An Analysis of Two In-Place Array Rotation Algorithms

CHING-KUANG SHENE

Department of Computer Science, Michigan Technological University, Houghton, MI 49931–1295, USA
Email: shene@mtu.edu

This paper presents a complexity analysis of two STL in-place rotation algorithms. If an array of n elements is rotated to the right Δ positions, the first STL version, which uses forward iterators, uses $n - \gcd(n, \Delta)$ swaps, while the second version, which uses random access iterators, uses only $n+\gcd(n, \Delta)$ array element movements. This paper also proves the optimality of the second version. A performance comparison is included.

Received January 20, 1997; revised January 16, 1998

1. INTRODUCTION

Rotating an array is a popular exercise of array/string processing in CS1 and CS2 classes and it is also part of many commonly seen libraries such as STL [1, 2]. Two types of solutions are available, one requiring an auxiliary array and the other being in-place. We shall present an analysis of two in-place rotation algorithms available in the STL distribution.

Without loss of generality, we shall assume that the array to be rotated has n integers $(0, 1, 2, \ldots, n-1)$ and use Δ to denote a rotation offset. Thus, rotating an array of six $(n = 6)$ elements $(0, 1, 2, 3, 4, 5)$ to the right four positions $(\Delta = 4)$ yields $(2, 3, 4, 5, 0, 1)$. Since rotating an array of n elements to the left $n - \Delta$ positions is equivalent to rotating the same array to the right Δ positions, only right rotations will be considered.

One way of measuring the complexity of a rotation algorithm is to count the number of array element movements, since these are the only explicitly involved operations. An array element movement involves either assigning a value into an array element or copying an array element to elsewhere. For simplicity, we shall use movement for array element movement.

A trivial in-place algorithm rotates the array to the right one position Δ times. This uses $(n + 1)\Delta$ movements. Another solution is using an auxiliary array. The last Δ elements are saved to an auxiliary array, the remaining $n-\Delta$ elements are moved right, and finally the saved Δ elements are copied back from the auxiliary array to the beginning of the given one. It uses $n + \Delta$ movements.

There are two in-place algorithms in the STL distribution. It will be shown that the first STL rotation algorithm, which uses forward iterators, uses $n - \gcd(n, \Delta)$ array element swaps. Note that each swap needs three movements. The second STL version, which uses random access iterators, can be derived from an analysis of permutations. We shall prove that this version uses only $n + \gcd(n, \Delta)$ movements and, in fact, we shall also show that it is optimal. This version is faster than the first one and theoretically it is also faster than the one mentioned earlier which uses an auxiliary array. Since the GCD computation may increase the computational cost, an experimental analysis is necessary to determine its impact.

In what follows, Section 2 provides an analysis of the first STL version, Section 3 derives the second version and presents a variation in which the GCD computation is implicit, Section 4 proves its optimality, Section 5 compares these algorithms, and Section 6 contains our conclusion.

2. THE FIRST STL VERSION

Figure 1 is a version equivalent to the first STL rotation algorithm in which `SWAP()` is a macro for swapping two given elements. It is known that the number of swaps is no more than n [1]. The aim of this section is to show that the number of swaps is actually $n - \gcd(n, \Delta)$.

This algorithm is based on two types of section-swapping. Given an array of n elements and a number $k < n$, the first type subdivides the array, from right to left, into $\lfloor n/k \rfloor + 1$ sections with $\text{mod}(n, k)$ elements in the first section and k elements in each of the remaining sections. The second type also subdivides the array into $\lfloor n/k \rfloor + 1$ sections; however, the second section has $\text{mod}(n, k)$ elements while each of the remaining sections has k elements.

The first type of section-swapping, working from right to left, swaps two adjacent sections until the left section is the first one. After completing this step, elements in the right-most $\lfloor n/k \rfloor - 1$ sections are rotated to the right k positions and the original rotation problem is reduced to rotating the new array of the first two sections (of $k + \text{mod}(n, k)$ elements) to the right k positions, or equivalently to the left $\text{mod}(n, k)$ positions.

The second type of section-swapping, also working from right to left, swaps sections with the first one until the second section is reached. After completing this step, the original

```
void  rotate(char a[], int size, int offset)
{
     int left, middle, right;

     right = size - 1;
     left = middle = right - offset;
     for (;;) {
          SWAP(a[left], a[right]);
          left--;
          right--;
          if (right == middle) {
               if (left < 0)
                    return;
               middle = left;
          }
          else
               if (left < 0)
                    left = middle;
     }
}
```

FIGURE 1. The first STL rotation algorithm: STL_1.

first section is moved to the end, the original third section is moved to the first, and all other sections, except for the second one which is fixed, are moved to the left one section. As a result, except for the $k + \text{mod}(n, k)$ elements in the first two sections, all other elements are rotated to the left k positions. Thus, the problem is reduced to rotating the array of the first two sections to the right $\text{mod}(n, k)$ positions and this can be handled by the first type.

The algorithm in Figure 1, beginning with the first type, alternately uses these two types to rotate the array. Let $[n, k]_i$ denote the type i ($i = 1$ or 2) section-swapping of array size n and section size k. For convenience, we shall use $a \bullet b$ to denote $\text{mod}(a, b)$. Note that $(a + b) \bullet b = a \bullet b$. Then, the rotation process can be described as follows:

$$
\begin{aligned}
[n, \Delta]_1 &\Rightarrow [\Delta + n \bullet \Delta, n \bullet \Delta]_2 \\
&\Rightarrow [n \bullet \Delta + \Delta \bullet (n \bullet \Delta), \Delta \bullet (n \bullet \Delta)]_1 \\
&\Rightarrow [\Delta \bullet (n \bullet \Delta) + (n \bullet \Delta) \bullet (\Delta \bullet (n \bullet \Delta)), \\
&\quad\ (n \bullet \Delta) \bullet (\Delta \bullet (n \bullet \Delta))]_2 \\
&\cdots \\
&\Rightarrow [n_j, k_j]_i
\end{aligned}
$$

This process will continue until $n_j \bullet k_j = 0$. In this case, both `right == middle` and `left < 0` are true and the function returns.

The section sizes in the above sequence are n, Δ, $n \bullet \Delta$, $\Delta \bullet (n \bullet \Delta)$, $(n \bullet \Delta) \bullet (\Delta \bullet (n \bullet \Delta))$ and so on, where n is added for convenience. It is not difficult to see that the ith term is the remainder of dividing the $(i-2)$th term by the $(i-1)$th term. This is exactly the Euclidean division algorithm for computing the GCD: $\gcd(a, b) = \gcd(b, a \bullet b)$ if $a \bullet b \neq 0$; otherwise, $\gcd(a, b) = b$. Therefore, if n_j is a multiple of k_j in the above sequence, $k_j = \gcd(n, \Delta)$.

Note that each decrement of variable `right` corresponds to exactly one swap. Since the un-rotated sections are always the left-most two and since the last iteration has section size $\gcd(n, \Delta)$, variable `right` stops at location $\gcd(n, \Delta)$ and the value of `left` is less than zero. Hence, the number of swaps is $n - \gcd(n, \Delta)$. In summary, we have the following proposition.

PROPOSITION 2.1. *The STL rotation algorithm using forward iterators uses* $n - \gcd(n, \Delta)$ *swaps to rotate an array of* n *elements to the right* Δ *positions.*

3. THE SECOND STL VERSION

In this section, we shall derive the second STL rotation algorithm (Subsection 3.1) and prove that it uses $n + \gcd(n, \Delta)$ movements and one extra memory location. This algorithm computes $\gcd(n, \Delta)$ as a preprocessing step. A variation which does not compute $\gcd(n, \Delta)$ explicitly is presented in Subsection 3.2.

3.1. Deriving the algorithm

For convenience, we shall use $\mathbf{Z}_n$, the additive group with modular arithmetic, as the domain of a rotation throughout this section. A rotation can be considered as a one-to-one function $f : \mathbf{Z}_n \mapsto \mathbf{Z}_n$ such that, for any $i \in \mathbf{Z}_n$, $f(i)$ is equal to $(i + \Delta)$ modulo n (i.e. $f(i) \equiv i + \Delta$ (modulo n)). Thus, a rotation is a special permutation on $\mathbf{Z}_n$. Since every permutation can be decomposed into disjoint cycles [3], a rotation can also be decomposed into disjoint and shorter rotations, and our problem is reduced to finding these disjoint cycles.

Consider an arbitrary element $i \in \mathbf{Z}_n$. Starting with i, it is mapped to $f(i)$, $f(i)$ is mapped to $f(f(i)) = f^2(i)$ and

```
void  Rotate(char a[], int size, int offset)
{
     int Cycles, Moves, From, To, i;
     char Save;

     Cycles = GCD(size, offset);
     Moves = size / Cycles;
     for (i = 0; i < Cycles; i++) {
          To = i;
          Save = a[To];
          From = To - offset + size;
          for (int j = 1; j < Moves; j++) {
               a[To] = a[From];
               To = From;
               From -= offset;
               if (From < 0) From += size;
          }
          a[To] = Save;
     }
}
```

FIGURE 2. The second STL rotation algorithm: $\mathbf{STL}_2$.

so on. Since $\mathbf{Z}_n$ is finite, the sequence $i = f^0(i)$, $f^1(i)$, $f^2(i), \ldots$, cannot be of infinite length and must repeat itself at a certain point. Let k be the minimum value such that $i = f^k(i)$ holds. Then, $i \equiv i + k\Delta$ (modulo n) and there exists $p \geq 0$ such that $i + k\Delta = pn + i$ holds. Therefore, $k\Delta = pn$, which is independent of i. Since $k\Delta$ is a multiple of both Δ and n and since k is the smallest value such that $k\Delta = pn$ holds, $k\Delta$ is the LCM (i.e. least common multiple) of Δ and n. Since $\text{lcm}(n, \Delta) = n\Delta/\gcd(n, \Delta)$, $k = n/\gcd(n, \Delta)$. Therefore, starting with i, it always takes $n/\gcd(n, \Delta)$ steps to return to i. This is a cycle of length $n/\gcd(n, \Delta)$. Since there are n elements in $\mathbf{Z}_n$, there are $\gcd(n, \Delta)$ cycles. The following lemma summarizes this fact.

LEMMA 3.1. *Cycle length. Any rotation on $\mathbf{Z}_n$ can be decomposed into* $\gcd(n, \Delta)$ *disjoint cycles each of which has* $n/\gcd(n, \Delta)$ *elements.*

EXAMPLE 1. Consider rotating $(0, 1, 2, 3, \ldots, 7)$ to the right six positions. This rotation can be decomposed into $\gcd(8, 6) = 2$ cycles each of which has $8/2 = 4$ elements as shown below.

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 6 & 7 & 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 & 6 \\ 6 & 0 & 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 3 & 5 & 7 \\ 7 & 1 & 3 & 5 \end{pmatrix}$$

After knowing that a rotation can be decomposed into disjoint cycles, we shall next find these cycles. Fortunately, finding one element for each cycle is enough, since the remaining ones can be generated from this 'representative'. The following lemma shows that elements in the range of 0 and $\gcd(n, \Delta) - 1$ belong to different cycles.

LEMMA 3.2. *Representative elements. Let i and j be two distinct integers in the range of* 0 *and* $\gcd(n, \Delta) - 1$. *Then, i and j belong to different cycles.*

Proof. If i and j belong to the same cycle, then $j = i + k\Delta$ (modulo n) holds for some $k > 0$. Thus, $i + k\Delta = pn + j$ holds for some $p \geq 0$. If $i > j$, we have $0 \leq i - j = pn - k\Delta < \gcd(n, \Delta)$. Dividing this expression by $\gcd(n, \Delta)$ yields the following:

$$0 \leq \frac{i-j}{\gcd(n,\Delta)} = \frac{n}{\gcd(n,\Delta)}p - \frac{\Delta}{\gcd(n,\Delta)}k < 1.$$

Since both $n/\gcd(n, \Delta)$ and $\Delta/\gcd(n, \Delta)$ are integers, the above must be identical to zero, and $i - j = 0$ holds. This is a contradiction and hence i and j cannot be members of the same cycle. The same argument works for the case of $j > i$. □

Therefore, the ith cycle contains and can be generated by i, where $0 \leq i < \gcd(n, \Delta)$. Example 1 has two cycles generated by 0 and 1. Since these cycles are disjoint, they can be rotated separately. Since one temporary location is needed to complete a rotation, rotating a cycle of length k uses $k + 1$ movements. Since there are $\gcd(n, \Delta)$ cycles, each of which has $n/\gcd(n, \Delta)$ elements, the total number of movements is $n + \gcd(n, \Delta)$.

REMARK 1. There is an interesting and different derivation of this algorithm. Consider the cyclic subgroup H generated by $\Delta \in \mathbf{Z}_n$. It can be shown with the technique in Lemma 3.1 that H is of order $n/\gcd(n, \Delta)$. Hence, by the Counting Lemma [3], the number of co-sets with respect to H is $\gcd(n, \Delta)$ and each co-set is of order $n/\gcd(n, \Delta)$. The co-set of $i \in \mathbf{Z}_n$ contains elements of

form $i + p\Delta$ (modulo n), where $0 \leq p < n/\gcd(n, \Delta)$. By Lemma 3.1, these elements are all distinct and form a cycle, and by Lemma 3.2, $0, 1, \ldots, \gcd(n, \Delta) - 1$ are 'representatives' in different co-sets. Therefore, each co-set (and H itself) is a cycle.

Figure 2 is an algorithm based on the above idea, where variables `size`, `offset`, `Cycles` and `Moves` correspond to n, Δ, $\gcd(n, \Delta)$ and $n/\gcd(n, \Delta)$. Each iteration of the outer `for` handles one cycle and the inner `for` rotates that cycle.

3.2. A variation

Recall that elements $0, 1, \ldots, \gcd(n, \Delta) - 1$ belong to different cycles (Lemma 3.2). As a result, the minimum of all values that `From` can take is $\gcd(n, \Delta)$. Therefore, a new variable `Bound` is used and is updated when `From` < `Bound` holds. Variable `Start`, with initial value zero, is the 'representative' of the current cycle and is always less than `Bound`. Figure 3 is an algorithm based on this idea. The `if` part is equivalent to the inner `for` loop in Figure 2. If the new value of `From` is not equal to the value of `Start`, rotation of the current cycle has not yet been completed; otherwise, the `else` part starts a new cycle.

4. OPTIMALITY

This section proves that every algorithm that rotates an array of n elements to the right Δ positions with one extra memory location performs no less than $n + \gcd(n, \Delta)$ movements. Thus, the algorithm presented in Subsection 3.1 is optimal. In the following, for each rotation algorithm, we construct an associated graph whose edges represent the movements of array elements. Then, we shall show that of all these associated graphs, the one with minimum number of edges consists of a set of $\gcd(n, \Delta)$ disjoint cycles each of which has $n/\gcd(n, \Delta)$ vertices. The existence of such a rotation algorithm is obvious, since the associated graph of the algorithm in Subsection 3.1 satisfies this condition.

Let $\mathcal{A}$ be an algorithm that rotates $V = (0, 1, 2, \ldots, n-1)$ to the right Δ positions. The associated graph of $\mathcal{A}$, $G_{\mathcal{A}} = (V, E)$, is constructed as follows. The set of vertices is V. If $\mathcal{A}$ moves i to j, or if $\mathcal{A}$ moves an element i to a temporary location and then moves it to j, a directed edge from i to j is added to the edge set E. Therefore, the number of movements is equal to or larger than $|E|$. We shall assume that there are no loops (i.e. directed edges whose starting and ending vertices are identical). This assumption is reasonable because such movements are redundant in any algorithm. Note that $G_{\mathcal{A}}$ is not simple, since there may be more than one edge of the same direction between two vertices. This construction establishes a mapping from the set of all rotation algorithms that rotate $(0, 1, 2, \ldots, n-1)$ to the right Δ positions to the set of their associated graphs. However, this mapping is not one-to-one, since two or more algorithms may have the same associated graph.

The following are important properties of an associated graph.

LEMMA 4.1. *Let $G_{\mathcal{A}}$ be the associated graph of an algorithm $\mathcal{A}$. Let* $\text{in}(v)$ *and* $\text{out}(v)$ *be the in-degree and out-degree of vertex $v \in V$. Then, we have*

1. *For every vertex $v \in V$, $\text{in}(v) = \text{out}(v) > 0$ holds.*
2. *The minimum number of edges $G_{\mathcal{A}}(V, E)$ can have is $|E| = |V|$.*
3. *If $G_{\mathcal{A}}(V, E)$ has the minimum number of edges, then $\text{in}(v) = \text{out}(v) = 1$ holds for every vertex, and $G_{\mathcal{A}}(V, E)$ consists of $\gcd(n, \Delta)$ directed cycles of length $n/\gcd(n, \Delta)$.*

Proof. For each location i, since the content of i is moved to a new location and a new value is moved into i from elsewhere, there is an edge starting at i and an edge with i its destination. Hence, $\text{in}(i) = \text{out}(i)$. If the in-degree (and out-degree) of a vertex, say i, is zero, then algorithm $\mathcal{A}$ never moves anything into (and out from) i. This can only happen when $\Delta = 0$. Thus, Property (1) holds.

Since $2|E| = \sum_{v \in V}(\text{in}(v) + \text{out}(v))$, from Property (1), $2|E| \geq \sum_{v \in V}(1 + 1) = 2|V|$ and $|E| \geq |V|$. Therefore, the minimum value of $|E|$ is $|V|$ and Property (2) holds.

The first part of Property (3) is easy to prove. If there is a vertex p with $\text{in}(p) = \text{out}(p) = 1 + k$, where $k > 0$, then $2|E| = (\text{in}(p) + \text{out}(p)) + \sum_{v \in V - \{p\}}(\text{in}(v) + \text{out}(v)) = 2(1 + k) + 2(|V| - 1) = 2(k + |V|) > 2|V|$, which is a contradiction.

Let $p \rightarrow q \rightarrow \cdots \rightarrow r$ be a path with maximum length. Since $\text{in}(p) = \text{out}(p) = 1$, there must be an edge $x \rightarrow p$. If $x \neq r$, then we have two cases to consider. First, if x is not a vertex on the path, then the path length is extended by one, causing a contradiction. Second, if x is a vertex on the path, then the out-degree of x becomes 2, violating the first part. Therefore, x must be r and we have a directed cycle. Note that since $\text{in}(v) = \text{out}(v) = 1$ holds for every vertex, this cycle can be removed from $G_{\mathcal{A}}(V, E)$. Repeating this process, $G_{\mathcal{A}}(V, E)$ will eventually be decomposed into a set of disjoint directed cycles.

If i is an arbitrary vertex of a cycle, its content is moved to $i + \Delta$ (modulo n). By Lemma 3.1, the length of this cycle is $n/\gcd(n, \Delta)$ and there are $\gcd(n, \Delta)$ cycles. □

Since the algorithm presented in Section 3 can be decomposed into disjoint directed cycles, the existence of an algorithm whose associated graph satisfies Property (3) is obvious. Therefore, the minimum number of edges is achievable and we have the following theorem.

THEOREM 4.1. *Optimality. Since rotating a cycle uses one temporary location, the minimum number of movements performed by an algorithm that rotates* $(0, 1, 2, \ldots, n-1)$ *to the right Δ positions is $n + \gcd(n, \Delta)$.*

5. AN EXPERIMENTAL ANALYSIS

Since the number of movements might not completely reflect the time complexity of an algorithm due to the fact that there are other supporting computations, we shall look at the actual running time data in this section. Four algorithms are

```
void  rotate(char a[], int size, int offset)
{
     int Start = 0, Bound = size;
     int Dist = size - offset;
     int To = Start, From = To + Dist;
     char Save = a[To];

     for (;;)
          if (From != Start) {
               a[To] = a[From];
               if (From < Bound) Bound = From;
               To = From; From -= offset;
               if (From < 0) From += size;
          }
          else {
               a[To] = Save;
               Start++;
               if (Start >= Bound) return;
               To = Start; From = To + Dist;
               Save = a[To];
          }
}
```

FIGURE 3. A variation of $\mathbf{STL}_2$: $\mathbf{STL}_3$.

compared: (1) $\mathbf{STL}_1$: the first STL version using forward iterators (Figure 1), (2) $\mathbf{STL}_2$: the second STL version using random access iterators and the Euclidean algorithm for computing GCD (Figure 2), (3) $\mathbf{STL}_3$: a variation of $\mathbf{STL}_2$ in which the GCD computation is incorporated into the algorithm body (Figure 3) and (4) **Space**: a simple algorithm using an auxiliary array. The machine with which performance data are collected is an SGI Indigo^2 with a 200 MHz MIPS R4400 CPU and the compiler is SGI's C compiler with option set to `-O2`.

5.1. A general comparison

In this general case, n and Δ are not relatively prime (i.e. $\gcd(n, \Delta) > 1$). The array to be rotated contains character data with $n = 5000$, 10,000, 15,000 and 20,000. For each input size n, the offset values are $n/10$, $2n/10$, $3n/10$, $4n/10$ and $n/2$. For each pair of n and Δ, the rotation function is called 10,000 times and the user time of one call is recorded. Although there are other methods for computing the GCD of two positive integers [4], the Euclidean algorithm is used in $\mathbf{STL}_2$.

Figure 4 shows the timing data of $n = 20{,}000$. It is clear that **Space** is uniformly faster than all three other algorithms except for $\Delta = n/2$ in the $\mathbf{STL}_1$ case. This is because $\mathbf{STL}_1$ only uses $n - \gcd(n, \Delta) = n/2$ swaps ($1.5n$ movements) if $\Delta = n/2$ (Section 2). On the other hand, the case of $\Delta = n/2$ is the slowest for $\mathbf{STL}_2$. This is perhaps because the cost of initializing the inner `for` loop $n/2$ times is high. Except for this extreme case, **Space** is the fastest, followed by $\mathbf{STL}_2$, then $\mathbf{STL}_1$ and $\mathbf{STL}_3$ is the slowest. Therefore, making the GCD computation implicit is perhaps not worth it, since the cost of supporting computations is too high. Note that $\mathbf{STL}_3$'s timing increases very slowly as Δ increases. Perhaps some other optimization techniques could be used to accelerate $\mathbf{STL}_3$.

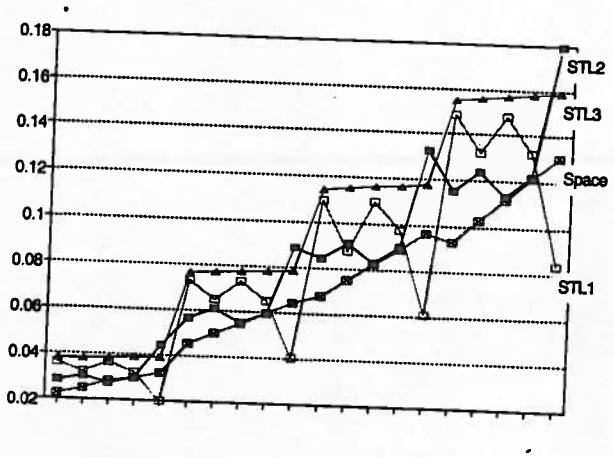

FIGURE 4. Timing data of the general case.

REMARK 2. It is well-known that the Euclidean algorithm iterates $O(\log_2 n)$ times to compute the GCD of n and Δ [4]. Each iteration uses one division. On the other hand, in $\mathbf{STL}_3$, the computation of GCD is incorporated into the algorithm with the help of variable `Bound`, which is maintained n times. Thus, Figure 4 shows that the cost of maintaining `Bound` n times is larger than the cost of performing $O(\log_2 n)$ divisions. Note that it is not clear whether $\mathbf{STL}_3$ is the best way of incorporating GCD into $\mathbf{STL}_2$.

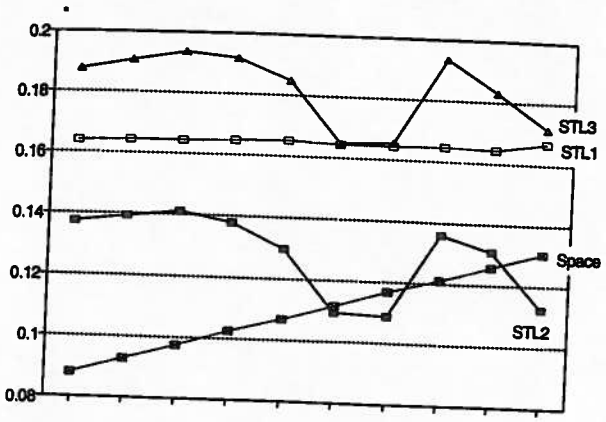

FIGURE 5. Timing data of the relatively prime case.

5.2. The relatively prime case

In this section, we shall study the effect of n and Δ being relatively prime. We shall use $n = 20{,}000$ and $\Delta = 997, 1997, 2997, \ldots, 9997$. Thus, n and Δ are relatively prime.

Figure 5 contains the timing data of the relatively prime case. Since **Space** always uses $n + \Delta$ movements and $\mathbf{STL}_2$ only needs $n + 1$, although **Space** is faster than $\mathbf{STL}_2$ when Δ is small, it becomes slower when Δ is close to $n/2$ and is much slower when $\Delta \geq n/2$. $\mathbf{STL}_1$ is always slower than both **Space** and $\mathbf{STL}_2$. Note that there seems a constant gap between $\mathbf{STL}_2$ and $\mathbf{STL}_3$, which may be interpreted as the extra cost for computing the GCD implicitly. This gap can be characterized with a simple regression model:

Dependent var.	Constant	Offset/1000	R^2
$\mathbf{STL}_3$–$\mathbf{STL}_2$	0.051569	0.000706 (0.0002)*	0.59

*Standard error of coefficient.

Thus, the average of these gaps is 0.051569 and, as offset increases, this gap increases very slowly at a rate of 0.000706/1000. Note that this rate is significant even though its value is very small. As a result, the extra cost of implicitly computing the GCD is significantly higher than computing it explicitly as a preprocessing step.

6. CONCLUSION

In this paper, we have presented a complexity analysis of two STL rotation algorithms and proved that the one using random access iterators is optimal. Theoretically, this optimal algorithm is the fastest. Practically, it also compares favourably against other algorithms, including the one requiring an auxiliary array. Performance data indicate that this could be the fastest in-place rotation algorithm for larger size.

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant CCR-9696084 (formerly CCR-9410707) and grant DUE-9653244. The author thanks the anonymous referees for suggesting the ideas in Remark 1 and Remark 2.

REFERENCES

[1] Musser, D. R. and Saini, A. (1996) *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA.

[2] Stepanov, A. A. and Lee, M. (1995) *The Standard Template Library*. Technical Report HPL-94-34, April 1994, revised July 7, 1995.

[3] Artin, M. (1991) *Algebra*. Prentice-Hall, Upper Saddle River, NJ.

[4] Bach, E. and Shallit, J. (1996) *Algorithmic Number Theory: Volume 1, Efficient Algorithms*. MIT Press, Massachusetts, MA.