing range of elements accumulated in their responsible node.

The reader can easily notice that $a_i = v_i$ if and only if i is an odd number. Also it is not necessary to keep the values of the vector v, because from the values of the binary indexed tree can be reconstructed these original values: $v_i = c_i - c_{i-1}$, while c_i can be obtained from the subroutine *GetSum*.

In the tree from Fig. 1, each bar represents the range of the elements from v whose cumulative total value is stored in the topmost position. (e.g. $a_{12} = v_{12} + v_{11} + v_{10} + v_9$). The branching ratio of each node corresponds to the number of trailing zeros in the associated binary representation, while the depth at each node is the Hamming weight of its binary index [4].

Among the basic operations defined for this data structure, we will concentrate only in this paper on three of them, used in the Algorithm 6: getting the cumulative frequency of an index, updating the table a in concordance with a new value v for a certain index, and finding the position for which the cumulative frequency has a specified value.

The name *binary indexed tree* is a result of the combination between the tree traversal algorithms and the binary representation of an index.

Updating the table. An important technical issue aims at obtaining the least significant 1 bit from the binary representation of a number. This operation is usefull for the navigation back and forth among the elements of the binary indexed tree, and can be done in several ways:

1. $k = k \wedge (-k)$ - bit AND ($\wedge$) between a number (k) and its two's complement ($-k$);
2. $k = k - (k \wedge (k-1))$ - difference between the current number (k) and bit AND ($\wedge$) between the same number (k) and its predecessor ($k-1$);
3. $k = k \wedge (2^z - k)$ - where z is a number having the property $2^z > N$.

Removing the least significant one bit of a number can also be done in more than one method, strongly connected with the above variants: $k = k - (k \wedge (-k))$ or $k = k \wedge (k-1)$.

Example 1. If $k = 10110100$, then $k - 1 = 10110011, k\text{and}(k-1) = 10110000$, while $k - (k\text{and}(k-1)) = 00000100, k = k\text{xor}(k\text{and}(k-1)) = 00000100$.

The number of iterations in the *while* group of instructions depends on the number of one bits from the index k. In order to update the table a with the new value v_i, it is necessary to update all the elements a_j where $a_j = \ldots + v_i + \ldots$ (subfrequencies a_j that includes/cover the value of v_i).

Example 2. If we have the $SetValue(16, 9, 3, A)$, first the procedure will update the value of a_9, $k = k + (k \text{ xor } (k \text{ and } (k-1))) = 10$, then, at the next step, a_{10} being updated, while $k = 10$. At the third step a_{12} is assigned with a new value and k becomes 16, while at the last step the a_{16} element is visited.

Getting the cumulative frequency. The subroutine *GetSum* returns the cumulative frequency of all elements between 1 and a specified upper limit, k. $k \leftarrow k \wedge (k-1)$ strip off the least significant 1 bit from the binary representation of the number k.

Algorithm 3 Set frequency value

Input: n - the total number of elements
k - the index of the element whose value must be set
value - the value to be set
a - the array with frequency

1: **procedure** SetValue($n, k, value, a$)
2: **while** ($k \geq n$) **do**
3: $a_k \leftarrow a_k + value$
4: $k \leftarrow k + (k \text{ xor } (k \text{ and } (k-1)))$
5: **end while**
6: **end procedure**

Example 3. By example, if $k = 7$ the *GetSum*(k) will be composed from $a_7 + a_6 + a_4 = v_7 + v_{5\ldots6} + v_{1\ldots4}$.

Algorithm 4 Get frequency sum

Input: k - the number of initial elements
a - the array with frequency

1: **function** GetSum(k, a)
2: $sum \leftarrow 0$
3: **while** ($k > 0$) **do**
4: $sum \leftarrow sum + a_k$
5: $k \leftarrow k \text{ and } (k-1)$
6: **end while**
7: **return** sum
8: **end function**

The number of the elements involved in the sum is less than or equal to the number of digits from the binary representation associated with the position, so the number of operations is $O(\log(n))$.

Finding a cumulative frequency. The *Search* algorithm (algorithm 5) is a modified version of binary search method. The function is searching for least element whose cumulative frequency is equal to *Value*. The prerequisites of this function is related to the absence of negative values (thus the cumulative frequencies are increasingly ordered) and the value for which the search is performed is present in the sequence (the search operation does not end with a failure).

Algorithm details. The procedure *ExtractBIT* (algorithm 6) is using a binary indexed tree for storing and retrieving the information related with adding/removing the elements from the set. As the reader noticed in the previous variants of this algorithm, it starts with the initialization step (lines 2-4). Here each element existing in the set is marked by adding value 1 to the table. In line 6 is stated the step when current element k is removed from the set, while at next line 8 the removal from binary indexed tree is realized.

Lines 13-26 set the limits where the search of the next element shall be performed. The complexity of this operation is $O(\log(n))$, thus the time complexity of the algorithm being $O(n \cdot \log(n))$. Its performances are better than the performances of algorithm 1 and 2 when $n \geq m$.

Algorithm 5 Search value

Input: left - the index of the search interval's left limit
right - the index of the search interval's right limit
value - the value to be searched
a - the array with frequency

1: **function** SEARCH($left, right, value, a$)
2: compute $span$ ▷ $span = 2^k$ where k is the least number such that $2^k \geq right$
3: $index \leftarrow left + 1$
4: **while** ($span > 0$) **do**
5: **if** $((index + span \geq right) \wedge (getSum(index + span, a) < value))$ **then**
6: $index \leftarrow index + span$
7: **end if**
8: $span \leftarrow span/2$
9: **end while**
10: **if** ($getSum(index, a) < value$) **then**
11: $index \leftarrow index + 1$
12: **end if**
13: **return** $index$
14: **end function**

Example 4. In order to understand better our approach, a sample will be chosen: we will start with $n = 13$, $k = 3$, $p = 4$ (number of elements is 13, the indices of the first element is 3, while the counting step is 4).

After the initialization phase (lines 2-4), the values of the elements from array b are:

Table 3. Initialization values

	1	2	3	4	5	6	7	8	9	10	11	12	13
v	1	1	1	1	1	1	1	1	1	1	1	1	1
b	1	2	1	4	1	2	1	8	1	2	1	4	1

The first element k is removed (line 6) and the update is reflected through the array b (line 8):

Table 4. Removing initial element

	1	2	3	4	5	6	7	8	9	10	11	12	13
v	1	1	0	1	1	1	1	1	1	1	1	1	1
b	1	2	0	3	1	2	1	7	1	2	1	4	1
perm	3	-	-	-	-	-	-	-	-	-	-	-	-

The result of the call of $search(left, right, steps, b, span)$ procedure is $k = 7$:

Table 5. Removing element $k = 7$

	1	2	3	4	5	6	7	8	9	10	11	12	13
v	1	1	0	1	1	1	0	1	1	1	1	1	1
b	1	2	0	3	1	2	0	6	1	2	1	4	1
perm	3	7	-	-	-	-	-	-	-	-	-	-	-

Algorithm 6 (n, m)-Josephus permutation variant 3

Input: n - the number of initial elements
k - the index of the first element removed
p - the step

Output: perm - the array with the extracted elements and their order

```
1:  procedure ExtractBIT(n, k, p; perm)
2:      for i ← 1, n do
3:          call SetValue(n, i, 1, b)
4:      end for
5:      count ← 1
6:      perm_count ← k
7:      count ← count + 1
8:      call SetValue(n, k, −1, b)
9:      while (count < n) do
10:         sumK ← GetSum(k, b)
11:         sumN ← GetSum(n, b)
12:         steps ← p mod sumN
13:         if (steps > 0) then
14:             if (steps ≥ (sumN − sumK)) then
15:                 left ← k, right ← n, steps ← steps + sumK
16:             else
17:                 left ← 0, right ← n, steps ← steps − (sumN − sumK)
18:             end if
19:         else
20:             steps ← sumK
21:             if (steps > 0) then
22:                 left ← 0, right ← n
23:             else
24:                 left ← k, right ← n, steps ← sumN
25:             end if
26:         end if
27:         k ← search(left, right, steps, b, span)
28:         perm_count ← k
29:         count ← count + 1
30:         call SetValue(n, k, −1, b)
31:     end while
32: end procedure
```

Then will follow $k = 11$, while at the next stage the value 2 is assigned to k:

Table 6. Removing element $k = 2$

	1	2	3	4	5	6	7	8	9	10	11	12	13
v	1	0	0	1	1	1	0	1	1	1	0	1	1
b	1	1	0	2	1	2	0	5	1	2	0	3	1
perm	3	7	11	2	-	-	-	-	-	-	-	-	-

The values at the last two phases are:

Table 7. Removing the last two elements

	1	2	3	4	5	6	7	8	9	10	11	12	13
v	0	0	0	1	0	0	0	0	0	0	0	0	0
b	0	0	0	1	0	0	0	1	0	0	0	0	0
perm	3	7	11	2	8	13	6	1	10	9	12	5	-

All the above algorithms have been implemented in *C* language and the extensive tests performed showed the efficiency of the last approach.

4. Conclusions

In this paper we have presented an efficient $O(n \cdot \log(n))$ algorithm for solving the Josephus problem. By using informal methods we have also showed the correctness of the algorithm. Another approach presented in Cormen et al [3], suggests to use the *interval tree* (augmented red-black tree) as a support data structure for determining the (n, m)-Josephus permutation. Our work differs from Augenstein et al [2] and above mentioned one, by the usage of another data structure, *binary indexed tree*, which is more easy to be implemented, occupies less memory space and has the same efficiency.

Acknowledgements. The work of two authors was supported by the Romanian National Council of Academic Research (CNCSIS) through the grant CNCSIS 55/ 2008.

References

[1] ALASDAIR R., MAC FHRAING (RANKIN R. A.), *The numbering of Fionn's and Dubhan's men, and the story of Josephus and the forty Jews*, Proc. Royal Irish Academy, Sect. A:52, 1948.

[2] AUGENSTEIN M., TENENBAUM A., *Program efficiency and data structures*, Proceedings of the eighth SIGCSE technical symposium on Computer science education, 1977.

[3] CORMEN T. H., LEISERSON C. E., RIVEST R. R., STEIN C., *Introduction to Algorithms*, Second Edition, McGraw-Hill Higher Education, 2002.

[4] FENWICK P. M., *A new data structure for cumulative frequency tables*, Software-Practice and Experience, vol. **24**, no. 3, pp. 327–336, 1994.

[5] GELGI F., *Time Improvement on Josephus Problem*, 2002.

[6] GRAHAM R. L., KNUTH D. E., PATASHNIK O., *Concrete Mathematics: A Foundation for Computer Science* , Second Edition, Reading, Addison-Wesley Massachusetts, 1994.

[7] HALBEISEN L., HUNGERBUHLER N., *The Josephus problem*, J. Thor. Nombres Bordeaux, vol. **9**, 1997.

[8] HENDERSON P. B., *The Josephus Flavius' problem*, ACM SIGCSE Bulletin, vol. **38**, Issue 2, 2006.

[9] HERSTEIN I. N., KAPLANSKY I., *Matters mathematical*, Second Edition, Chelsea Publishing Company, 246 pp., 1978.

[10] JAKÓBCZYK F., *On the generalized Josephus problem*, Glasgow Math. J., **14**, 1973.

[11] KNUTH D. E., *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, Third Edition, Reading, Addison-Wesley, 1997.

[12] KNUTH D. E., *The Art of Computer Programming, vol. 3: Sorting and Searching*, Second Edition, Reading, Addison-Wesley, 1998.

[13] LLOYD A. M., *An O(n log m) algorithm for the Josephus problem*, Journal of Algorithms, vol. 4, no. 3, 1983.

[14] ODLYZKO A. M., WILF H. S., *Functional Iteration and the Josephus Problem.*, Glasgow Math. J., **22**, 1991.

[15] SKIENA S., *Josephus' Problem.*, #1.4.3 in *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Reading, MA: Addison-Wesley, vol. **4**, no. 3, 1990.

[16] WOODHOUSE D., *Programming the Josephus problem*, ACM SIGCSE Bulletin, vol. **10**, Issue 4, 1978.