A Heap-Based Optimal Inversions-Sensitive Sorting Algorithm *

Undergraduate Senior Honors Thesis - Srinath Sridhar
Supervising Professor: Vijaya Ramachandran

April 18, 2003

We introduce a heap-based sorting algorithm that makes $n \lg(I/n) + O(n \lg\lg(I/n) + n)$ comparisons and runs in time $O(n \lg(I/n) + n)$, where I is the number of inversions in the input sequence and n the number of items to be sorted. The coefficient 1 in the leading term for the number of comparisons matches the information-theoretic lower bound. The algorithm is simple and uses elementary data structures.

1 Introduction

We consider the problem of designing a sorting algorithm that is adaptive to the pre-sortedness in the input sequence as measured by the number of inversions. Adaptive algorithms are particularly useful when handling applications where input sequences are partially sorted. Mannila [Ma85] formalized the concept of presortedness, and identified the number of inversions as an important measure.

We assume the input sequence $X =< x_1, x_2, \cdots, x_n >$ has distinct keys. An *inversion* is an ordered pair (i, j), where $x_i > x_j$ and $i < j$. Hence the total number of inversions in X is

$$I = Inv(X) = |\{(i, j) | 1 \leq i < j \leq n \text{ and } x_i > x_j\}|$$

*Supported in part by Undergraduate Research Opportunities Program (UROP) at UT-Austin for which funding was provided by Cisco Systems and Proctor & Gamble. Srinath Sridhar is also a recipient of the Nortel Networks scholarship.

The information-theoretic lower bound on the number of comparisons required to sort X was shown to be $n \lg(I/n) + O(n)$[1] in [GMPR77]. Hence an algorithm is time-optimal if it runs in $O(n \lg(I/n) + n)$. It is well-known that trying to measure the actual constants involved in an algorithm is not useful. However the number of comparisons performed by an algorithm remains invariant across different machines and platforms. For operations such as sorting big records or files, comparisons consume significant amount of time. This makes measuring the number of comparisons performed by an algorithm important.

In this paper, we describe a new time-optimal algorithm that makes $n \lg(I/n) + O(n \lg\lg(I/n) + n)$ comparisons. This is an *optimal* algorithm for inversions-sensitive sorting in the sense that it is time-optimal and the number of comparisons it performs matches the information-theoretic lower bound up to lower order terms. (To be precise, the number of comparisons performed by our algorithm is optimal with respect to its leading term and near optimal with respect to the second term. This is explained in the following section.)

2 Earlier Results

Adaptive sorting using the finger trees data structure introduced in [GMPR77], was the first inversions-sensitive time-optimal sorting algorithm. Mehlhorn [Me79] introduced an algorithm with the same time bounds as finger trees. Both of these algorithms are considered impractical. As summarized by Elmasry [El02], other algorithms that are time-optimal and inversions-sensitive are Blocksort [LP96] which runs in place and tree-based Mergesort [MEP96] which is time-optimal with respect to several other measures of pre-sortedness. Splitsort [LP91] and adaptive Heapsort [LP93] require $2.5n \lg n$ comparisons. Splaysort by repeated insertions [ST85] is another time-optimal algorithm shown in [MPW98] to be practically efficient. A survey of adaptive sorting algorithms is given in [Ca92].

Elmasry [El02] introduced Binomial and Trinomial Sort algorithms. Binomial Sort algorithm requires $2n \lg(I/n) + O(n)$ comparisons and Trinomial Sort algorithm requires $1.89n \lg(I/n) + O(n)$ comparisons. The algorithms

[1] For the rest of the paper we use lg to denote $\log_2$

use structures similar to binomial and trinomial heaps. However there appears to be a major problem with the 'glue' step in Binomial Sort. In this write-up, we correct this error and also reduce the number of comparisons.

We describe an algorithm that makes $n \lg(I/n) + O(n \lg\lg(I/n) + n)$ comparisons and is time-optimal. Our algorithm is based on Binomial Sort [El02], but we changed the invariants, modified some of the procedures and removed the problematic glue step. We have also added new components to improve the upper bound on comparisons. Therefore, we describe our algorithm from scratch.

Very recently Elmasry and Fredman have developed an optimal (time and comparisons) inversions-sensitive sorting algorithm [EF03]. The number of comparisons performed by their algorithm is $n \lg(I/n) + O(n)$. It is based on an insertion sort approach. From a theoretical stand point, both our algorithm and the Elmasry-Fredman algorithm achieve the coefficient 1 in the leading term for comparisons. Their algorithm has a better lower order term. However, we note that [EF03] state that they have insetion-sort and merge-sort based approaches. They however do not refer to any heap-based algorithm. Our algorithm is heap-based. Also, [EF03] refer to the Trinomial Sort algorithm in [El02] as a practical algorithm, and they ask for a practical algorithm that is competitive in the worst-case with the best non-optimal algorithms. Since our algorithm is arguably simpler than Trinomial Sort, it could serve as a candidate for a practical, optimal adaptive sorting algorithm. Also our algorithm is based on very simple data structures, and our results are entirely self-contained. In contrast, the algorithm in [EF03] uses the more involved data structure in [AL90]. (We do not know whether our algorithm is competitive in the strong sense asked by Elmasry and Fredman. This will require experimentation).

3 The Main Algorithm

3.1 An Overview of the Algorithm

A high-level description of the algorithm is given in pseudo-code 1. Terms and details are described later. The concept of an Inv-Adaptive transform [El02] is explained in section 3.2.2. The 'Oracle' mentioned in lines 2 and 3 is treated as a blackbox in this section, and is described and analyzed in section 4 and 5.

3.1.1 Pseudo-code 1: A High level description

1. Perform an Inv-Adaptive Binomial Build-Heap to obtain a binomial heap P, with at most $\lg n$ roots, placed in the "*root-list*".

2. Initialize an Oracle that return the root-node with the minimum key.

3. Loop for n times. (**Main Loop**)

   (a) $w := find\text{-}min(Oracle)$, print $key(w)$

   (b) Delete w from *root-list* and perform $fix\text{-}up$ step

   (c) Update Oracle

3.2 Terms

3.2.1 Binomial Queue

A rank k binomial tree B_k [Vu78] is defined recursively. The rank of a binomial tree is equal to its height and the rank of a node in a binomial tree is equal to the height of the subtree rooted at that node. B_0 is a single node tree. Tree B_k consists of 2 copies of B_{k-1} linked together making the root of one of the trees, the rightmost child of the other. We will use binomial trees with the following properties for B_k, a rank k binomial tree:

1. There are 2^k nodes.

2. The height of the tree is k

3. The root has k children and if the children of the root are numbered from $left$ to $right$ by $0, 1, \cdots, k-1$, child i is the root of subtree B_i.

4. There are $\max(1,\ 2^{k-i-1})$ nodes of rank i in a binomial tree B_k where $0 \leq i \leq k$

A *binomial queue* [Vu78] has the structure of a binomial tree. Associated with each node is a key, and heap property is maintained. We assume a good implementation of a binomial queue, where constant time access to the leftmost child, rightmost child and the right sibling of a node is possible.

Properties 1-3 are well known and can be found in Algorithms books such as [CLRS01]. We show the proof for the 4^{th} property of Binomial Trees.

Proof by Induction (of the 4^{th} property of binomial trees): B_0 has one node of rank 0. Tree B_1 has one node of rank 1 and 1 node of rank 0. By inductive hypothesis let the above property be true for all trees B_k where $k \leq p-1$. Now consider B_p. It is composed of two B_{p-1} trees linked together. The root of B_p is the only node whose rank is increased (from B_{p-1}). The rank of the root of B_p is one greater than the rank of the root of $B_p - 1$. Now consider any i, $0 \leq i \leq p-2$. By inductive hypothesis, the number of nodes of rank i on each of the two B_{p-1} trees is $2^{p-1-i-1}$. Hence, the number of nodes of rank i on B_{p-1} is 2^{p-i-1}. There is only one node of rank $p-1$ and only one node of rank p. This completes the proof by induction.

3.2.2 Inv-Adaptive Transformation:

Given a forest F of rooted ordered trees with a key at each node, $Pre(F)$ is defined as the pre-order traversal of F. The trees and the children of a node are traversed in left to right order. A transformation on F that results in F' is called *Inv-Adaptive* if the number of inversions in $Pre(F')$ is no more than the number of inversions in $Pre(F)$ [El02].

3.3 Basic Steps

3.3.1 Heapify:

This operation is performed at a root when the heap property holds at all nodes of the queue except the root. Heapify amounts to sinking the key at the root to the correct position. We now describe the version of heapify given in [El02]. In order to perform heapify, we maintain at each node a prefix minimum pointer (pm), pointing to the node with the minimum key among its left siblings. Heapify starts by finding the path from the root to the leaf, where every node has the smallest key among its siblings. This is done by following the *pm* pointers of the rightmost child at each level. The key value of the root is compared to nodes of the minimum path in a bottom-up manner. The key of the root is inserted in the appropriate position and all the keys above this position are shifted up. The pm pointers are updated. The number of comparisons required for this procedure is $\lg k + 1$, where k is the size of the tree [El02]. Since for nodes in the minimum path, either the pm pointer is updated or a comparison with the root is made, but not both. The length of the minimum path and the number of right siblings to the nodes in the minimum path adds up to $\lg n$.

We use the above method with one change. We place the children of each root on a heap instead of maintaining pm pointers. This heap is updated during heapify. With this addition the number of comparisons made during heapify is increased to $\lg n + 1 + O(\lg \lg n)$, but the benefit is that we can also perform a constant number of insertions and deletions on this heap within the same bound. The reason for performing heapify in this manner will become clear later.

3.3.2 Build-Heap:

This operation is performed in step 1 to convert the input sequence of n items into a binomial queue. To build a binomial queue B_k, we first link two B_{k-1} roots without a comparison by making the right root, the rightmost child of the left root. If the key of the root of the right queue is smaller than the key of the root of the left queue, then the two values are swapped and a heapify is performed on the root of the right queue. This swap is therefore Inv-Adaptive. The number of comparisons needed to build a heap is defined by the recurrence: $B(n) = 2B(n/2) + \lg n + 2$ and B(2) = 1, hence $B(n) = \theta(n)$.

3.4 Pseudo-rank, Invariants and Lightness

Associated with each node is a non-negative integer *pseudo-rank*. The pseudo-rank r_x of a node x is an approximation to its binomial rank. This is similar to, though not the same as, the 'pseudo-rank' in [El02].

The following invariants will hold at the start of each iteration of the Main Loop (step 3 of pseudo-code 1).

Invariant 1: For all roots x, r_x is strictly greater than the pseudo-ranks of all the roots in the root-list to the left of x.

Invariant 2: A root of pseudo-rank r has children with consecutively increasing pseudo-ranks from 0 to $r-2$, $r-1$ or r
Invariant 3: If a child of the root has pseudo-rank r, then its structure is identical to either B_r or B_{r-1}.
Invariant 4: For any node x that is neither the root nor the child of the root, the sub-tree rooted at x is a binomial tree and its pseudo-rank equals its binomial-rank.

A root node of pseudo-rank r is a *light node* if its rightmost child has pseudo-rank $r-2$. A child of a root is a *light node* if its pseudo-rank is one greater than its binomial rank. (A leaf is never light.) All other nodes are normal nodes. Initially, when we have a collection of binomial trees, the pseudo-rank equals the binomial rank for every node, hence all nodes are normal, and invariants 1-4 hold.
Lemma 1: If x is the root of any tree T, then $2^{r_x-2} < size(T) \leq 2^{r_x+1}$.
Proof: The quantity $size(T)$ is maximized when x has children of pseudo-ranks 0 through r_x and all the children are normal nodes. Therefore, $size(T) = 1 + 2^0 + 2^1 + ... + 2^{r_x} = 2^{r_x+1}$. The quantity $size(T)$ is minimized when x has children of pseudo-ranks 0 through $r_x - 2$ and all the children are light(except the node of pseudo-rank 0). Hence, $size(T) = 1 + 1 + 2^0 + 2^1 + ... + 2^{r_x-3} = 1 + 2^{r_x-2}$.[]

3.5 Fix-up

A root node, say w, has been removed. The goal of **fix-up** is to process the children of w, so that we can merge new roots into the root-list while maintaining invariants 1-4. The fix-up step performs two operations: **Combine** (to enforce invariant 2) and **Fusion** (to enforce invariant 1). Pseudo-code 2 describes the fix-up step. The Combine and Fusion pseudo-codes are called during the fix-up routine.

3.5.1 Pseudo-code 2: Pre-processing for fix-up

Recall that P is the Inv-adaptive heap which was built in step 1 of pseudo-code 1; w is the node deleted in the Main Loop; For node x, let k_x denote its key.

1. x := root to the immediate left of w in the root-list of P before w's deletion

2. If x does not exist then merge the remaining children of w as roots into the root-list of P and return

3. else do

   (a) y := leftmost child of w

   (b) Define Working Forest WF :=sub-list of siblings of y up to either

      i. node with pseudo-rank $r_x + 1$ if present, else

      ii. node with pseudo-rank r_x if present, else

      iii. the node with pseudo-rank $r_x - 1$

   (c) If WF contains only one node (which is y) then do

      i. If x contains children then perform **fusion** step (pseudo-code 4)

      ii. else do (this is a special case of fusion)

         A. Make x the parent of y and update C_x

         B. If $k_x > k_y$ then swap the two keys(not nodes). fi.

         C. $r_x := 1$ and merge x and the remaining children of w as roots into the root-list of P

   (d) else perform **combine** step (pseudo-code 3)

3.5.2 Pseudo-code 3: Combine

1. $r_y := r_x + 1$ and make y the parent of all the other nodes of WF

2. Convert C_w to C_y.

3. Traverse through the nodes in WF to find the first normal node h

4. If h is found then do

   (a) Decrease the pseudo-ranks of all left siblings of h by 1

   (b) s := the rightmost child of h

(c) Splice out the sub-tree rooted at s and make it the immediate right sibling of h

(d) $r_s := r_h$; $r_h := r_h - 1$ and update C_y

5. else

   (a) Decrease the pseudo-ranks of all children of y by 1

   (b) Case 1: Rightmost child of y has pseudo-rank r_x
   or Case 2: Rightmost child of y has pseudo-rank $r_x - 1$

   $r_y := r_x + 1$

   (c) Case 3: Rightmost child of y has pseudo-rank $r_x - 2$

   Perform heapify and **fusion** on y and return.

6. Perform heapify on y

7. Merge the new roots created during the combine/fusion steps and the other children of w as roots into the root-list of P

3.5.3 Pseudo-code 4: Fusion

1. $z :=$ the rightmost child of x

2. Case 1: $r_z = r_x - 2$

   (a) If $k_x > k_y$ then do

   Swap the keys (not nodes) of x and y and perform heapify on y

   (b) $r_y := r_x - 1$ and make y the rightmost child of x and update C_x

3. Case 2: $r_z = r_x - 1$

   (a) If $k_x > k_y$ then do

   Swap the keys (not nodes) of x and y and perform heapify on y

   (b) $r_y := r_x$ and make y the rightmost child of x and update C_x

4. Case 3: $r_z = r_x$

(a) Splice out the sub-tree rooted at z, make z a new root and initialize C_z.

(b) If $k_z > k_y$ then do

Swap the keys (not nodes) of z and y and perform heapify on y

(c) Case a: z has structure B_{r_x}

$r_y := r_x$, $y :=$ the rightmost child of z, $r_z := r_x + 1$; update C_z

(d) Case b: z has structure $B_{r_x - 1}$

$r_y := r_x - 1$, $y :=$ the rightmost child of z, $r_z := r_x + 1$; update C_z

3.6 Correctness

Invariants 1 through 4 hold at the start of the first iteration of the Main Loop, when the root-list of P is a collection of binomial trees. In pseudo-code 2, since $r_w \geq r_x + 1$, by Invariant 2, w has a child of pseudo-rank $r_x - 1$, and hence WF contains nodes of pseudo-ranks $0, \cdots, r$, where $r_x - 1 \leq r \leq r_x + 1$. We now argue that the invariants are restored after Combine and Fusion, and hence after Fix-up. Step 3(c)(ii)C in Pseudocode 2 is analyzed at the end of this section.

3.6.1 Combine:

A new root y is created at the end of combine. We show that r_y does not violate invariant 1 and that children of y preserve invariants 2 and 3.
Case when h is found (line 4): After y was made the root, its children formed an increasing sequence of pseudo-ranks $1, 2, \cdots, r_h - 1, r_h, r_h + 1, \cdots$. When the pseudo-ranks of all the nodes to the left of h is decreased by 1, the sequence of pseudo-ranks becomes $0, 1, 2, \cdots, r_h - 2, r_h, r_h + 1, \cdots$ To resolve the difference of 2 in pseudo-rank between h and its left sibling, we split h in such a way that we could assign $r_h - 1$ to one part and r_h to another. Since h was a normal node, its structure is identical to B_{r_h}. The sub-tree rooted at s has the structure of $B_{r_h - 1}$. This allows us to assign the sub-tree rooted at s pseudo-rank r_h and make it a light node. The remaining sub-tree

rooted at h has structure of B_{r_h-1} and can be given a pseudo-rank $r_h - 1$. Hence the split obeys invariant 3. This restores the increasing sequence of pseudo-ranks. Hence invariant 2 is also maintained. Even if the last child of y has pseudo-rank $r_y - 2$ and gets split, it would become a light node of rank $r_y - 2$. This preserves invariants 1-4.
Case when h is not found (line 5): If the rightmost child of y has pseudo-rank r_x or $r_x - 1$, we can assign y pseudo-rank $r_x + 1$ without violating invariant 2. If the rightmost child of y has pseudo-rank $r_x - 2$ then we cannot assign y pseudo-rank $r_x + 1$. In this case we perform a fusion step.

3.6.2 Fusion:

Lemma 2: If a new root of tree T with pseudo-rank r is created, $size(T) \geq 2^{r-1}$. (In other words, the new root is either a normal node, or a light root of a binomial tree B_{r-1})
Proof: Below along with the rest of the correctness, we discuss the cases when new roots can be created and show that this property is true. This is in case 3 in fusion and in the section on merging new roots.[]
Lemma 3: If a fusion step is required then the tree rooted at node y is a binomial tree of pseudo-rank $r_x - 1$.
Proof: A fusion step is required only if all the children of y were found to be light. Therefore using invariant 3, we can say that the sub-trees rooted at the children of y (after combine & before fusion) have structures $B_0, \cdots, B_r$, where $r = r_{x-2}$ or r_{x-1} or r_x. Now, all the children of y are normal nodes. Moreover, if the fusion step is required then the children of y need to form a sequence of $B_0, \cdots, B_{r_{x-2}}$. Therefore, y would have structure B_{r_x-1}, a binomial tree with pseudo-rank = binomial-rank = $r_x - 1$.[]

We now prove the correctness of the different cases of fusion step. We show that nodes r_x and r_z do not violate invariant 1. The children of x and z could change during fusion. We show that invariants 2 and 3 hold for all children of x and z. We also show that z does not violate Lemma 2.
Case 1: $r_z = r_x - 2$. If $k_x > k_y$ then swapping the two keys is Inv-adaptive. This is because k_y is smaller than the keys of all descendants of x and y. Hence shifting the key of y to the first place among trees rooted at x and y can only reduce inversions. Heapify is inv-adaptive. Making y the rightmost child of x does not change the pre-order traversal, and so is inv-adaptive. Since y has the structure B_{r_x-1}, assigning it pseudo-rank $r_x - 1$ preserves

invariants 2 and 3.

<u>**Case 2:**</u> $r_z = r_x - 1$. This case is similar to case 1, the only difference being that y is assigned pseudo-rank r_x. This makes y a light node. Invariants 2 and 3 are preserved.

<u>**Case 3:**</u> $r_z = r_x$. Splicing out z and making it a new root does not change the pre-order traversal and therefore is inv-adaptive. Since z was a child of the root, by invariant 3, the sub-tree rooted at z is binomial.

<u>Case 3a:</u> z has structure B_{r_x}. Since we increment r_z to $r_x + 1$, it is allowed to have children with pseudo-rank up to $r_x + 1$. We assign y pseudo-rank r_x and make it the rightmost child of z. This preserves invariant 2. Invariant 3 is preserved since y is a binomial tree B_{r_x-1}. Before being fused with y, z was the root of a binomial tree B_{r_z}. Hence the size of the tree rooted at z is $\geq 2^{r_z}$. This preserves Lemma 2.

<u>Case 3b:</u> z has structure B_{r_x-1}. We assign y pseudo-rank $r_x - 1$ which is 2 lower than $r_x + 1$. Hence invariant 2 holds. Invariant 3 holds since y is root of a binomial tree B_{r_x-1}. The size of the tree rooted at z before being fused with y is 2^{r_x-1} and size of the tree rooted at y is 2^{r_x-1}. Therefore, size of the new root z after fusion is 2^{r_x} (This tree is a binomial tree B_{r_x}). Hence Lemma 2 holds.

3.6.3 Merging new roots

In step 3(c)(ii)C of pseudo-code 2 and at the end of the combine, we merge the new roots into the root-list. Each child c of w that lies outside WF has the property that $r_x + 1 < r_c < r_w + 1$. This is because if $r_c \leq r_x + 1$ it would be a part of WF. Also $r_c \leq r_w$ by invariant 2. The root created during the combine and fusion is a root of pseudo-rank $r_x + 1$. The first root to the right of w (before its deletion) has pseudo-rank $> r_w$. Hence the new root created in combine/fusion and the other children of w can be merged into the root-list without violating Invariant 1.

Also consider any child c of w that lies outside WF and becomes a new root. Let it have pseudo-rank r. By invariant 3, c has the structure of B_r or B_{r-1}. Hence the size of the sub-tree rooted at c is $\geq 2^{r-1}$. This preserves Lemma 2.

3.7 Time Complexity Analysis

3.7.1 Analysis of Combine:

Consider the j^{th} iteration of the Main Loop. We perform one heapify operation on a tree of pseudo-rank $r_x + 1$. By Lemma 1, this takes less than $r_x + 3 + O(\lg r_x)$ comparisons. Now $k_w < k_x < k_z$ where z is any descendant of x. Hence by Lemma 1, there are at least $2^{r_x - 2} = I_j$ (say) inversions associated with w. Hence, $r_x + 3 + O(\lg r_x) = \lg(I_j) + 5 + O(\lg\lg I_j)$

Hence, total number of comparisons over the entire algorithm for the combine operation is bounded by $\sum_{j=1}^{j=n}(\lg(I_j) + O(\lg\lg(I_j)) + 5)$. For a convex function $f(x)$ and for any function $p(x)$ with for all x, $0 \leq p(x) \leq 1$ and $\sum p(x) = 1$, according to Jensen's inequality,

$$\sum_x p(x) f(x) \geq f(\sum_x p(x) x).$$

Using the convexity of $-\lg(x)$ we get,

$$\begin{aligned}
\sum_{x=1}^{x=n} (1/n)(-\lg(I_x)) &\geq -\lg(\sum_{x=1}^{x=n} (I_x)/n) \\
\sum_{x=1}^{x=n} (1/n)(\lg(I_x)) &\leq \lg(\sum_{x=1}^{x=n} (I_x)/n) \\
\sum_{x=1}^{x=n} (\lg(I_x)) &\leq n\lg(\sum_{x=1}^{x=n} (I_x)/n) \\
&\leq n\lg(I/n)
\end{aligned}$$

since $I_1 + I_2 + ... + I_n \leq I$.

Similarly Using the convexity of $-\lg\lg(x)$ we get,

$$\begin{aligned}
\sum_{x=1}^{x=n} (1/n)(-\lg\lg(I_x)) &\geq -\lg\lg(\sum_{x=1}^{x=n} (I_x)/n) \\
\sum_{x=1}^{x=n} (1/n)(\lg\lg(I_x)) &\leq \lg\lg(\sum_{x=1}^{x=n} (I_x)/n) \\
\sum_{x=1}^{x=n} (\lg\lg(I_x)) &\leq n\lg\lg(\sum_{x=1}^{x=n} (I_x)/n) \\
&\leq n\lg\lg(I/n)
\end{aligned}$$

since $I_1 + I_2 + ... + I_n \leq I$.

Therefore the entire time is bounded by $n\lg(I/n) + O(n\lg\lg(I/n)) + 5n$

3.7.2 Analysis of Fusion:

In the Fusion step we perform at most one heapify operation, one initialization of C and one update of C on a tree with pseudo-rank r_x. This requires less than $2r_x + 4 + O(\lg r_x)$ comparisons. The $O(\lg r_x)$ terms comes from updating the C heap. The fusion step occurs only in the case when y had light children of pseudo-ranks from 1 to $r_x - 1$. Using Lemma 2, we can say that there should have been a root of pseudo-rank $r_x + 1$ in the root-list of P for $2^{r_x} - 2^{r_x - 1} = 2^{r_x - 1}$ deletions and no fusion could have taken place on this tree. Borrowing a constant from each of these deletions would be sufficient to pay off for the comparisons performed during the fusion step. Hence, all fusion steps during the algorithm would require only $O(n)$ comparisons.

3.7.3 Initialization of the C heaps:

For any root x, C_x is a separate heap containing its children. It takes $O(\lg k)$ time to initialize C_x if the size of the subtree rooted at x is k. Durin Fix-up we need to update C_w so that it corresponds to C_y. This can be done by performing multiple deletes for each of the children of w that are not children of y from highest to lowest pseudo-ranks. The analysis for the total number of comparisons required for the initialization of all C's is deferred until case 1.2 of section 5.4, where we prove that this is linear.

Elmasry [El03] has pointed out to us a different approach that does not use the C heaps, but instead slightly modifies the way a node is promoted as a root during the combine step, as follows. After the root is removed, in the above description a node of pseudo-rank 0 (node y) becomes the new root (line 1 of pseudo-code 3). Instead of this, we compare the keys of y and the right sibling of y and promote the node with the larger key. This way, the prefix-minima pointers of the other right siblings of y need not be updated. Though this operation is no longer Inv-Adaptive it can create at most n inversions during the course of the algorithm and therefore would not affect the time-complexity.

4 A Simple Oracle

In section 3 we treated the oracle as a black-box and described the algorithm. We now provide two different techniques to realize the oracle. In this section we describe a simple method that is optimal when the number of inversions in the input is $\omega(1)n \lg\lg n$. This sets up the basic idea which is used in the main oracle, described in section 5, to achieve optimal bounds in general.

The main idea in the simple oracle to arrange the nodes of the root-list in a secondary binary heap S. The reason for performing the right to left incremental pairing pass in the Binomial Sort algorithm [El02] was to find the node with the minimum key. By arranging the nodes in S, the structure of P becomes more rigorous.

4.1 Description of the Sorting Algorithm Using a Simple Oracle

We now go back to pseudo-code 1. Lines 1 and 2 are self-explanatory.We now walk through one iteration of the loop in lines 3-7. In line 4, the root w in the root-list of P is found by querying the Oracle. Line 5 performs the fix-up step. Since the fix-up could create new roots, the Oracle is updated in line 6.

Time taken by the algorithm is the sum of the time taken for all the fix-up steps and the time taken for updating the Oracle. It is assumed that the oracle can answer each query in constant time. The number of comparisons similarly is the sum of the number of comparisons spent over all the fix-up steps and the number of comparisons for updating the Oracle.

4.2 Description and Analysis

4.2.1 Initialization:

We just insert all the roots into S one by one. Time for each insertion into S is bounded by $O(\lg\lg n)$. There can be at most $\lg n$ nodes in the root-list. Therefore the total time for initializing the Oracle is $O(\lg n \lg\lg n)$.

4.2.2 Updates:

If w has no children, then it can be removed from P by changing its key value to ∞ performing a $heapify$ operation. This would take time $O(\lg\lg n)$. If w has children then we handle three cases.

Case 1: The root-list of P contains x, a new root w' with $r_{w'} = r_x + 1$ and new roots to the right of w'.

Case 2: The root-list of P contains x, which was fused together with a child of w. No new roots are to the right of x.

Case 3: The root-list of P contains x, a new root z that was formerly a child of x and w' was fused into z. No new roots are present to the right of z

<u>Case 1:</u> The minimum heap node of S, w is now replaced with w'. A heapify step on S is performed so that w' sinks to its correct location in S. The new roots to the right of w' in P are inserted into S.
<u>Case 2:</u> During the fusion step the key of x in P could have changed. The value k_x could have only decreased. Hence we perform a decrease-key operation in S on the node x. The first new root is inserted into S by changing the node w and performing a heapify on S. Other new roots are inserted in normal manner into S.
<u>Case 3:</u> The minimum heap node of S, w is now replaced with z. A heapify step on S is performed so that z sinks to its correct location in S. The new roots to the right of w' in P are inserted into S.

The analysis is quite straight-forward. A node is inserted into S at most once. This implies that the time spent and the number of comparisons on all the heapify operations is $O(n \lg\lg n)$. Also, at most one decrease-key or one delete (implemented by changing key to ∞ and performing a heapify) operation might be performed during each iteration of the loop. The total time and number of comparisons for all the decrease-key/delete operations is bounded by $O(n \lg\lg n)$. Hence the bound on the time/comparisons for the entire Oracle is $O(n \lg\lg n)$. Using this Oracle the new algorithm would require at most $n \lg(I/n) + O(n \lg\lg n)$ comparisons.

4.2.3 Optimality:

If the number of inversions $I > n \lg^{\omega(1)} n$, then

$$I/n > \lg^{\omega(1)} n$$

$$\lg(I/n) > \omega(1) \lg\lg n$$

$$n \lg(I/n) > \omega(1) n \lg\lg n$$

In this case, the above algorithm runs in optimal time. Moreover the number of comparisons is $n \lg(I/n) + o(n \lg(I/n))$ which achieves the constant coefficient 1 in the leading term, matching that of the information theoretic lower bound.

5 The Main Oracle

The Oracle is used in steps 2 and 3 of pseudo-code 1 to extract the minimum key in each iteration of the Main Loop. In this section we describe and analyze the Oracle.

Recall that the primary heap P is the main heap that is used in pseudo-code 1. In the Oracle we use secondary binary heaps $S[1 \cdots \lg\lg n + 1]$, containing the keys at the roots of the trees in P. We also use tertiary binary heaps $T[1 \cdots \lg\lg\lg n + 1]$, containing the keys at the roots of heaps in S.

5.1 Description and Initialization

The roots of the binomial queues in P are divided into $\lg\lg n + 1$ groups. Group i contains roots of P that have pseudo-ranks in the interval $[2^{i-1}, 2^i)$. This is maintained as an invariant throughout the algorithm. The $(\lg\lg n + 1)^{th}$ group contains roots of pseudo-rank up to $2^{\lg\lg n} = \lg n$. Since the maximum pseudo-rank of a root is $\lg n$, $\lg\lg n + 1$ groups are sufficient to include all the roots. We place the roots in group i in a binary heap $S[i]$. We now create a new list 'higher-list' of nodes corresponding to the minimum key of each group. We define a collection i to be the set of all nodes in higher-list of pseudo-ranks in $[2^{2^{i-1}}, 2^{2^i})$. The Tertiary binary heap $T[i]$ contains nodes in collection i and also the minimum node of collection $i+1$ (if it exists). Hence the minimum node of $T[i]$ is the suffix minimum of roots in P with pseudo-ranks in $[2^{2^{i-1}}, \lg n]$.

5.2 Initialization

We begin with a build-heap step on the nodes of group $(\lg\lg n + 1)$ to create $S[\lg\lg n + 1]$. Build-heap is then performed on group $\lg\lg n$ to create $S[\lg\lg n]$. This continues up to $S[1]$. Now we perform a build-heap on the nodes of collection $(\lg\lg\lg n + 1)$ to create $T[\lg\lg\lg n + 1]$. After this, we perform a build-heap on the nodes of collection $(\lg\lg\lg n)$ along with the minimum of $T[\lg\lg\lg n + 1]$ to form $T[\lg\lg\lg n]$. This procedure to create $T[i]$ continues until a Build-Heap creates $T[1]$. Now the heap minimum of $T[1]$, is the minimum of all the roots of P.

5.2.1 Analysis:

The number of nodes in the root-list of P is $\lg n$. Since build-heap takes linear time, the time to create S is $O(\lg n)$. Similarly, the time to create T is bounded by $O(\lg\lg n)$ since there are $O(\lg\lg n)$ groups. Initialization of the Oracle is therefore sub-linear.

5.3 Updates

After the fix-up of the root-list of P, we need to update the Oracle. Here we may encounter one of the following two cases (and sub-cases). We use the same notation as fix-up. Let y be the new root formed at the end of the combine step.

Case 1: No fusion occurs.
Case 1.1: $r_y = r_x + 1 = r_w$. No new root other than y remains in P.
Case 1.2: Several new roots that were formerly the children of w are present to the right of y with $pseudo - rank > r_y$.

Case 2: Fusion occurs.
Case 2.1: x is fused with y. No new root remains in the root-list of P.
Case 2.2: During the fusion step, z becomes a new root. Node y is fused with z. No other new root remains in P.

We now describe the updates for the above cases.
Case 1.1: First we will deal with the case when all children of w are combined together to form a single tree rooted at y. Since w was the smallest element of the root-list, it should be the smallest element in its secondary heap, say $S[j]$, and also in its tertiary heap say $T[l]$ and all tertiary heaps to the left of $T[l]$. In $S[j]$, the key of w is changed to that of y. Now this new key representing y is sunk into the correct position in heap $S[j]$, by a heapify operation. Similarly w in $T[l]$ is changed to the new minimum of $S[j]$ and a

binary heapify is performed in $T[l]$. Heap $T[l]$ now contains a new minimum node say u. We proceed to change the field of w to that of u in $T[l-1]$ and perform a heapify on $T[l-1]$ to obtain a new minimum node. This procedure (called 'propagate' for future reference) of changing the node corresponding to w with the node corresponding to the minimum of the tertiary heap to the right goes all the way up to $T[1]$.
Case 1.2: We insert all new roots to the right of x into their respective heaps in S and T. We start from the rightmost new root z' (say) and insert it into S and T as with y in case 1.1 (if z' is in the same group as w, it replaces w). The other new roots to its left are then inserted one by one into S and if needed into T. We extract w out of heaps S and T (if present). We now perform propagate starting from $T[l]$.
Case 2.1: $Key(x)$ could have decreased during the fusion step. First we perform a decrease-key operation on node x in $S[j']$ where j' is the group in which x is present. If x was previously present in collection l', we perform a decrease-key on node x in $T[l']$. If x because of the decrease-key in $S[j']$ became the minimum of $S[j']$, then it replaces the previous minimum of $S[j']$ in $T[l']$ and a decrease-key is performed. We delete w from $S[j]$ and $T[l]$ where j and l is the group and collection in which w was present and then perform propagate.
Case 2.2: Processing of z in this case is identical to that of y in case 1.1.

5.4 Analysis:

In this section we analyze the number of comparisons performed in each of the cases mentioned above. We note here that every node contains two fields: key and pseudo-rank. Throughout the algorithm, keys of the nodes are swapped and not the nodes themselves. Hence unless explicitly changed, a node's pseudo-rank remains the same.

Case 1.1: Let $r_w = r_y = p$. Since size of $S[j] \leq p$, time to update $S[j]$ is $O(\lg p)$. Since size of $T[l] \leq \lg p$ time to update $T[l]$ is $O(\lg\lg p)$. Node w belongs to collection $\lceil \lg(\lceil \lg p \rceil + 1) \rceil + 1$. Therefore the total time bound on all the heapify calls in T is $O(\lg\lg p)$.

As computed in section 5.4, number of inversions with respect to w is $\geq 2^{p-2} = I_j$. Therefore the time taken and the number of comparisons made during one update of the Oracle executing case 1.1 is $O(\lg\lg I_j)$.

Case 1.2: We define an interior node as any node in P that is not the root. An old root is defined as a root that was not created during the current iteration of the Main Loop. A new root is any root created during the current iteration of the Main Loop.

Inserting y and other new roots: Analysis for inserting y into S and into T follows from case 1.1 and takes $O(\lg\lg I_j)$ time. We now analyze the cost to insert the other new roots. During the course of the algorithm a node's pseudo-rank is increased only when it is made a new root during combine, or when it becomes a new root during fusion. Also note that once a node becomes a root, it never becomes an interior node.

Now consider any new root ρ that was created with pseudo-rank r_ρ and currently has pseudo-rank r'_ρ. We consider the cases: $r'_\rho \leq r_\rho$ and $r'_\rho > r_\rho$. Briefly, in the first case we use the binomial tree property 4 to show that the number of comparisons over all insertions in S and T is $O(n)$. In the second case we amortize the cost of insertions in S and T against other costs.

Case 1.2.A: $r'_\rho \leq r_\rho$. Since at the first iteration of the Main Loop, P is a collection of binomial trees, the number of times nodes of initial pseudo-rank r are inserted into S and T during the course of the algorithm is no more than $2^{\lg n-(r+1)}$, using property 4 of binomial trees. Inserting ρ in S and T takes time $O(\lg r'_\rho) = O(\lg r_\rho)$. Hence, the cost of insertions into S and T of all nodes ρ whose pseudo-rank r'_ρ at insertion is $\leq$ the initial pseudo-rank r_ρ is bounded by

$$O(2^{\lg n-3}\lg 2 + 2^{\lg n-4}\lg 3 + \cdots + 2^{\lg n-\lg n}\lg(\lg n-1))$$

$$= O(n/2^3\lg 2 + n/2^4\lg 3 + \cdots + n/2^{\lg n}\lg(\lg n-1))$$

$$= O(n(\lg 2/2^3 + \lg 3/2^4 + \cdots + \lg(\lg n-1)/2^{\lg n-1}))$$

We now bound the decreasing series $\sum_{i=2}^{\infty}\lg i/2^i$ $a_2 \leq 0.5$

Ratio between successive terms $\leq (\lg 4/16)/(\lg 3/8) = 1/\lg 3 \leq 0.6$ Therefore, $\sum_{i=2}^{\infty}\lg i/2^i \leq 0.5(1/(1-0.6)) = 1.25$ Hence, the total number of comparisons during all case 1.2.A executions of Oracle is $O(n)$.

Case 1.2.B: $r'_\rho > r_\rho$. Node ρ was a child of w in the previous iteration - an internal node. The only way in which an interior node could have greater pseudo-rank than it initially had, would be if it was promoted during combine and immediately was fused. As explained in section 5.4, fusion occurs rarely. Moreover the analysis of fusion accounted for $2r_x + O(\lg r_x)$ comparisons per fusion involving x. All instances of case 1.2.B can be charged on the $O(\lg r_x)$ factor.

Updating C_y: We perform a number of deletes from C_w to create C_y. We note that each node of pseudo-rank r that is deleted from C_w is also inserted into S with the same pseudo-rank. Hence each such deletion can be charged against the $O(\lg r)$ comparisons spent for its insertion into S. Hence the number of comparisons made to convert C_w to C_y throughout the algorithm is linear.

Case 2.1: The analysis is the similar to case 1.1

Case 2.2: The analysis is identical to case 1.1

The total time spent by the Oracle over all calls is bounded by $O(\sum_{j=1}^{j=n} \lg\lg I_j) + O(n)$. We now use Lagrange's multipliers to maximize $f(I_1, I_2, ..., I_n) = \sum_{i=1}^{i=n} \lg\lg(I_i)$ subject to the constraint $g(I_1, I_2, ..., I_n) = \sum_{i=1}^{i=n} I_i \leq I$, where I is the total number of inversions in the input sequence. We want to find a scalar λ such that $\bigtriangledown f = \lambda \bigtriangledown g$.

$$\frac{\partial f}{\partial I_i} = \frac{\partial \frac{\ln\ln I_i}{ln 2}}{\partial I_i}$$

$$= \frac{1}{I_i \ln 2 \ln I_i}$$

$$\lambda \frac{\partial g}{\partial I_i} = \lambda$$

Therefore,

$$(1/\lambda) = I_1 \ln 2 \ln I_1 = I_2 \ln 2 \ln I_2 = \cdots = I_n \ln 2 \ln I_n$$

and so,

$$I_1 = I_2 = \cdots = I_n$$

Therefore, f is maximized at $I_i = I/n$ for all i and the maximum is $n \lg\lg(I/n)$.

Hence the total time spent by the Oracle is $O(n \lg\lg(I/n) + n)$.

This gives us the main theorem.

Theorem: The heap-based algorithm sorts a sequence of n items with I inversions in time $O(n \lg(I/n)+n)$ and makes at most $n \lg(I/n)+O(n \lg\lg(I/n)+n)$ comparisons.

Proof: The proof follows from the invariants, lemmas 1-3 and the analysis of the Oracle.

6 Discussion

6.0.1 Lower order terms:

An open question is whether we can reduce the $O(n \lg\lg(I/n))$ term to $O(n)$ thus matching the lower bound to within a constant factor of the lower order term. However we consider this a minor theoretical question since this has already been achieved with insertion and merge sort based techniques [EF03]. The main issue here is whether this can be done by a method that uses simple data structures. Note that the $n \lg\lg(I/n)$ term in our algorithm is mainly contributed by n delete-min operations on heaps of size about $\log(I/n)$. Since $\log\log n$ is less than 6 for input sizes that we may expect to deal with in the foreseeable future, for practical purposes that term in our bound can be viewed as a linear term with a modest constant factor.

It is not possible to improve the $O(n \lg\lg(I/n))$ to $O(n)$ in this exact framework by improving the Oracle. This is because if each update of the Oracle can be performed in $o(\lg\lg k)$ time where $\lg k$ is the number of roots, then we can sort a sequence of $\lg k$ items in $o(\lg k \lg\lg k)$ time. This is not possible because of the sorting lower-bound.

6.0.2 Single copy of keys:

We need to maintain just one copy of the keys. The Secondary and Tertiary heaps maintain pointers to the primary heap items.

6.0.3 Choice & implementation of S and T:

Several versions of S and T would work including other types of heaps. We explain one method below. We always have no more than $\lg n$ nodes in S for the simple Oracle and a fixed number of nodes for each $S[i]$ and $T[i]$ in the main Oracle. The nodes that are in S or T but not in the root-list have key value ∞. Insertions into S and T, can be implemented as standard decrease-key operations reducing keys from ∞ to finite values. The array for the binary heap can be initialized at the beginning with ∞ values. Since the root-list would never contain more than $\lg n$ nodes for the simple Oracle and the fixed limit for the main Oracle, S and T would never run out of space.

7 Acknowledgements:

I greatly thank Prof. Vijaya Ramachandran for her guidance and encouragement. I also thank Prof. Mike Fredman for his comments on the write-up.

References

[AL90] A. Anderson and T. W. Lai. Fast updating of well-balanced trees. *Proceedings of Scandinavian Workshop on Algorithm Theory* (1990), 111-121

[Br78] M.Brown. Implementation and analysis of binomial queue algorithms. *SIAM J. Comput.* 7 (1978), 298-319.

[BT80] M.Brown and R.Tarjan. Design and analysis of data structures for representing sorted lists. *SIAM J.Comput.* 9 (1980), 594-614.

[Co00] R.Cole. On the dynamic finger conjecture for splay trees. Part II:The proof. *SIAM J.Comput.* 30 (2000), 44-85.

[Ca92] V.Estivill-Castro. A survey of adaptive sorting algorithms *ACM Comput. Surv.* vol 24(4)(1992), 441-476.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press, 2001

[El02] A. Elmasry. Priority Queues, Pairing and Adaptive Sorting. *ICALP 2002 Proceedings* LNCS 2380: p 183

[El03] A. Elmasry. Private communication, February 2003.

[EF03] A. Elmasry and M. Fredman. Adaptive Sorting and the Information Theoretic Lower Bound *STACS 2003 Proceedings*

[GMPR77] L.Guibas, E.McCreight, M.Plass and J.Roberts. A new representation of linear lists. *ACM Symp. Theory of Computing.* 9 (1977), 49-60.

[Kn98] D.Knuth. The Art of Computer Programming.Vol III:Sorting and Searching. *Addison-Wesley, second edition.* (1998).

[LP91] C.Levcopoulos and O.Petersson. Splitsort -An adaptive sorting algorithm. *Information Processing Letters.* 39 (1991), 205-211.

[LP93] C.Levcopoulos and O.Petersson. Adaptive Heapsort. *Journal of Alg.* 14 (1993), 395-413.

[LP96] C.Levcopoulos and O.Petersson. Exploiting few inversions when sorting: Sequential and parallel algorithms. *Theoretical Computer Science.* 163 (1996), 211-238.

[Ma85] H.Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Comput.* C-34 (1985), 318-325.

[Me79] K.Mehlhorn. Sorting presorted files. *Proc. 4th GI Conference on Theory of Computer Science.* , LNCS 67 (1979), 199-212.

[Me84] K.Mehlhorn Data Structures and Algorithms. Vol.1. Sorting and Searching. *Springer-Verlag, Berlin/Heidelberg.* (1984).

[MEP96] A.Moffat, G.Eddy and O. Petersson Splaysort: fast, versatile, practical. *Softw. Pract. and Exper.* Vol. 126(7)(1996), 781-797.

[MPW98] A.Moffat, O.Petersson and N.Wormald A tree-based Mergesort. *Acta Informatica, Springer-Verlag.* (1998), 775-793.

[ST85] D.Sleator and R.Tarjan. Self-adjusting binary search trees. *J.ACM* 32(3)(1985), 652-686.

[Vu78] J.Vuillemin. A data structure for manipulating priority queues. *CACM* 21(4) (1978),309-314.