Rotated Library Sort

Franky Lam\textsuperscript{1} \quad Raymond K. Wong\textsuperscript{1,2}

\textsuperscript{1} School of Computer Science \& Engineering
University of New South Wales
NSW 2052, Australia

\textsuperscript{2} National ICT Australia, Australia

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 \( O(2^{n+\frac{1}{2}}) \) 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 (Fredman \& Willard 1994). 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 \( \lg n \) as \( \log_{2} n \)). This paper assumes all the \( n \) elements, within the universe \( U \), stored in the array \( A \). This means that 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 (Knuth 1998) which is based on element comparisons. It is similar to how humans 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 creates 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 and a total of \( O(n^2) \) operations makes it impractical in most situations, 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.

This paper is organized as follows. Section 2 discusses previous work related to this paper. We then present the rotated sort algorithm in Section 3 that achieves the \( O(n^{1.5} \lg n) \) operations. After that we discuss the time and space complexity as well as their tradeoffs in Section 4 and Section 5. Section 6 shows how to achieve \( O(2^{n+\frac{1}{2}}) \) operations by applying the idea recursively and Section 7 combines the idea of both LIBRARY SORT and ROTATED SORT. Finally, Section 8 concludes the paper.

2 Background

2.1 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:

- \texttt{insert}(\( x, S \)): insert \( x \) into \( S \).
- \texttt{member}(\( x, S \)): return whether element \( x \in S \).
- \texttt{select}(\( j, S \)): return the \( j \)-th smallest element where \( \texttt{select}(1, S) \) is the smallest element in \( S \) and \( \texttt{select}(j, S) < \texttt{select}(k, S) \) if \( j < k \).
- \texttt{predecessor}(\( j, S \)): special case of \( \texttt{select}(j-1, S) \), but \( \texttt{select}(j, S) \) is already known.
- \texttt{successor}(\( j, S \)): special case of \( \texttt{select}(j+1, S) \), but \( \texttt{select}(j, S) \) is already known.

This model defines incremental sorting as a series of \( \texttt{insert}(x, S) \) from the input sequence \( X = \{x_1, \ldots, x_n\} \), such that we can query the array \( S \) using \texttt{select} and \texttt{member} between insertions; or we can traverse \( S \) using \texttt{predecessor} and \texttt{successor} between insertions. The traversal functions might seem to be redundant, but in fact they are only redundant when \( \texttt{select} \) can be done in \( O(1) \) operations, which Corollary 1 shows that we have to relax this requirement. For most cases, when \( \texttt{select} \) cannot be done in constant time, \( \texttt{predecessor} \) and \( \texttt{successor} \) 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 \( X \).

Although there was no strict guidelines, but similar to most other definition of incremental algorithms, we only consider a particular algorithm is an incremental sorting algorithm if the runtime of its query functions after every individual insertion is comparable to the runtime of the same query functions of normal sorting algorithm after \( n \) insertions.
2.2 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 (Estivill-Castro & Wood 1992) define an operation $\text{inv}(X)$ to measure the disorder of $X$, where $\text{inv}(X)$ 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 $\text{inv}(X) + n - 1$ comparisons and $\text{inv}(X) + 2n - 1$ 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 $\text{select}$ requires at least $O(\text{inv}(X))$ 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 $\text{select}$ 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 $\text{select}$ should still run reasonably fast, or it will lose the purpose of being incremental.

2.3 Variants of Insertion Sort

2.3.1 Fun Sort

Biedl et al (Biedl et al. 2004) have shown an in-place variant of INSERTION SORT called FUN SORT that achieves worst case $O(n^2 \log n)$ 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, up to $2(j - i) - 1$ reductions; because for all $k, i < k < j$, if $(i,k)$ is an inversion, either $(i,k)$ or $(k,j)$ is an inversion, or both. As stated before, any algorithm that can maintain $O(n^2)$ element swaps. By observation, its performance is rather poor in the worst case, as $\binom{n}{2}$ swaps are required, but its average 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.

2.3.2 Library Sort

Bender et al (Bender et al. 2004) have shown that by having a $\epsilon \log n$ bits space overhead as gaps, and keeping gaps evenly distributed by redistributing the gaps when the $2^j$-th element is inserted, GAPPED INSERTION SORT, or LIBRARY SORT for short, has a high probability of achieving $O(n \log n)$ 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 $(1 + \epsilon)\log n$ bits as it needs to create a temporary continuous array $A'$. Alternatively, their approach can be improved by tagging a temporary auxiliary $\epsilon \log n$ size array $A'$, making the algorithm less elegant but not affecting the time bound or space bound.

Unfortunately, $\epsilon$ needs to be chosen beforehand, and large $\epsilon$ does not guarantee $O(n \log n)$ operations as they have made an assumption that $A$ is a randomly permuted sequence within $U$. To describe it in another way, the algorithm can randomly permute the input with $O(n)$ time before sorting. By permuting the input, the algorithm becomes insensitive to the input sequence, which by definition, LIBRARY SORT is not an adaptive algorithm that can take advantage of a nearly sorted sequence. Under the incremental sorting model, input comes individually, permuting the future input is impossible. With the incremental sorting model, without the permutation of input and with adversary insertion (such as reverse sorted order that can happen fairly often in real life scenarios), the performance of this algorithm degrades to amortized $\Omega(\sqrt{n})$ operations per insertion, regardless of the $\epsilon$. This makes the worst cost $O(n^{1.5})$ operations, although it might be possible to improve the runtime cost to worst case amortized $O(\log^2 n)$ per insertion (Bender et al. 2002). Although in their assumptions the time bound is amortized per insertion, regardless of the disorder of the input sequence, as their algorithm needs to rebalance the gaps on the $2^j$-th insertion.

Finally, Bender et al did not address that their approach takes worst case $O(j + \epsilon n)$ operations to perform $\text{select}(j, A)$, which finds the $j$-th smallest element in an array $A$. This is because the $j$-th smallest element does not locate at $A_{j-1}$. It locates at somewhere between $A_{j-1}$ to $A_{j-1 + 1/\epsilon n}$ depending on the distribution of the gaps. Without knowing the location of the gaps, a linear scan is required to determine the rank of the $j$-th smallest element between insertions. It is possible to improve $\text{select}$ by using more space to maintain the locations of gaps, using a similar structure like the weight-balanced B-tree by Dietz (Dietz 1989).

2.3.3 Rotated Sort

ROTATED INSERTION SORT, or just ROTATED SORT for short, is based on the idea of the *implicit data structure* called *rotated list* (Munro & Szwarc 1979). Implicit data structure is where the relative ordering of the elements is stored implicitly in the pattern of the data structure, rather than explicitly storing the relative ordering using offsets or pointers. Rotated list achieves $O(n^{1.5} \log n)$ operations using constant $O(w)$ bits temporary space, or $O(n \log n)$ operations with extra $\Theta(\sqrt{n} \log n)$ bits temporary space, regardless of $w$. It is adaptive as its runtime depends on $\text{inv}(X)$. It is incremental as $\text{select}$ can be done in constant time.

3 Rotated Sorted

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 $O(\log n)$ 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.

3.1 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 $L[i + 1 \mod n] < L[i + 1 \mod n]$, $i \neq m$. We need $\log n$ comparisons to find the positions of the minimum and maximum elements in the array, or constant time if we have maintained a $\log n$ bits pointer to store $m$ explicitly for $L$.

This paper uses the same terminologies from Frederickson (Frederickson 1983), where the rotated list $L$ has
two functions — easyExchange, where the 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 (Frederickson 1983), 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 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 easy exchange and hard exchange examples on a rotated list.

Normalization can be done in \(O(n)\) time, an obvious way to achieve this 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: Transformation of a rotated list

```plaintext
Algorithm 1 Transformation of a rotated list \(L\) to a sorted list \(L'\) with \(2n\) words read and \(2n\) words write. \(L[m]\) is the largest element and \(n = |L|\).
```

```plaintext
normalize(m, L)
1. if \(m < \frac{n}{2} - 1\) then
2. swap(L[0],...,m, L[m+1],...,2m+1]
3. normalize(2m+1, L[m+1],...,n−1)
4. elif \(m > \frac{n}{2} - 1\) 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]
```

3.2 Implicit Dynamic Dictionary

The dynamic dictionary problem is defined as follows. Given a set \(D \subseteq U\), \(|D| = n\), we need to implement efficiently member \((x, D)\) to determine whether \(x \in D\) and insert \((x, D)\) that insert \(x\) into \(D\). It is a subset of the incremental sorting problem. Given a monotonic (strictly) increasing integer function \(f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+\), dynamic dictionary can be implemented implicitly by using an array \(A\), and be visualized as a 2-level rotated lists. We divide \(A\) into a list of \(r\) pairs \(D = (P_0, \ldots, P_r)\), 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 re-dividing when the array grows. This invariant needs to be guaranteed in order to have the runtime guarantee as it controls the number of soft exchanges performed per insertion.

### 4 Analysis

**Lemma 1.** 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:

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

We can now analyze the total runtime cost of maintaining the offset \(m\) for the largest element \(L_k[m]\) on all rotated lists \(L_k\).

**Lemma 2.** The total space cost of maintaining \(M = \{m_1, \ldots, m_r\}\), where \(m_k\) is the position of the largest element for the rotated lists \(L_k\), is \(\sum_{i=1}^{r} [\lg f(i)]\) bits, or it can be done in \(\Theta(\sqrt{\lg n})\) bits.

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

**Lemma 3.** 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 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 \(\sum_{i=1}^{k} (f(i) + 1) > j\). When \(L_k\) is found, we can get \(m_k\) in \(O(1)\) operations with \(\Theta(\sqrt{n} \lg n)\) bits space using Lemma 2. From Lemma 1, we need at most \(r \leq \sqrt{2n}\) rotated lists and storing the beginning offset of any rotated list takes at most \([\lg n]\) 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 \(g(x) = f_k(x)\), then \(O(1)\) operation can be done without the offset table. For example, using \(f(i) = i\), \(g(x) = \sqrt{2x}\), \(L_k\) can be found by doing \(k = \sqrt{2j + 9/4 - 3/2}\). Without maintaining \(m_k\), it takes \(O(1)\) time to find \(L_k\), along with an extra \([\lg f(k)]\) comparisons to find \(m_k\), in worst case where \(k = r\), the time complexity becomes \(O(\lg f(r))\).

**Lemma 4.** member can be done in \(O(\lg r + \lg f(r))\) operations. On optimal space, we need no more than \(\frac{1}{2} [\lg n] + O(1)\) comparisons, or no more than \([\lg n] + O(1)\) comparisons using \(\Theta(\sqrt{n} \lg n)\) bits.
Proof. We perform member \((x, D)\) by doing a binary search on all the \(r\) singleton elements \((e_0, \ldots, e_r)\) in \(D\) 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 \([\lg r] + 2[\lg f(k)]\). 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 \([\lg r] + 2[\lg f(i)]\) for \(f = i\), from the proof at Lemma 3, \(L_k\) can be found in \(O(\lg n)\) and \(O(1)\). After \(n\) insertions, we only need to perform normalize, in which the runtime \(O(n)\) takes the lower term of the sort. Now we simplify the function \(\text{select}(j, A) = A[j - 1]\). The above leads to the proof of the following:

Corollary 2. \ \text{predecessor and successor can be done in} \(O(1)\) \text{operations adaptively with} \(O(\sqrt{n}\lg n)\) \text{bits} if \(g(i)\) exists and it can be done in \(O(1)\) operations after \(n\) insertions for \(\text{ROTATED SORT without using extra space.}

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

5 Choosing the Increasing Function

The increasing function \(f\) affects the time complexity of the insertion and thus the sorting time. We have shown in Theorem 1 that using the slowest increasing integer function, \(\text{ROTATED SORT}\) takes worst case \(O(n^{1.5})\) operations.

Note that the dominant time is spent on performing easy exchange on \(O(\sqrt{n})\) rotated lists for every insertion. One idea to improve this is to reduce \(f\) from \(O(\sqrt{n})\) to \(O(\lg n)\) by using an exponential growing function. However, the larger the ratio of \(f + 1 / f(i)\), 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 5 by choosing the appropriate increasing function.

Theorem 3. \ \text{The function} \(f(i) = i\) \text{is optimal, up to a constant factor, to control the increasing size for the 2-levels rotated lists in} \(\text{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 \(n\), then from Lemma 1, we know the area \(\int_n^0 f(x + 1) = n\). Even if we assume \(n\) does not grow (thus we allow the change of range of \(f = 1\) 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 \(O(\sqrt{2n + \sqrt{n}}) = O(\sqrt{n})\) operations. Therefore, from Lemma 5, the slowest increasing integer function \(f(i) = i\) is already close to the optimal up to a constant factor.

6 Multi-Level Rotated List

To reduce the number of hard and easy 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 \(l = [\lg n]\) rotated lists.
with exponential growing size, and the maximum number of levels \( l \) is at most \( \lg n \).

**Lemma 6.** \( \text{insert}(x, S) \) can be done in \( O(2^n \sqrt{n} \log n) \) operations by using an \( l \)-levels rotated list, showed by Raman et al (Raman et al. 2001).

With Lemma 6, \textsc{Rotated Sort} can be done in \( \sum_{i=1}^{n} (2^i + \frac{1}{i}) \) 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 3.** The optimal number of levels on the multi-leaves rotated list is \( l = \sqrt{\log n} \). As \( 2^l = n^\frac{1}{l} \Rightarrow l = \log n^\frac{1}{2} \Rightarrow l = \sqrt{\log n} \).

**Theorem 4.** \textsc{Rotated Sort} can be done in \( O(2^\sqrt{\log n} n^{2/3} \log n) \) operations.

**Proof.** From Lemma 6 and Corollary 3, we know that the above time bound can be achieved, amortized, by doing normalization on every \( 2^l \)-th insertion. The same bound can be de-amortized easily, by having \((i+1)\)-level rotated list for rotated lists \( L_{2^{i+1}}, \ldots, L_{2^n} \).

The runtime on Theorem 4 is smaller than \( O(n^{1.5}) \) but larger than \( O(n \log n) \), and they are all growing in a decreasing rate with respect to \( n \).

The advantage of \textsc{Insertion Sort} is that not only it is incremental, but also adaptive, where traditional \textsc{Insertion Sort} performs exactly \( n \cdot \text{inv}(X) + 2n - 1 \) data moves (Estivill-Castro & Wood 1992). Knuth (Knuth 1998) and Cook et al (Cook & Kim 2000) showed that \textsc{Insertion Sort} is best for nearly sorted sequences. The same adaptive property can also apply for \textsc{Rotated Sort}.

**Lemma 7.** \textsc{Rotated Sort} can be done in best case \( O(n) \) operations.

**Proof.** During \text{insert}, changing the worst case cost by only a constant, we can perform binary search by searching from the last singleton element \( r \), instead of \( r \). This only increases the number of comparison by \( 1 \) but reduces the \( \log r \) comparisons of singleton elements to only \( 1 \). In the situation where hard exchange or soft exchange is performed, making the time complexity \( O(n) \).

**Theorem 5.** \textsc{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 \( \text{inv}(X) \). We need to perform a sequence of exponential searches of \( X \) from the tail of \( (r_{-2}, r_{-2}, \ldots, r_{-2}) \) until \( r_{-2} < x \) and \( r_{-2} < x \), then we begin a binary search of \( x \) between \( r_{-2} \) and \( r_{-2} \).

## 7 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 \( \epsilon(f) \). We now treat \( J_i = \{ K_i, L_i \} \) 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 \( J_i \) contains gaps, only the initial hard exchange on \( J_i \) is performed. No soft exchange nor hard exchange on the final rotated list \( J_{i-1} \) is required. If \( J_i \) is full before the insertion, we still need to perform soft exchanges from the rotated list \( J_{i+1} \) up to the rotated list \( J_{i+2} \). 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 between gaps.

**Lemma 8.** For a given input sequence of size \( n \), the cost of all re-balancing is \( O(n) \) and the amortized re-balancing cost is \( O(1) \) per insertion (Bender et al. 2004).

Similar to the Library Sort, after the \( 2^l \)-th element insertion, the array \( A \) need to be rebalanced with the cost specified in Lemma 8, but we can save the cost of normalization. i.e., If we do apply the rotated list divisions recursively, from Theorem 4, the optimal level (i.e., \( l \)) grows after the \( 2^l \)-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 the rebalances from \( O(n) \) to \( O(r) \). 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 9.** It is possible to query the sum of all previous gaps before the rotated list \( L_i \) in constant worst case time and updates in \( O(r) \) worst case time, with only extra \( O(r \log(r)) \) bits space.

**Proof.** For simplicity, we do not consider gaps after the last element. Each rotated list \( L_i \) has at most \( \epsilon(f) \) gaps, the largest rotated list \( L_i \) has at most \( \epsilon(f) \) 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 (Raman et al. 2001) solved in the above bounds.

**Lemma 10.** \textsc{select} can be done in \( O(1) \) with extra \( O(\sqrt{n} \log n) \) space in Rotated Library Sort.

**Proof.** Trivial. We perform \text{select}(j, S) similar to \textsc{Rotated Sort}, but we need to add the sum of all previous gaps to \( j \) using Lemma 9, which also takes \( O(1) \) time.

It is possible to avoid rebalancing on the \( 2^l \)-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 \( \epsilon(f) \) elements. When \( J_i \) is full after a hard exchange, we pop the largest \( \delta = \epsilon(f) \) elements after the hard exchange and
perform the soft exchange of $\delta$ elements on the rotated lists $(J_{k+1}, \ldots, J_{r-2})$. $\delta$ will get smaller and eventually becomes zero. If we assume the elements of the input sequence are randomly distributed, $\delta$ 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 6.** insert in Rotated Library Sort can be done in amortized $O(\log r + f(r))$ operations.

**Proof.** For insert in Rotated Library Sort, hard exchanges on $J_k$ are unavoidable initially. However, the larger the $\epsilon$ is, fewer hard exchanges on $J_{k-1}$ will be required at the end. Therefore, the worst case scenario happens when insertions occur at $J_k$ where $k = r/2$. Each insertion consists of a binary search with $O(\log r)$ 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 consist the $((2\epsilon f(k) - 1))$-th insertion require only $O(\log r + f(k))$ 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 $(\epsilon f(k))$-th and $((\epsilon f(k))$-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 $O(\log r + f(k))$ 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 $O(\log r + f(k) + (r - k + f(r))/\epsilon)$. So with a large enough $\epsilon$, the insertion cost in the worst case scenario is close to amortized $O(\log r + f(k)) \leq O(\log r + f(r))$ operations, instead of $O(r + f(r))$ from Lemma 5, which is clearly an improvement. $\square$

8 Conclusions

This paper presents 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 (Bender et al. 2004) and does not rely on the distribution of input. It shows that the Rotated Insertion Sort can be done in $O(n^{1.5} \log n)$ time with $O(w)$ temporary space, which is a tight space bound; or it can be run in $O(2n^{1+\epsilon})$ operations, using only a lower order $\Theta(\sqrt{n} \log n)$ 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


