Array Rotation: Dolphin Algorithm

Nitin Verma
mathsanew.com

May 4, 2021

Consider an array a of n elements; the element type can be anything. Given any integer k, $1 \le k \le (n-1)$, we need to right-shift this array circularly by k elements. That means, if A_i denotes the initial value of $a[i]$ for all indices i, then elements of array a should be rearranged such that the following hold:

$$\begin{aligned} & (\forall i : 0 \le i < k : a[i] = A_{n-k+i}) \ \wedge \ (\forall i : k \le i < n : a[i] = A_{i-k}) \\ \Leftrightarrow \quad & (\forall i : 0 \le i < n : a[i] = A_{(i-k) \bmod n}) \end{aligned}$$

In words, each element of the array shifts right (to higher index) by k positions, while wrapping around the boundary at index $n-1$ if required. We will refer this process as "rotating the array by k".

Another way to look at this process is that, it interchanges the two blocks of elements which are at index ranges $[0, n-k-1]$ and $[n-k, n-1]$; so it can also be called "block interchange".

In this article, we will discuss the *Dolphin Algorithm* to perform this task, its proof and one modification to it. The algorithm seems to be first published in a 1966 paper by Fletcher and Silver [1], and was rediscovered and named such in [2]. Some places on internet also call it *Juggling Algorithm*.

Due to the rotation, if the element at index i ends up being placed at index j, we will refer such j as the "target-index" of i, and i as the "source-index" of j. Notice that the task of rotating an array is a way of rearranging its elements such that the element at index i gets moved to the target-index given by function $f(i) = (i + k) \bmod n$. We can also say that, an index i receives the element from source-index given by function $g(i) = (i - k) \bmod n$.

Now, consider the below sequence of indices in which each index is followed by its target-index:

$$S_i : i, f(i), f(f(i)), f(f(f(i))), \ldots \tag{1}$$

Since f can only take up to n values, this sequence must start repeating at some point. Note that the $(j+1)^{th}$ element $(j \geq 0)$ of this sequence is given by:

$$\begin{aligned} f(f(f(\ldots\{j \text{ times}\}(i)))) &= (((((i+k) \bmod n)+k) \bmod n)+k \ldots) \bmod n \\ &= (i+jk) \bmod n \end{aligned} \tag{2}$$

The Dolphin Algorithm makes use of the nature of this sequence, which we will understand further with the help of modular arithmetic. In the next section, we prepare the relevant mathematical results.

Modular Arithmetic

Please refer to the proof of theorem 2 in article titled *Multiples of an Integer Modulo Another Integer* [3]. Based on that, we can write the following theorem:

Theorem 1. *For any integers a, n with $n > 0$, $a \bmod n \neq 0$, and $g = gcd(a, n)$, denote $(ai) \bmod n$ as v_i for all integers $i \geq 0$, and n/g as t. Then:*

1. *$v_0, v_1, v_2, \ldots, v_{t-1}$ are all distinct, and form the set $G = \{0, g, 2g, \ldots, ((n/g)-1)g\}$,*
2. *for any integer $j \geq 0$, $v_{t+j} = v_j$. That is, in the sequence of v_i values, the subsequence $(v_0, v_1, v_2, \ldots, v_{t-1})$ keeps repeating.*

Note that we did not claim anything about which value v_i maps to which value in G. Further in this theorem, what can we say about values of the form $(ai+b) \bmod n$, for some integer $b \geq 0$? Note,

$$(ai+b) \bmod n = ((ai) \bmod n + b) \bmod n = (v_i + b) \bmod n$$

From theorem 1, we know that the v_i values only belong to the set G. Inspecting the set G, we know that if $0 \leq b < g$, then $0 \leq (v_i + b) < n$, and so:

$$(v_i + b) \bmod n = v_i + b$$

So, if $0 \leq b < g$, then:

$$(ai + b) \bmod n = v_i + b = (ai) \bmod n + b$$

That is, the sequence of values $(ai + b) \bmod n$ is obtained simply by adding b to each element in the sequence of values $(ai) \bmod n$. We can write the following conclusion using theorem 1:

Corollary 2. *For any integers a, b, n with $n > 0$, $0 \leq b < g$, $a \bmod n \neq 0$, and $g = gcd(a, n)$, denote $(ai + b) \bmod n$ as v'_i for all integers $i \geq 0$, and n/g as t. Then:*

1. *for all integers $i \geq 0$, $v'_i = (ai) \bmod n + b$,*
2. *$v'_0, v'_1, v'_2, \ldots, v'_{t-1}$ are all distinct, and form the set $G_b = \{b, g + b, 2g + b, \ldots, ((n/g) - 1)g + b\}$,*
3. *for any integer $j \geq 0$, $v'_{t+j} = v'_j$. That is, in the sequence of v'_i values, the subsequence $(v'_0, v'_1, v'_2, \ldots, v'_{t-1})$ keeps repeating.*

The symbols of a, n, i, j, g used in above theorems have been chosen to match those in the referred proof of [3]; they do not correspond to other objects of this article.

Dolphin Algorithm

We now move back to the discussion about sequence (1). As found in (2), the elements of sequence S_i are given by $(i + jk) \bmod n$, for $j = 0, 1, 2, \ldots$. Let $g = gcd(k, n)$ and $t = n/g$. Using corollary 2, we can conclude this about sequence S_i for $0 \leq i < g$:

(a) the first t elements of the sequence are all distinct, forming the set $G_i = \{i, g + i, 2g + i, \ldots, ((n/g) - 1)g + i\}$. Also, the first element simply equals i.

(b) after the first t elements, the subsequence of these t elements keeps repeating in the sequence S_i. That is, the $(t+1)^{th}$ element equals 1^{st} element (which is i), and so on.

(c) for all $j \geq 0$, the $(j + 1)^{th}$ element is given by $(jk) \bmod n + i$. So, the $(j + 1)^{th}$ element of the g sequences $S_0, S_1, \ldots, S_{g-1}$ are consecutive.

We will be referring to the subsequence of first t elements of each sequence S_i as "cycle C_i", which are nothing but elements of set G_i in some

order. Note that, the t^{th}/last value of C_i has its target-index as the first value of C_i; hence the name "cycle".

Now observe that, for the g sets $G_0, G_1, \ldots, G_{g-1}$:

$$G_i \cap G_{i'} = \emptyset, \quad \text{for all } 0 \leq i, i' < g \text{ with } i \neq i' \tag{3}$$

$$\bigcup_{i=0}^{g-1} G_i = \{0, 1, 2, \ldots, n-1\} \tag{4}$$

What we have so far discussed forms the mathematical basis of the Dolphin Algorithm, which we describe now. For rotating the array, that is to rearrange its n elements, the algorithm considers one cycle C_i at a time, for $i = 0, 1, 2, \ldots, g-1$. Recall that the cycle C_i consists of $t = n/g$ array indices starting with i, such that each index x is followed by its target-index $f(x)$.

The elements of array a which are located at indices of cycle C_i, can be rearranged as required by moving them as per the cycle's order. Now, for iterating over the cycle C_i, we can either step in the forward direction as:

$$i, f(i), f(f(i)), \ldots$$

or backward direction as:

$$i, g(i), g(g(i)), \ldots$$

In the first way, a source-index is followed by its target-index. In the second one, it is the reverse. Below method implemented in C shows a way to rearrange the elements indexed by cycle C_i using the backward iteration:

```
/* n added below as x-k can be negative */
#define g(x) ((x-k+n)%n)

/* Process one cycle C_i. k must satisfy: 1 <= k <= (n-1) */
void process_a_cycle(int a[], int n, int k, int i)
{
  int si, ti; /* source-index, target-index */
  int external;

  ti = i;
  si = g(ti);
  external = a[ti]; /* created a vacancy/hole at ti */

  /* loop-invariants:
       P1: ti is the target-index for si, i.e. g(ti)=si.
       P2: the hole is located at ti. */
  while(si != i)
  {
    /* fill-up the hole at ti, now creating a hole at si */
    a[ti] = a[si];
    ti = si;
    si = g(ti);
  }

  /* (P2: the hole is located at ti) holds, and ti has source-index
     (si = i) */
  a[ti] = external;
}
```

Notice how the elements could be shifted along the cycle while requiring extra memory for only one array element. The forward iteration can also be implemented slightly differently.

Now, due to (3) and (4) we know that if we rearrange each of the g cycles $C_0, C_1, \ldots, C_{g-1}$ as above, we will have rearranged all the n elements of the array. So, the following method must result in rotation of the array as required. For computing the gcd, it uses *Euclid's Algorithm*.

```
void dolphin(int a[], int n, int k)
{
  int i, g;

  g = gcd(n, k);

  for(i = 0; i < g; i++)
    process_a_cycle(a, n, k, i);
}
```

```
int gcd(const int X, const int Y)
{
  int x, y;

  x = X; y = Y;

  while(x != 0 && y != 0)
  {
    if(x > y)
      x = x%y;
    else
      y = y%x;
  }

  if(x != 0)
    return x;
  else
    return y;
}
```

Note that this algorithm moves each array element (except $a[i]$) only once, directly to its target-index, hence avoiding any intermediate moves. It was named "Dolphin Algorithm" by [2] because, "the movement of the array values looked like dolphins leaping out of the water and disappearing again at random places".

A Modification

The Dolphin Algorithm rearranges all n/g elements of one cycle before moving on to another cycle; there are total g such cycles to consider. We already know that among these g cycles of n/g elements each, no two have a common element. Further, from point (c) at the beginning of the last section, the $(j+1)^{th}$ element (for all $j \geq 0$) of all these cycles are consecutive, forming a block of g contiguous indices in the array. Starting positions of these blocks is given by the elements of cycle C_0, which are the set $G_0 = \{0, g, 2g, \ldots, ((n/g)-1)g\}$. There are total n/g such blocks.

This allows that we can process all the g cycles together, by moving together the elements at their $(j+1)^{th}$ index to the respective target-indices (which too must be contiguous). That is, we now move a block of g contiguous elements at a time, to its target location.

But this will require extra memory to store first g elements of the array, like we stored a single element in the method *process_a_cycle*(). The $g =$

$gcd(k, n)$ may be large requiring more memory. So, instead of processing all the g cycles together, we can pick up to c ($1 \leq c \leq g$) cycles at a time, where c can be a parameter. These picked c cycles should have their corresponding ($(j+1)^{th}$ in each cycle) indices contiguous in the array. Below is an implementation of this algorithm:

```
/* k must satisfy: 1 <= k <= (n-1) */
void cycling_blocks(int a[], int n, int k, int c)
{
  int i = 0, g;
  int si, ti; /* source-index, target-index as per cycle C_i */
  int external[c];

  g = gcd(n, k);

  while(i < g)
  {
    if(c > g-i)
     c = g-i;

    /* process these c cycles together: C_i,C_{i+1},...,C_{i+c-1} */

    /* notice the similarity between this shifting of blocks and
       shifting of single elements in process_a_cycle(). The
       loop-invariants are similar to that method. */

    ti = i;
    si = g(ti);
    memcpy(external, &a[ti], c*sizeof(int));
    /* above created a vacancy/hole of c elements, starting at ti */

    while(si != i)
    {
      memcpy(&a[ti], &a[si], c*sizeof(int));
      ti = si;
      si = g(ti);
    }

    memcpy(&a[ti], external, c*sizeof(int));
    i = i+c;
  }
}
```

The original Dolphin Algorithm visits the entire range of array in hops of size k, while moving one element at a time. Hence, (depending upon k, n) it may not show good spatial locality of reference. This modified version (if

$g > 1$ and $c > 1$) reduces the number of visits across the entire array, and hence must exhibit more spatial locality.

We may call this modified algorithm as "Cycling Blocks Algorithm", because blocks of contiguous elements are shifted along a cycle. Note, for $g = 1$, the blocks can have a single element only, and so it becomes equivalent to the original Dolphin Algorithm.

The idea of processing the g cycles together was also discussed briefly in [4] (section 2.3.3), which also mentions about its use in [1].

■

References

[1] W. Fletcher, R. Silver. *Interchange of Two Blocks of Data.* C. ACM, Vol 9 (5) (1966), 326.

[2] D. Gries, H. Mills. *Swapping Sections.* Technical Report (Cornell University), 81-452 (1981). `https://hdl.handle.net/1813/6292`.

[3] Nitin Verma. *Multiples of an Integer Modulo Another Integer.* `https://mathsanew.com/articles/multiples_of_an_integer_modulo_another_integer.pdf` (2020).

[4] J. Bojesen, J. Katajainen. *Managing Memory Hierarchies (MSc Thesis).* `http://hjemmesider.diku.dk/~jyrki/PE-lab/Jesper/bojesen00.ps` (2000).