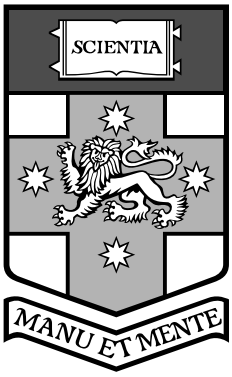
Rotated Library Sort

Franky Lam and Raymond K. Wong
National ICT Australia
and School of Computer Science & Engineering
University of New South Wales
NSW 2052, Australia
Email: franky.lam@nicta.com.au, wong@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0506
March 2005

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES



Abstract

This paper investigates how to improve the worst case runtime of INSERTION SORT while keeping it in-place, incremental and adaptive. To sort an array of n elements with w bits for each element, classic INSERTION SORT runs in $O(n^2)$ operations with wn bits space. GAPPED INSERTION SORT has a runtime of $O(n \lg n)$ with a high probability of only using $(1 + \epsilon)wn$ bits space. This paper shows that ROTATED INSERTION SORT guarantees $O(\sqrt{n} \lg n)$ operations per insertion and has a worst case sorting time of $O(n^{1.5} \lg n)$ operations by using optimal $O(w)$ auxiliary bits. By using extra $\Theta(\sqrt{n} \lg n)$ bits and recursively applying the same structure l times, it can be done with $O(2^l n^{1+\frac{1}{l}})$ operations. Apart from the space usage and time guarantees, it also has the advantage of efficiently retrieving the i-th element in constant time. This paper presents ROTATED LIBRARY SORT that combines the advantages of the above two improved approaches.

1 Introduction

In this paper, given the universe $U = \{1, \ldots, u\}$, we use the transdichotomous machine model [7]. The word size w of this machine model is $w = O(\lg u)$ bits and each word operation of this model takes $O(1)$ time (this paper defines $\lg$ as $\log_2$). This paper assumes all the n elements stored in the array A are within the universe U, meaning each element takes exactly w bits and the array A takes wn bits in total. The traditional INSERTION SORT algorithm belongs to the family of *exchange sorting algorithms* [8] which is based on element comparisons. It is similar to how human sort data and its advantage over other exchange sorting algorithms is that it can be done *incrementally*. The total order of all elements are maintained at all times, traversal and query operations can be performed on A as INSERTION SORT never violates the invariants of a sorted array. It is also *adaptive*, as its runtime is proportional to the order of the insertion sequence. During insertion of a new element x to an existing sorted array A, INSERTION SORT finds the location of x for insertion and create a single gap by right-shifting all the elements larger than x by one position. Obviously, its worst case $\lg n! + n$ comparisons combined with its worst case $\Omega(n^2)$ element moves, with a total of $O(n^2)$ operations, makes it impractical except for sorting with a small n or when the insertion sequence is mostly sorted. This paper investigates how to improve INSERTION SORT while keeping its nice incremental and adaptive properties.

Incremental Sorting Problem

First we define the incremental sorting problem as maintaining a sequence S (not necessarily an array A) of n elements in universe U subject to the following functions:

- `insert`(x, S): insert x into S.
- `member`(x, S): return whether element $x \in S$.
- `select`(j, S): return the j-th element x where `select`$(1, S)$ is the smallest element in S and `select`$(j, S) <$ `select`(k, S) if $j < k$. `select`(j, S) is not necessarily equal to $S[j-1]$.
- `predecessor`(j, S): special case of `select`$(j-1, S)$, but `select`(j, S) is already known.
- `successor`(j, S): special case of `select`$(j+1, S)$, but `select`(j, S) is already known.

This model defines incremental sorting as a series of `insert`(x, S) from the input sequence $X = \langle x_1, \ldots, x_n \rangle$, such that we can query the array S using `select` and `member` between insertions; or we can traverse S using `predecessor` and `successor` between insertions. It seems that the traversal functions are redundant, but in fact they are only redundant when `select` can be done in $O(1)$ operations, which Corollary 1 shows that we have to relax this requirement. For most cases, when `select` cannot be done in constant time, `predecessor` and `successor` can still be done in $O(1)$ operations. It is possible that some incremental sorting algorithms can be done in-place if they reuse the same space of the input sequence X.

Adaptive Sorting Problem

The adaptive sorting problem is defined as any sorting algorithm with its runtime proportional to the disorder of the input sequence X. Estivill-Castro et al [5] define an operation `inv`(X) to measure the disorder of X, where `inv`(X) denotes the exact number of *inversions* in X. (i, j) is an inversion if $i < j$ and $x_i > x_j$. The number of inversions is at most $\binom{n}{2}$ for any sequence, therefore any exchange sorting algorithm must terminate after $O(n^2)$ element swaps. Clearly INSERTION SORT belongs to the adaptive sorting family as it performs exactly `inv`$(X) + n - 1$ comparisons and `inv`$(X) + 2n - 1$ data moves.

Corollary 1 *Any comparison based, in-place, incremental and adaptive sorting algorithm that uses only $O(w)$ temporary space and achieves $O(1)$ operations for* `select` *requires at least $O(\texttt{inv}(X))$ swaps.*

It is trivial that with the above scenario, INSERTION SORT is the only optimal sorting algorithm as there are no other possible alternative approaches that can satisfy all the above constraints, thus we have to relax some of the requirements — this paper assumes `select` does not need to run in $O(1)$ time, meaning partial order is tolerable until all elements in X are inserted. It is essential that `select` should still run reasonably fast, otherwise it has lost the purpose of being incremental.

1.1 Variants of Insertion Sort

1.1.1 Fun Sort

Biedl et al [3] showed an in-place variant of INSERTION SORT called FUN SORT that achieves worst case $\Theta(n^2 \lg n)$ operations. They achieve the bound by applying binary search to an unsorted array to find an inversion and reduce the total number of inversions by swapping them. By picking two random elements ($A[i]$ and $A[j]$, $i < j$) and swapping them if it is an inversion, the total number of inversions is reduced by at least one; because for $i < k < j$, either (i, k) or (k, j) is an inversion, or both. As stated before, any algorithm will terminate after $O(n^2)$ element swaps. By observation, its performance is rather poor in the worst case, as $\binom{n}{2}$ swaps are required, but its average case runtime seems rather fast. Strictly speaking, FUN SORT does not belong to a variant of INSERTION SORT as it is not strictly incremental, but it is an interesting adaptive approach.

Library Sort

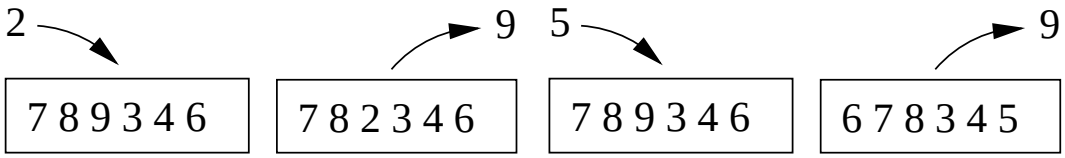
Bender et al [2] show that by having a ϵwn bits space overhead to leave room for gaps, and keeping gaps evenly distributed by redistributing the gaps when the 2^i-th element is inserted, GAPPED INSERTION SORT, or LIBRARY SORT for short, has a high probability of achieving $O(n \lg n)$ operations. As most sorting algorithms can be done in-place, we can make a fair assumption that the sorted result must use the same memory location. The auxiliary space cost of LIBRARY SORT is $(1 + \epsilon)wn$ bits as it needs to create a temporary continuous array A'. Alternatively, their approach can be improved by tagging a temporary auxiliary ϵwn space to A, thus creating a virtual $(1 + \epsilon)wn$ size array A', making the algorithm less elegant but not affecting the time bound or space bound.

Unfortunately, ϵ needs to be chosen beforehand, and large ϵ does not guarantee $O(n \lg n)$ operations as they have made a big assumption that A is a randomly permuted sequence within U. With adversary insertion, such as reverse sorted order (which happens fairly often in real life scenarios), this algorithm degrades to amortized $\Omega(\sqrt{n})$ operations per insertion, regardless of the ϵ, making the worst case $O(n^{1.5})$ operations, although it might be possible to improve the runtime cost to worst case amortized $O(\lg^2 n)$ per insertion [1]. Even within their assumption, the time bound is amortized, regardless of the disorder of the input sequence, as their algorithm needs to rebalance the gaps on the 2^i-th insertion.

Finally, Bender et al did not address that their approach takes $O(n)$ operations to perform `select`(j, A) that finds the j-th element in an array A, because the j-th element does not locate at $A[j - 1]$ but locate at somewhere between $A[j - 1]$ to $A[j - 1 + \epsilon j/n]$. A linear scan is required to determine the rank of the j-th element. It is possible to improve `select` by using more space to maintain the locations of gaps, however that improvement is not the focus of this paper.

Rotated Sort

ROTATED INSERTION SORT, or just ROTATED SORT for short, is based on the idea of the *implicit data structure* called *rotated list* [9]. Implicit data structures is where the relative ordering of the elements



(a) before easy exchange (b) after easy exchange (c) before hard exchange (d) after hard exchange

Figure 1: An example of $O(1)$ easy change and $O(n)$ hard exchange on a rotated list

are stored implicitly in the pattern of the data structure, rather than explicitly storing the relative ordering using offsets or pointers. Rotated List achieves $O(n^{1.5} \lg n)$ operations using constant $O(w)$ bits temporary space, or $O(n^{1.5})$ operations with extra $\Theta(\sqrt{n} \lg n)$ bits temporary space, regardless of w. It is adaptive as its runtime depends on $\texttt{inv}(X)$ and it is incremental as `select` can be done in constant time.

This paper is organized as follows. First we present the idea of the algorithm in Section 2 that achieves the $O(n^{1.5} \lg n)$ operations, then we show some time and space complexity and their tradeoffs in Section 3. Section 5 shows how to achieve $O(2^l n^{1+\frac{1}{l}})$ operations by applying the idea recursively and Section 6 combines the idea of both LIBRARY SORT and ROTATED SORT. Section 7 concludes this paper.

2 Rotated Sort

In essence, the rotated sort is done by controlling the number of element shifts from $O(n)$ shifts per insertion to a smaller term, such as $O(\sqrt{n})$ shifts or even $O(\lg n)$ shifts, by virtually dividing A into an alternating singleton elements and rotated lists that satisfies a partial order. By having an increasing function that controls the size of all the rotated lists, we only need to push the smallest elements and pop the largest element between a small sequence of rotated lists per insertion.

2.1 Rotated List

A rotated list or sorted *circular array*, is an array $L = [0, \ldots, n-1]$ with a largest element $L[m] > L[i]$, $0 \leq i < n$ and $L[i \pmod n] < L[i+1 \pmod n]$, $i \neq m$. We need $\lceil \lg n \rceil$ comparisons to find the position of the minimum and maximum element in the array, or constant time if we have maintained a $\lceil \lg n \rceil$ bits pointer to store m explicitly for L.

This paper uses the same terminologies from Frederickson [6], where the rotated list L has two functions — `easyExchange`, where the new smallest element $x < L[i]$, $0 \leq i < n$ replaces the largest element $L[m]$ and returns $L[m]$; `hardExchange` is identical to easy exchange, but x can be any number. This paper defines an extra function `normalize` that transform the rotated list to a sorted array.

As described in [6], easy exchange can be done in $O(1)$ operations once $L[m]$ is found, as the operation only needs to replace $L[m]$ with the new smallest element x. Array L still satisfies as a rotated list, but the position m' of the new largest element $L[m']$ is left-circular-shifted by one ($m' = m - 1$, or $m' = n - 1$ if $m = 0$). Hard exchange is $O(n)$ since it needs to shift all the elements larger than x in the worst case. Figure 1 shows an example of both easy exchange and hard exchange on a rotated list.

Normalization can be done in $O(n)$ time, an obvious way is by having a temporary duplicate but the exact bound can also be achieved in-place recursively by using Algorithm 1, which has exactly optimal

$2n$ words read and $2n$ words write for the array L. The same algorithm can also be done iteratively.

Algorithm 1 Transform a rotated list L to a sorted list L' with $2n$ words read and $2n$ words write. $L[m]$ is the largest element and $n = |L|$.

```
normalize(m, L)
1: if m < n/2 − 1 then
2:     swap(L[0, ..., m], L[m + 1, ..., 2m + 1])
3:     normalize(2m + 1, L[m + 1, ..., n − 1])
4: elif m > n/2 − 1 then
5:     swap(L[0, ..., n − m − 2], L[m + 1, ..., n − 1])
6:     normalize(m, L[n − m − 1, ..., m])
7: else
8:     swap(L[0, ..., m], L[m + 1, ..., n − 1])
```

2.2 Implicit Dynamic Dictionary

The *dynamic dictionary* problem is defined as given a set $D \subseteq U$, $|D| = n$ and supports $\texttt{member}(x, D)$ to determine whether $x \in D$ and $\texttt{insert}(x, D)$ that insert x into D. It is a subset of the incremental sorting problem. Given a monotonic increasing integer function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$, dynamic dictionary can be implemented implicitly by using an array A, and visualize it as a 2-level rotated lists. We divide A into a list of r pairs $D = \langle P_0, \ldots, P_r \rangle$, each pair P_i consists of a singleton element e_i and a sub-array L_i of size $f(i)$ that is used as a rotated list. For an array of size n, we have $n \leq \sum_{i=1}^{r}(f(i) + 1)$. The purpose of having a monotonic increasing integer function is that the number of blocks will always be proportional to the array size, regardless of the number of insertions. This also avoids amortized runtime cost as it requires no redividing when the array grows. This invariant needs to be guaranteed in order to guarantee the runtime as it controls the number of soft exchanges performed per insertion.

3 Analysis

Lemma 2 *The total number of rotated lists, or the total number of singleton elements in the implicit dictionary structure of size n is at most $\lceil\sqrt{2n}\rceil$, regardless of the increasing function f.*

Proof: To make it simpler, we can increase n to $n' = \sum_{i=1}^{r}(f(i) + 1) \geq 2r$, and if we use the slowest increasing function on $\mathbb{Z}^+$ where $f(i) = i$, then:

$$\begin{aligned} n' &= \sum_{i=1}^{r}(i+1) \\ 2n' &= r^2 + 3r \\ 2n' + \frac{9}{4} &= (r + \frac{3}{2})^2 \\ r &= \sqrt{2n' + \frac{9}{4}} - \frac{3}{2} \\ r &\leq \sqrt{2n'} \end{aligned}$$

□

We can now analyze the total runtime of functions by maintaining the offset m for the largest element $L_k[m]$ on all rotated lists L_k.

Lemma 3 *The total space cost of maintaining $M = \langle m_1, \ldots, m_r \rangle$, where m_k is the position of the largest element for the rotated lists L_k, is $\sum_{i=1}^{r} \lceil \lg f(i) \rceil$ bits, or it can be done in $\Theta(\sqrt{n} \lg n)$ bits.*

Proof: Using $f(i) = i$ and Lemma 2, we have the list of rotated lists $\langle L_1, \ldots, L_r \rangle$ of size $\langle 1, \ldots, \sqrt{2n} \rangle$. The sum of the bits required is $\sum_{i=1}^{\sqrt{2n}} \lg i = \lg(\sqrt{2n}!)$. By Stirling's approximation, it is reduced to approximately $\sqrt{2n} \lg \sqrt{2n} - \sqrt{2n} + 1 = \Theta(\sqrt{n} \lg n)$.

□

Lemma 4 `select` *takes $O(1)$ operations using extra $\Theta(r \lg n)$ bits space, or it can be done in extra $\Theta(\sqrt{n} \lg n)$ bits space. On optimal space,* `select` *takes $O(\lg f(r))$ operations.*

Proof: To calculate $\texttt{select}(j, S)$, we need to find which rotated list L_k that it is located, meaning we need to find the smallest k such that $\sum_{i=1}^{k+1}(f(i) + 1) > j$. When L_k is found, we can get m_k in $O(1)$ operations with $\Theta(\sqrt{n} \lg n)$ bits space using Lemma 3. From Lemma 2, we need at most $r \leq \sqrt{2n}$ rotated lists and storing the beginning offset of any rotated list takes at most $\lceil \lg n \rceil$ bits. Therefore, we can hold the whole offset table in $\Theta(r \lg n) = \Theta(\sqrt{n} \lg n)$ bits. If there exists a function $g(x) = \int_1^k f(x)$, then $O(1)$ operation can be done without the offset table. For example, using $f(i) = i$, $g(x) = \frac{x^2+x}{2}$, L_k can be found by doing $k = \sqrt{2j + 9/4} - 3/2$.

Without maintaining m, it takes $O(1)$ time to find L_k, along with an extra $\lceil \lg f(k) \rceil$ comparisons to find m_k, in worst case where $k = r$, the time complexity becomes $O(\lg f(r))$.

□

Lemma 5 `member` *can be done in $O(\lg r + \lg f(r))$ operations. On optimal space, we need no more than $\frac{3}{2}\lceil \lg n \rceil + O(1)$ comparisons, or no more than $\lceil \lg n \rceil + O(1)$ comparisons using $\Theta(\sqrt{n} \lg n)$ bits.*

Proof: We perform $\texttt{member}(x, D)$ by doing a binary search on all the r singleton elements $\langle e_0, \ldots, e_r \rangle$ to determine which rotated list L_k does x belong to (or it returns the position of singleton element $e_k = x$ itself if we are lucky), then followed by a binary search on the rotated list $L_k[0, \ldots, f(k) - 1]$ to find the largest element m_k and finally perform another binary search on L_k to find x. The total number of comparisons is $\lceil \lg r \rceil + 2\lceil \lg f(k) \rceil$. In the worst case where $k = r$, the search is within the last (and largest) rotated list L_r. Let $f(i) = i$ then the worst case cost is $\lceil \lg r \rceil + 2\lceil \lg f(r) \rceil = 3\lceil \lg \sqrt{2n} \rceil = \frac{3}{2}\lceil \lg n \rceil + O(1)$. Using Lemma 3, we eliminate one binary search and the cost is reduced to $\lceil \lg r \rceil + \lceil \lg f(r) \rceil = \lceil \lg n \rceil + O(1)$.

□

The probability of finding a singleton element is $P(x \in \langle e_0, \ldots, e_r \rangle) = \frac{r}{n}$ and the probability of finding an element in the rotated list L_i is $P(x \in L_i) = \frac{f(i)}{n}$. If we assume each element in the implicit dictionary D are equally likely to get selected in the search, with maintaining M, the average number of comparisons of the search is $\frac{r}{n}\lceil \lg r \rceil + \sum_{i=1}^{r} \left(\frac{f(i)}{n} (\lceil \lg r \rceil + \lceil \lg f(i) \rceil) \right)$, which is equal to:

$$\lceil \lg r \rceil + \frac{\lg H(f(r))}{n}$$

where $H(n)$ represents the hyperfactorial of n.

Lemma 6 *With optimal space,* `insert` *is $O(\lg r + \sum_{i=1}^{r-1} \lg f(i) + f(r))$ operations, or it can be done in no more than $O(\sqrt{n} \lg n)$ operations. Using $\Theta(\sqrt{n} \lg n)$ bits, the time complexity is $O(r + f(r))$, or it can be done in $O(\sqrt{n})$ operations.*

Proof: To perform insertion, first we need to locate the rotated list L_k for insertion by performing a $\lceil \lg r \rceil$ search on the singleton elements, then a hard exchange is performed on L_k, which is followed by a sequence of soft exchanges will be done from L_{k+1} to L_{r-1} and terminated with either a hard exchange or append to L_r. The total cost is $\lceil \lg r \rceil + f(k) + \sum_{i=k+1}^{r-1} \lceil \lg f(i) \rceil + f(r) + O(1)$, or just $O(\lg r + \sum_{i=1}^{r-1} \lg f(i) + f(r))$ as $f(k) \ll f(r)$. In worst case, where $k = 1$, using $f(i) = i$, the cost is $O(\lg \sqrt{2n} + (\sqrt{2n} \lg \sqrt{2n} - \sqrt{2n} + 1 - (2 \lg 2 - \lg 2 + 1)) + \sqrt{2n}) = O(\sqrt{n} \lg n)$ operations.

From Section 2.1, with space specified in Lemma 3, soft exchange takes $O(1)$ time. In worst case, where $k = 1$, we need to perform soft exchange on all rotated lists, except L_k and L_{r-1}, thus $r - 2$ rotated lists in total. Therefore, the total time complexity of $\lceil \lg r \rceil$ binary search, initial hard exchange on L_1, sequence of $r - 2$ soft exchanges and the final hard exchange on L_{r-1} is $\lceil \lg r \rceil + f(k) + r + f(r) + O(1) = O(r + f(r))$. Using $f(i) = i$, we have $O(\sqrt{2n} + \sqrt{2n}) = O(\sqrt{n})$ operations. □

Theorem 7 ROTATED SORT *can be done in worst case $O(n^{1.5} \lg n)$ operations with only $O(w)$ bits space, or in worst case $O(n^{1.5})$ operations with $\Theta(\sqrt{n} \lg n)$ bits space.*

Proof: First, visualize an array A as a concatenation of an implicit dictionary D with size 0 with the input sequence X with n remaining elements. We increase D by inserting $A[i]$ into D at every step i. Using $f(i) = i$, from Lemma 6 where each insertion takes worst case $O(\sqrt{i} \lg i)$, the total can be done in $\sum_{i=1}^{n} (\sqrt{i} \lg i) \approx \int_1^n (\sqrt{i} \lg i) = O(n^{1.5} \lg n)$. With $\Theta(\sqrt{n} \lg n)$ bits space, from Lemma 6, it takes $\sum_{i=1}^{n} \sqrt{i} \approx \int_1^n \sqrt{i} = O(n^{1.5})$ operations.

□

Theorem 8 `select`(j, A) *can be done adaptively in constant time with extra $\Theta(\sqrt{n} \lg n)$ bits space and it can be done in $O(1)$ operations after n insertions for* ROTATED SORT *without using extra space.*

Proof: Using the $O(1)$ time function $g(k) = \frac{k^2+3k}{2}$ for $f(i) = i$, from the proof at Lemma 4, L_k can be found in $O(1)$, `select`(j, A) can be implemented simply using $A[(m_k + j - g(k)) \pmod{f(k)} + g(k)]$. Once after n insertions, we only need to perform `normalize`, which the runtime $O(n)$ takes the lower term of the sort. Now we simplify the function `select`$(j, A) = A[j - 1]$. The above proves directly to the following:

□

Corollary 9 `predecessor` *and* `successor` *can be done in $O(1)$ operations adaptively with $O(\sqrt{n} \lg n)$ space if $g(i)$ exists and it can be done in $O(1)$ operations after n insertions for* ROTATED SORT *without using extra space.*

Proof: Trivial. They are both special cases of `select`. Alternatively, `successor` can be performed even faster by checking m_i and m_{i+1}, where L_i is the rotated list that `select`(j, S) belongs to.

□

4 Choosing the Increasing Function

The increasing function f affects the time complexity for the insertion and thus the sorting time. We have shown in Theorem 7 that using the slowest increasing integer function, ROTATED SORT takes worst case $O(n^{1.5})$ operations.

Note that the dominant time belongs to performing easy exchange on $O(\sqrt{n})$ rotated lists for every insertion. One idea to improve it is to reduce r from $O(\sqrt{n})$ to $O(\lg n)$ by using an exponential growing function. However, the larger the ratio of $f(i + 1)/f(i)$, the more expensive it is to perform a hard exchange on the rotated list. In the case where $f(i) = 2^i$, hard exchange takes worst case $n/2$ right-shifts on the last rotated list L_{r-1}. We need to minimize the insertion cost $O(\lg r + f(r))$ from Lemma 6 by choosing the appropriate increasing function.

Theorem 10 *The function $f(i) = i$ is optimal, up to a constant factor, to control the increasing size for the 2-levels rotated lists in* ROTATED SORT.

Proof: If we make r as the x-axis and $f(r)$ as the y-axis and we limit the maximum range of both axes to n, and from Lemma 2, we know the area $\int_1^r (f(x) + 1) = n$. Even if we assume n does not grow, thus we allow the change of rate of f to be 1, the optimal function is where $r = f(r) = \sqrt{n}$, as the problem is equivalent to minimizing the circumference of a fixed rectangular area. With those values, insertion takes $O(\lg r + f(r)) = O(\sqrt{n})$ operations. Therefore, from Lemma 6, the slowest increasing integer function $f(i) = i$ is already close to the optimal up to a constant factor.

□

5 Multi-Level Rotated List

To reduce the number of hard and soft exchanges, we can apply the idea of rotated list divisions recursively on each rotated list itself. Each sub-array L within A are further divided up recursively for l number of times; we can see that even for the fast growing function $f(i) = 2^i$, an array of size n will consist of at most $l = \lceil \lg n \rceil$ rotated lists with exponential growing size and the maximum number of levels l is at most $\lg n$.

Lemma 11 $\texttt{insert}(x, S)$ *can be done in $O(2^l n^{\frac{1}{l}})$ operations by using an l-levels rotated list, showed by Raman et al [10].*

With Lemma 11, ROTATED SORT can be done in $\sum_{i=1}^{n} (2^l i^{1+\frac{1}{l}})$ operations; we know that to minimize the sorting cost, l should be chosen to minimize $2^l n^{\frac{1}{l}}$. We can always choose the perfect l but make the cost amortized, by performing normalization that takes $O(n)$ operations whenever the array grows until l is not optimal. A perfectly sorted array can be visualized as an l-levels rotated list, regardless of l. We can maintain the optimal value of l by normalization, with the amortized constant cost. Therefore, the overall sorting cost can remain the same.

Corollary 12 *The optimal number of levels on the multi-levels rotated list is $l = \sqrt{\lg n}$. As $2^l = n^{\frac{1}{l}} \implies l = \lg n^{\frac{1}{l}} \implies l = \sqrt{\lg n}$.*

Theorem 13 ROTATED SORT *can be done in $O(2^{\sqrt{\lg n}} n^{1+\frac{1}{\sqrt{\lg n}}})$ operations.*

Proof: From Lemma 11 and Corollary 12, we know that the above time bound can be achieved, amortized, by doing normalzation on every 2^{2i}-th insertion. The same bound can be done deamortized easily, simply by having $(i + 1)$-level rotated list for rotated lists $\langle L_{2^{2(i-1)}}, \ldots, L_{2^{2i}} \rangle$.

□

The runtime on Theorem 13 is smaller than $O(n^{1.5})$ but larger than $O(n \lg n)$ and they are all growing in a decreasing rate with respect to n.

The advantage of INSERTION SORT is that not only it is incremental, but also adaptive, where traditional INSERTION SORT performs exactly $\texttt{inv}(X) + 2n - 1$ data moves [5]. Knuth [8] and Cook et al [4] showed that INSERTION SORT is best for nearly sorted sequences. The same adaptive property can also apply for ROTATED SORT.

Lemma 14 ROTATED SORT *can be done in best case $O(n)$ operations.*

Proof: During `insert`, changing the worst case cost by only a constant, we perform the binary search by searching from the last singleton element e_r instead of $e_{r/2}$. This only increases the number of comparison by 1 but reduces the $\lceil \lg r \rceil$ comparisons of singleton elements to only 1. In the best situation, no hard exchange or soft exchange is performed, making the time complexity $O(n)$.

□

Theorem 15 ROTATED SORT *can be adaptive according to the inversion of* X.

Proof: Instead of optimizing for just the best case, we want to generalize it for any nearly sorted sequence S, where the total cost is proportional to $\texttt{inv}(X)$. We need to perform a sequence of exponential searches of x from the tail of $\langle e_{r-1}, e_{r-2}, \ldots e_{r-2^k} \rangle$ until $e_{r-2^k} < x$ and $e_{r-2^{k+1}} > x$, then we begin a binary search of x between e_{r-2^k} and $e_{r-2^{k+1}}$.

□

6 The Best of Both Worlds — Rotated Library Sort

Instead of using multi-level rotated list, an alternative way to minimize the total number of soft exchanges and hard exchanges is to combine the concept of gaps from LIBRARY SORT with ROTATED SORT. For every rotated list L_i, we maintain an extra array K_i with the size $\epsilon f(i)$. We now treat $J_i = \langle K_i, L_i \rangle$ as one single array that acts as a rotated list. We maintain the total number of gaps (its total value) and the position offset of the largest element m_i for J_i instead of L_i. In this setting, the gaps of J_i are always located between the smallest element and the largest element.

During insertion, if the initial rotated list J_k contains gaps, only the initial hard exchange on J_k is performed. No soft exchanges nor hard exchange on the final rotated list J_{r-1} is required. If J_k is full before the insertion, we still need to perform soft exchanges from the rotated list J_{k+1} up to the rotated list J_{r-2}. However, these soft exchanges will stop at the first rotated list that contains at least one gap. This can also be seen as an improved version of LIBRARY SORT — by clustering the gaps into r blocks in order to find them quickly without right-shifting all the elements in between gaps.

Lemma 16 *For a given input sequence of size* n*, the cost of all rebalances is* $O(n)$ *and the amortized rebalance cost is* $O(1)$ *per insertion [2].*

Similar to the LIBRARY SORT, after the 2^i-th element insertion, the array A need to be rebalanced with the cost specified in Lemma 16; but we can save the cost of normalization. i.e., If we do apply the rotated list divisions recursively, from Theorem 13, the optimal level (i.e., l) grows after the 2^{2i}-th element insertion. As a result, rebalancing that includes the effect of normalization will automatically adjust the optimal recursion level. Since array rebalancing is needed regularly during element insertions and rebalancing does have the normalization effect, the frequency of normalization is less than the frequency of rebalancing. Note that it is possible to improve the cost of all rebalances from $O(n)$ to $O(\epsilon r)$. However, this improvement will not affect the $O(1)$ amortized rebalance cost and it will not include the effect of normalization, we will omit its discussions here.

Lemma 17 *It is possible to query the sum of all previous gaps before the rotated list* L_k *in constant worst case time and updates in* $O(r^\epsilon)$ *worst case time, with only extra* $O(r \lg(\epsilon f(r)))$ *bits space.*

Proof: For simplicity, we do not consider gaps after the last element. Each rotated list L_i has at most $\epsilon f(i)$ gaps, the largest rotated list L_r has at most $\epsilon f(r)$ gaps. The problem is identical to the problem of *partial sum* with r elements with the universe $\{1, \ldots, f(r)\}$ that Raman et al [10] solved in the above bounds.

□

Lemma 18 `select` *can be done in* $O(1)$ *with extra* $O(\sqrt{n}\lg n)$ *space in* ROTATED LIBRARY SORT.

Proof: Trivial. We perform `select`(j, S) similar to ROTATED SORT, but we need to add the sum of all previous gaps to j using Lemma 17, which also takes $O(1)$ time.

□

It is possible to avoid rebalancing on the 2^i-th element insertion. Instead of performing a sequence of soft exchanges with each soft exchange inserting a single smallest element and returning a single largest element, we can perform the soft exchange with $\epsilon f(k)$ elements. When J_k is full after a hard exchange, we pop the largest $\delta = \epsilon f(k)$ elements after the hard exchange and perform the soft exchange of δ elements on the rotated lists $\langle J_{k+1}, \ldots, J_{r-2}\rangle$. δ will get smaller and eventually becomes zero. If we assume the elements of the input sequence are randomly distributed, δ will decrease in an increasing rate as the size of the extra arrays K increases monotonically according to the function f. Soft exchange will then cost $O(\delta)$ operations, while the worst case cost for hard exchange remains unchanged.

Theorem 19 `insert` *in* ROTATED LIBRARY SORT *can be done in amortized* $O(\lg r + f(r))$ *operations.*

Proof: For `insert` in ROTATED LIBRARY SORT, hard exchanges on J_k are unavoidable initially. However, the larger the ϵ is, fewer hard exchanges on J_{r-1} will be required at the end. Therefore, the worst case sceanrio happens when insertion occurs at J_k where $k = r/2$. Each insertion consists of a binary search with $O(\lg r)$ operations. The first $(\epsilon f(k) - 1)$ insertions include a hard exchange, that costs worst case $O(f(k))$ operations, because of the empty gaps. The $(\epsilon f(k))$-th insertion will incur the initial hard exchange plus a soft exchange on J_{k+1} that costs $O(\epsilon f(k))$ operations. It terminates because the number of gaps in $J_{k+1} >$ that in J_k. Since J_k contains $\epsilon f(k)$ gaps, the $(\epsilon f(k) + 1)$-th insertion till the $(2\epsilon f(k) - 1)$-th insertion require only $O(\lg r + f(k))$ operations. Then the $(2\epsilon f(k))$-th insertion needs to perform more soft exchanges. The difference between the numbers of soft exchanges of the $(i\epsilon f(k))$-th and $((i+1)\epsilon f(k))$-th insertions will increase by at most one (i.e., the difference will be either zero or one). The difference decreases until the number of soft exchanges hit its bound $r - k$. When the bound of $r - k$ soft exchanges is reached, we need the final hard exchange with worst case cost $O(f(r))$ operations. We can clearly see the pattern here, i.e., every $\epsilon f(k)$ insertions require $O(\lg r + f(k))$ operations, then followed by a single insertion that requires $O((r - k) + f(r))$ operations. From this observation, we can approximate that the amortized cost is $O(\lg r + f(k) + (r - k + f(r))/\epsilon)$. So with a large enough ϵ, the insertion cost in the worst case sceanrio is close to amortized $O(\lg r + f(k)) \leq O(\lg r + f(r))$ operations, instead of $O(r + f(r))$ from Lemma 6, which is clearly an improvement. □

7 Conclusions

This paper shows an alternative approach called ROTATED INSERTION SORT to solve the high time complexity of INSERTION SORT. The approach is incremental yet adaptive, it uses less space than GAPPED INSERTION SORT [2] and does not rely on the distribution of the input. It shows that the ROTATED INSERTION SORT can be done in $O(n^{1.5}\lg n)$ time with $O(w)$ temporary space, which is tightly space bound; or it can be run in $O(2^l n^{1+\frac{1}{l}})$ operations, using only a lower order $\Theta(\sqrt{n}\lg n)$ bits space. This paper further shows a possible combined approach called ROTATED LIBRARY SORT.

There are several problems remain open — first, which function is the best function for the ROTATED LIBRARY SORT to virtually divide the array? Are there any other in-place, incremental and adaptive approaches that outperform ROTATED LIBRARY SORT? What are the time bound, space bound and their tradeoffs between the extra space use, `member`, `insert` and `select`?

References

[1] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In Rolf H. Möhring and Rajeev Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

[2] Michael A. Bender, Martin Farach-Colton, and Miguel Mosteiro. Insertion sort is o(n log n). *CoRR*, cs.DS/0407003, 2004.

[3] Therese Biedl, Timothy Chan, Erik D. Demaine, Rudolf Fleischer, Mordecai Golin, James A. King, and J. Ian Munro. Fun-sort–or the chaos of unordered binary search. *Discrete Applied Mathematics*, 144(3):231–236, 2004.

[4] Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Commun. ACM*, 23(11):620–624, 1980.

[5] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992.

[6] Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, 1983.

[7] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[8] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

[9] J. Ian Munro and Hendra Suwanda. Implicit data structures (preliminary draft). In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 108–117. ACM Press, 1979.

[10] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *WADS '01: Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 426–437. Springer-Verlag, 2001.