On Adaptive Integer Sorting

Anna Pagh[1], Rasmus Pagh[1], and Mikkel Thorup[2]

[1] IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark
{annao, pagh}@itu.dk
[2] AT&T Labs — Research, Shannon Laboratory, Florham Park, NJ 07932
mthorup@research.att.com

Abstract

Abstract. This paper considers integer sorting on a RAM. We show that adaptive sorting of a sequence with qn inversions is asymptotically equivalent to *multisorting* groups of at most q keys, and a total of n keys. Using the recent $O(n\sqrt{\log \log n})$ expected time sorting of Han and Thorup on each set, we immediately get an adaptive expected sorting time of $O(n\sqrt{\log \log q})$. Interestingly, for any positive constant ε, we show that multisorting and adaptive inversion sorting can be performed in *linear time* if $q \leq 2^{(\log n)^{1-\varepsilon}}$. We also show how to asymptotically improve the running time of any traditional sorting algorithm on a class of inputs much broader than those with few inversions.

1 Introduction

Sorting is one of the most researched problems in computer science, both due to the fundamental nature of the problem and due to its practical importance. An important extension to basic sorting algorithms is *adaptive* sorting algorithms, which take advantage of any existing "order" in the input sequence to achieve faster sorting. This is a classical example of the trend in modern computer science to express the complexity of an algorithm not only in terms of the *size* of the problem instance, but also in terms of *properties* of the problem instance. On the practical side, it has been observed that many real-life instances of sorting problems do indeed have structure (see, e.g., [12, p. 54] or [13, p. 126]), and this can be exploited for faster sorting.

A classical measure of order, or rather disorder, is the number of *inversions* in the sequence, that is, the number of pairs of elements that are placed in wrong order compared to the sorted sequence. Using finger search trees one can obtain a sorting algorithm that sorts a sequence of n elements with qn inversions in time $O(n \log q)$, and there is also a lower bound of $\Omega(n \log q)$ on the number of comparisons needed to sort such a sequence in the worst case [8].

After Fredman and Willard's seminal work [6], there has been an extensive interest in algorithms that exploit features of real computers to break comparison-based lower bounds for the case of integer keys stored in single words assuming unit cost operations on words. The model used for such studies is the so-called *word RAM*. The main idea, which goes back to classical techniques such as radix

sort and hashing, is to exploit that one can work with the *representation* of integers. In combination with tabulation and word-level parallelism techniques this has lead to asymptotically very fast sorting algorithms. The currently fastest deterministic and randomized sorting algorithms use $O(n \log \log n)$ time and $O(n\sqrt{\log \log n})$ expected time, respectively, and linear space [9, 10].

The basic research question in this paper is how well we can exploit few inversions in integer sorting. The most obvious way forward is to use previous reductions from inversion sort to finger searching, but for integer keys, this approach has a matching upper- and lower-bound of $\Theta(n\sqrt{\log q / \log \log q})$ time [3, 4]. However, in this paper, we will get down to time $O(n \log \log q)$ and $O(n\sqrt{\log \log q})$ expected time, matching the best bounds for regular sorting. Also, more surprisingly, we will show that we can sort in linear time if for some constant $\varepsilon > 0$, we have $q \leq 2^{(\log n)^{1-\varepsilon}} = n^{1/\log^{\varepsilon} n}$.

Our results are achieved via reduction to what we call *multisorting*, which is the problem of sorting several groups of keys with some bound on the size of a group. The output is a list for each group of the keys of the group in sorted order.

To appreciate the power of multisorting, suppose the total number of keys over all groups is n and that all the keys have values in $[n]$. We can now prefix each key with the group it belongs to so that the values become pairs (i, x) where i is the group and x is the original value. Using an array over $[n]$, we can sort the keys with the new values in linear time with a 2-pass radix sort. Each group now has its keys sorted in a segment of the whole list, and these segments are easily extracted. Thus, the whole multisort is performed in linear time. If, however, the group size q is much smaller than n, say, $q = 2^{\sqrt{\log n}}$, we would not know how to sort the groups individually in linear time. This power of multisorting has been used for regular (non-adaptive) sorting in [9].

We show in Section 3 that inversion sorting and multisorting are equivalent in a strong sense, namely, that we can multisort using only one inversion sorting and linear extra time and the other way around.

Theorem 1. *The following problems can be reduced to each other in $O(n)$ time and space on a RAM with word size w:*

- **Multisorting:** *Sort n w-bit integer keys in groups, where each group consists of at most q keys.*
- **Inversion sorting:** *Sort a list of n w-bit integer keys with a known upper bound qn on the number of inversions.*

The reduction of inversion sorting to multisorting assumes an upper bound qn on the number of inversions to be known. However, it is easy to make the algorithm automatically adapt to the number of inversions, with only a constant factor increase in running time. The idea is well known: Perform doubling search for the running time of the inversion sorting algorithm. For a precise statement of this reduction see Section 2.

Multisorting can of course be done by sorting each group separately. If a group has less than q keys, the recent randomized sorting algorithm by Han

and Thorup [10] sorts it in $O(\sqrt{\log \log q})$ expected time per key. Thus, for multisorting n keys in groups of size at most q, we get an algorithm running in $O(n\sqrt{\log \log q})$ expected time. More interestingly, in Section 4, we show that we efficiently can multisort much larger groups than we can solve individually. In particular, for any positive constant ε, we show that we can multisort in linear time with group size up to $2^{(\log n)^{1-\varepsilon}}$. Note that this is close to best possible in that if ε was 0, this would be general linear time sorting. On the other hand, if the groups are to be sorted individually, the largest group size we can handle in linear time is only $2^{(\log n)^{1/2-\varepsilon}}$ using the signature sorting of Andersson et al. [2]. Our efficient new multisorting result is as follows.

Theorem 2. *On a RAM, we can multisort n integer keys in groups of size at most q in linear space and $O(n(\log \log n)/(1+\log \log n - \log \log q))$ expected time. In particular, this is linear time for $q \leq 2^{(\log n)^{1-\varepsilon}}$, for any constant $\varepsilon > 0$.*

Combining with Theorem 1, we get the following adaptive sorting result:

Corollary 1. *We can sort n keys with qn inversions in linear space and $O(n(\log \log n)/(1+\log \log n - \log \log q))$ expected time. In particular, this is linear time for $q \leq 2^{(\log n)^{1-\varepsilon}}$, for any constant $\varepsilon > 0$.*

1.1 Our overlooked reduction

In a slightly weaker form (c.f. Lemma 1), our reduction from inversion sorting to multisorting is quite simple and works on a comparison-based pointer-machine model of computation. We find it quite surprising that it has been overlooked for so long. Of course, in the classical comparison based model, from a theoretical perspective, the reduction to finger searching is optimal in that it gets down to $O(n \log q)$ time for qn inversions.

From a practical perspective, however, a finger search data structure with all its parent, children, and horizontal pointers is pretty slow with a large space overhead. With our reduction, we partition our data set arbitrarily into groups of some limited size, that we sort with our favorite tuned sorting algorithm, be it quick sort, merge sort, or even insertion sort if the groups are small enough.

1.2 Other measures of disorder

It is folklore knowledge that if a sequence has r *runs*, that is, r increasing and decreasing subsequences, then one can sort in time $O(nf(r))$, where $f(r)$ is the time per operation on a priority queue with r keys. This time bound is incomparable to bounds in terms of the number of inversions. However, in Section 5 we consider the disorder measure Osc [11] which is a natural common generalization of both inversions and runs. Based on *any* non-adaptive sorting algorithm, we get a sorting algorithm that is extra efficient on inputs with low Osc disorder. In particular, the obtained algorithm will perform well on inputs with a small number of inversions or runs.

2 Eliminating the need for a bound on inversion number

From Theorem 1 it follows that if we can multisort n keys with group size at most q in time $S(n, q)$, then we can sort n keys with up to qn inversions in $O(S(n, q))$ time. Within this time-bound, we can also output an error if we have problems because the number of inversions is bigger than qn. However, in adaptive sorting, no-one tells us the number of inversions. On the other hand, suppose we can generate a sequence of q_i with $q_0 = 1$, and such that $S(n, q_i) = \Theta(n2^i)$. Then we can run our algorithm for up to nq_i inversions for $i = 0, 1, \ldots$ until we successfully produce a sorted list. Even if the number qn of inversions is unknown, we get a running time of

$$\sum_{i=0,1,\ldots,q_{i-1}<q} O(S(n, q_i)) = O(S(n, q))$$

For example, with the function $S(n, q) = O(n(\log \log n)/(1 + \log \log n - \log \log q))$ in Theorem 2, we can take $q_i = 2^{\log n^{1-1/2^i}}$. The time we have to compute q_i is $O(S(n, q_i))$, which is far more than we need.

Very generally, when we have a function like $S(n, \cdot)$ where can find a sequence of arguments that asymptotically doubles the value, we call the function *doubleable*. Here, by finding the sequence, we mean that we can construct the relevant parts of it without affecting our overall time and space consumption.

3 Equivalence between inversion sorting and multisorting

In this section we show Theorem 1.

Lemma 1. *For any integer $r \geq 1$ we can reduce, with an additive $O(n)$ overhead in time, inversion sorting of n integer keys with at most qn inversions to performing the following tasks:*

(i) Multisorting n integer keys in groups with at most r integer keys in each, followed by

(ii) Inversion sorting $4qn/r$ integer keys with an upper bound qn on the number of inversions

Proof. The following algorithm will solve the problem as described in the lemma.

1. Divide the n keys into $\lceil n/r \rceil$ groups with r keys in each group, except possibly the last. Denote the groups $G_1, \cdots, G_{\lceil n/r \rceil}$. For ease of exposition let $G_{\lceil n/r \rceil + 1}$ be an empty group.
2. Multisort the n keys in the groups of size at most r, and let $G_i[1], G_i[2], \ldots$ be the sorted order of the keys in group i.
3. Initialize $\lceil n/r \rceil + 1$ empty lists denoted S_1, and $U_1, \ldots, U_{\lceil n/r \rceil}$.
4. Merge all the groups by looking at two groups at a time. For $i = 1, \ldots, \lceil n/r \rceil$ add the keys in groups G_i and G_{i+1}, to S_1, U_i, or U_{i+1} one at a time as described below. We say that a key in a group G_i is *unused* if it has not been added to any of the lists S_1 or U_i.

While there are any unused keys in G_i and no more than $r/2$ used keys in G_{i+1} we repeatedly find the minimum unused key. In case of a tie the key from G_i is considered the minimum. The minimum is inserted in S_1 if the previous key inserted in S_1 is not larger, and otherwise it is inserted in U_i or U_{i+1} depending on the group it came from. Any remaining keys in G_i after this are inserted in U_i.

5. Let U be the concatenation of the lists $U_1, \ldots, U_{\lceil n/r \rceil}$.
6. Inversion sort the list U with qn as an upper bound on the number of inversions. Denote the resulting sorted list S_2.
7. Merge the two sorted lists S_1 and S_2.

It is easy to see that the two merging stages, in steps 4 and 7 of the algorithm, take linear time and that the multisorting in step 2 is as stated in the lemma. Since the ordering of the keys in U is the same as in the original sequence the number of inversions in U is bounded by qn.

What remains to show is that the list U has size at most $4qn/r$. We argue that in all cases in step 4 where a key is added to the list U_i or U_{i+1}, this key is involved in at least $r/2$ inversions. First of all, the minimum key can only be smaller than the last key inserted in S_1 when we have just increased i, and it must come from G_{i+1}. But then it must have at least $r/2$ inversions with unused keys in G_i. Secondly, whenever we put remaining keys from G_i in U_i, each of these has at least $r/2$ inversions with used keys from G_{i+1}. Since each key in U is involved in at least $r/2$ inversions and there are at most qn inversions in total, and since each inversion involves two keys, we have an upper bound on the size of U of $4qn/r$.

Corollary 2. *On a RAM, inversion sorting of n integer keys, given an upper bound qn on the number of inversions, can be reduced to multisorting n integer keys in groups of size at most q, using $O(n)$ time and space.*

Proof. Using Lemma 1 with $r = q \log n$ we get that the time for inversion sorting is $O(n)$ plus the time for multisorting n keys in groups of size at most $q \log n$ and inversion sorting $4n/\log n$ keys with at most qn inversions. Sorting $4n/\log n$ keys can be done in time $O(n)$ by any optimal comparison-based sorting algorithm.

Multisorting n keys in groups of size at most $q \log n$ can be done by multisorting the n keys in groups of size at most q and merging the groups into larger ones in $O(n)$ time. A group of size at most $q \log n$ is divided up into at most $\log n$ subgroups of size at most q. To merge these sorted subgroups into one sorted group, start by inserting the minimum from each subgroup into an atomic heap [7]. Repeatedly remove the minimum from the heap and let it be the next key in the merged group. If the deleted minimum comes from subgroup i, then insert the next key, in sorted order, from subgroup i into the atomic heap. Insertion, deletion, and findmin in an atomic heap with at most $\log n$ keys takes $O(1)$ time, after $O(n)$ preprocessing time [7]. The same atomic heap can be used for all mergings and the total time to merge all groups is hence $O(n)$.

Lemma 2. *On a RAM with word size w, multisorting n w-bit integer keys in groups of size at most q, can be done by inversion sorting n w-bit integer keys with an upper bound of qn inversions plus $O(n)$ extra time and space.*

Proof. The following algorithm will solve the problem as described in the lemma. Assume that the word length w is at least $2\lceil \log n \rceil$. If it is smaller then the sorting can be done in linear time using radix sort.

1. Sort all the keys with respect to the $2\lceil \log n \rceil$ least significant bits, using radix sort. Keep a reference to the original position as extra information.
2. Distribute the keys in the original groups, using the references, such that each group now is sorted according to the $2\lceil \log n \rceil$ least significant bits.
3. For each key x, construct a new w-bit word d_x, where the first $\lceil \log n \rceil$ bits contain the number of the group that the key comes from. The following $w - 2\lceil \log n \rceil$ bits contain the $w - 2\lceil \log n \rceil$ most significant bits in x, and the last $\lceil \log n \rceil$ bits contain x's rank after the sorting and re-distribution in steps 1 and 2.
4. Inversion sort the constructed words from step 3 with qn as an upper bound on the number of inversions.
5. Construct the multisorted list by scanning the list from step 4 and using the rank (the last $\lceil \log n \rceil$ bits) to find the original key.

Steps 1, 2, 3 and 5 can clearly be implemented to run in $O(n)$ time, and provided that qn is an upper bound on the number of inversions for the sorting in step 4 the time for the algorithm is as stated in the lemma. To see that qn is an upper bound on the number of inversions, note that since the most significant bits in the words sorted in step 4 encode the group number, there can be no inversions between words in different groups. Since a group contains at most q keys there are at most q inversions per word, and qn is an upper bound on the total number of inversions.

What remains to show is that the algorithm is correct. For two keys x and y in the same group and $x < y$, it holds that the words d_x and d_y constructed in step 3 have the property $d_x < d_y$. This together with the fact that the group number is written in the most significant bits in the words sorted in step 4 gives that the list produced by the last step is the input list multisorted as desired.

Theorem 1 follows from Corollary 2 and Lemma 2.

4 Fast multisorting on a RAM

Multisorting can of course be done by sorting one group at a time. In particular, by [9, 10] multisorting can be done in linear space and $O(n\sqrt{\log \log q})$ expected time or deterministic time $O(n \log \log q)$.

If $\log q \leq w^{1/2-\varepsilon}$ for some $\varepsilon > 0$, we can use the linear expected time sorting of Andersson et al. [2] on each group. Since $w \geq \log n$, this gives linear expected time sorting for $q \leq 2^{(\log n)^{1/2-\varepsilon}}$. In this section we show how to exploit the

situation of sorting multiple groups to increase the group size to $2^{(\log n)^{1-\varepsilon}}$, which is much closer to n. This gives linear time inversion sorting algorithms for a wide range of parameters.

For our fast randomized sorting algorithm we use the following slight variant from [10] of the signature sorting of Andersson et al. [2].

Lemma 3. *With an expected linear-time additive overhead, signature sorting with parameter r reduces the problem of sorting q integer keys of ℓ bits to*

(i) the problem of sorting q reduced integer keys of $4r \log q$ bits each and
(ii) the problem of sorting q integer fields of ℓ/r bits each.

Here (i) has to be solved before (ii).

Employing ideas from [10], we use the above lemma to prove

Lemma 4. *Consider multisorting n integer keys in groups of size at most q. In linear expected time and linear space, we can reduce the length of keys by a factor $(\log n)/(\log q)$.*

Proof. To each of the groups S_i of size at most q, we apply the signature sort from Lemma 3 with $r = (\log n)/(\log q)$. Then the reduced keys from (i) have $4r \log q = O(\log n)$ bits.

We are going to sort the reduced keys from all the groups together, but prefixing keys from S_i with i. The prefixed keys still have $O(\log n)$ bits, so we can radix sort them in linear time. From the resulting sorted list we can trivially extract the sorted sublist for each S_i.

We have now spent linear expected time on reducing the problem to dealing with the fields from Lemma 3 (ii) and their length is only a fraction $1/r = (\log q)/(\log n)$ of those in the input.

We are now ready to show Theorem 2.

Proof of Theorem 2: We want to multisort n keys in groups of size at most q, where each key has length w. We are going to reduce the length of keys using Lemma 4. Using a result of Albers and Hagerup [1], we can sort each group in linear time, when the length of keys is reduced to $w/(\log q \log \log q)$. Consequently, the number of times we need to apply Lemma 4 is $O(\log_{(\log n)/(\log q)}(\log q \log \log q)) = O((\log \log q)/(\log \log n - \log \log q))$. Since each application takes linear expected time, this gives an expected sorting bound of $O(n\lceil (\log \log q)/(\log \log n - \log \log q)\rceil)$. This expression simplifies to the one stated in the theorem. For $q = 2^{(\log n)^{1-\varepsilon}}$ the time for the algorithm is $O(n/\varepsilon)$. □

Comparing the expression of Theorem 2 with the $O(n\sqrt{\log \log q})$ bound obtained from [10] on each group, we see that Theorem 2 is better if $(\log \log n - \log \log q) = \omega(\sqrt{\log \log q})$, that is, if $q = 2^{(\log n)^{1-\omega(1/\sqrt{\log \log n})}}$.

5 More general measures of disorder on a RAM

This section aims at obtaining a RAM sorting algorithm that performs well both in terms of the number of inversions and the number of runs. This is achieved by bounding the time complexity in terms of the more general Osc disorder measure of Levcopoulos and Petersson [11].

For two integers a, b and a multiset of keys S define $osc(a, b, S) = |\{x \in S \mid \min(a, b) < x < \max(a, b)\}|$. For a sequence of keys $X = x_1, \ldots, x_n$ define

$$Osc(X) = \sum_{i=1}^{n-1} osc(x_i, x_{i+1}, X) \ .$$

Clearly, $Osc(X) \leq n^2$. From [11], we know that $Osc(X)$ is at most 4 times larger than the number of inversions in X and at most $2n$ times larger than the number of runs. Hence, an algorithm that is efficient in terms of $Osc(X)$ is simultaneously efficient in terms of both the number of inversions and the number of runs.

In comparison-based models the best possible time complexity in terms of $Osc(X)$ is $O(n \log(Osc(X)/n))$. In this section we show that we can replace the logarithms by the time complexity $t(n)$ of a priority queue, which in turn is the per key cost of non-adaptive sorting of up to n keys [14]. More precisely,

Theorem 3. *Assume that we can sort up to q integer keys in linear space and time $O(q \cdot t(q))$ where t is double-able (c.f. §2). Then we can sort a sequence X of n keys in time*

$$O\left(n + \sum_{i=1}^{n-1} t(osc(x_i, x_{i+1}, X))\right) \ .$$

In particular, if t is convex, this is $O(n \cdot t(Osc(X)/n))$.

The rest of this section proves Theorem 3. First we will show how to implement a priority queue with a certain property, and then show how to use this for Osc adaptive sorting.

5.1 A last-in-fast-out priority queue

In this section we describe a priority queue with the property that it is faster to delete keys that were recently inserted into the priority queue than to delete keys that have been in the priority queue for long. The time to delete a key x will depend on the number of inserts after key x was inserted. We call this property of a priority queue, which was previously achieved for the cache-oblivious model in [5], *last-in-fast-out*.

We define a Delete-Insert(x, y) operation on a priority queue as a delete operation on key x followed by an insertion of key y, where there may be any number of operations in between. For a key x in a priority queue we denote by $inserts(x)$ the number of keys inserted in the priority queue after x is inserted and before x is deleted from the priority queue.

Theorem 4. *Assume that we can sort up to n integer keys in linear space and time $O(n \cdot t(n))$ where t is double-able (c.f. §2). Then there exists a last-in-fast-out priority queue, with capacity for holding at most n keys at any time, supporting the operations* Init() *in* $O(1)$ *time,* FindMin() *in time* $O(1)$, Insert(x) *in time* $O(t(n))$, *and* Delete-Insert(x, y) *in amortized time* $O(t(inserts(x)))$, *using linear space.*

Proof. According to the reduction in [14] the sorting bound assumed in the theorem means that there is a linear space priority queue supporting FindMin() in time $O(1)$, Delete(x) in time $O(t(n))$, and Insert(x) in time $O(t(n))$, where n is the number of keys in the priority queue.

Our data structure consists of a number of priority queues, denoted by $PQ_1, PQ_2, \ldots$ with properties as described above. We let n_i denote an upper bound on the size of PQ_i, where n_i satisfies $\lceil \log(t(n_i)) \rceil = i$. Since $t(n)$ is double-able we can compute n_i in time and space $O(n_i)$. The keys in each priority queue PQ_i are also stored in a list, L_i, ordered according to the time when the keys were inserted into PQ_i, i.e., the first key in the list is the most recently inserted key and the last is the first inserted key. There are links between the keys in PQ_i and L_i. Each key in the data structure is stored in one of the priority queues, and has a reference to its host. The minimum from each of the priority queues is stored in an atomic heap together with a reference to the priority queue it comes from. The atomic heap supports operations Insert(x), Delete(x), FindMin(), and Find(x) in constant time as long as the number of priority queues is limited to $\log n$ [7]. There will never be a need for an atomic heap of size greater than $\log n$. This is due to the definition of n_i and because, as we will see in the description of Insert(x), a new priority queue will only be created if all smaller priority queues are full.

Init() is performed by creating an empty priority queue PQ_1 and an empty atomic heap prepared for a constant number of keys. This can clearly be done in $O(1)$ time. Note that as the data structure grows we will need a larger atomic heap. Initializing an atomic heap for $\log n$ keys takes $O(n)$ time and space, which will not be too expensive, and it is done by standard techniques for increasing the size of a data structure.

FindMin() is implemented by a FindMin() operation in the atomic heap, returning a reference to the priority queue from which the minimum comes, followed by a FindMin() in that priority queue, returning a reference to the minimum key. FindMin() can be performed in $O(1)$ time.

Delete-Insert(x, y) is simply implemented as a Delete(x) followed by a Insert(y), hence we do not care about the pairing in the algorithm, only in the analysis.

Delete(x) is implemented as a Delete(x) in the priority queue, PQ_i, in which the key resides, and a Find(x) in the atomic heap. If the key is in the atomic heap, then it means that x is the minimum in PQ_i. In this case x is deleted from the atomic heap. The new minimum in PQ_i (if any) is found by a FindMin() in PQ_i and this key is inserted into the atomic heap. The time for the operations on

the atomic heap is constant. $\mathsf{Delete}(x)$ and $\mathsf{FindMin}()$ in PQ_i takes time $O(t(n_i))$ and $O(1)$ respectively. We will later argue that $t(n_i) = O(t(inserts(x)))$.

When an $\mathsf{Insert}(x_1)$ operation is performed the key x_1 is inserted into PQ_1 and into L_1 as the first key. This may however make PQ_1 overfull, i.e., the number of keys in PQ_1 exceeds n_1. If this is the case, the last key in L_1, denoted x_2, is removed from PQ_1 and inserted into PQ_2. If the insertion of x_1 or the deletion of x_2 changes the minimum in PQ_1 then the old minimum is deleted from the atomic heap and the new minimum in PQ_1 is inserted into it. The procedure is repeated for $i = 2, 3, \ldots$ where x_i is removed from PQ_{i-1} and inserted into PQ_i. We choose x_i as the last key in L_{i-1}. This is repeated until PQ_i is not full or there are no more priority queues. In the latter case a new priority queue is created where the key can be inserted. Assume that an insertion ends in PQ_i. The time for the deletions and insertions in $PQ_1, \ldots, PQ_i$ is $O(\sum_{j=1}^{i} t(n_j))$. Since the sizes of the priority queues grows in such a way that the time for an insertion roughly doubles from PQ_j to PQ_{j+1} this sum is bounded by $O(t(n_i))$. All other operations during an insert takes less time. In the worst case we do not find a priority queue that is not full until we reach the last one. Hence, the time for insert is $O(t(n))$.

Now it is easy to see that if key x is stored in PQ_i then there must have been at least n_{i-1} inserts after x was inserted. The bound for the delete part of $\mathsf{Delete\text{-}Insert}(x, y)$ follows since $t(n_{i-1}) = \Theta(t(n_i))$.

To show that the insert part of a $\mathsf{Delete\text{-}Insert}(x, y)$ operation only increases the time by a constant factor we use a potential argument. The delete operation will pay for the insert. For the sake of argument say that whenever a key x is deleted from PQ_i the delete operation pays the double amount of time compared to what it uses and leaves $O(t(n_i))$ time for the insertion of a key in its place. Whenever a key y is inserted into the data structure, the first non-full priority queue PQ_i is the place where the insertion stops. This insertion is paid for by the saved potential, unless the insertion does not fill the place of a deleted key. This only happens so many times as the number of $\mathsf{Insert}(x)$ operations not coupled with a deletion. Hence, the total extra time is $O(t(n))$ for each such insertion. The amortized bound on $\mathsf{Delete\text{-}Insert}(x, y)$ follows.

The space usage of the data structure is clearly bounded by $O(n)$.

5.2 Our general adaptive sorting algorithm

Theorem 5. *Suppose that we have a linear space last-in-fast-out priority queue for integer keys, with amortized time complexity $O(t(n))$ for insertion and $O(t(inserts(x)))$ for deleting the minimum x followed by an insertion. Also, suppose t is non-decreasing and double-able, and that $t(n) = O(\log n)$. Then there is an integer sorting algorithm that uses linear space and on input $X = x_1, \ldots, x_n$ uses time*

$$O\left(n + \sum_{i=1}^{n-1} t(osc(x_i, x_{i+1}, X))\right) \ .$$

In particular, if t is convex, this is $O(n \cdot t(Osc(X)/n))$.

Proof. Initially we split the input into $O(n/\log n)$ groups of $\Theta(\log n)$ keys, and sort each group in linear time using an atomic heap. Call the resulting list X'. By convexity of t, and since $t(n) = O(\log n)$, the sum $\sum_{i=1}^{n-1} t(osc(x_i', x_{i+1}', X'))$ is larger than the sum in the theorem by at most $O(n)$. To sort X' we first put the minimum of each group into the last-in-fast-out priority queue in $O(n)$ time. We then repeatedly remove the minimum from the priority queue, and insert the next key from the same group, if any. The total time for reporting the first key from each group is clearly $O(n)$. Insertion of a key x_i' that is not first in a group is amortized for free and the deletion cost is $O(t(inserts(x_i'))) = O(t(osc(x_{i-1}', x_i', X')))$. Summing over all keys we get the desired sum.

Theorems 4 and 5 together imply Theorem 3. As an application, using the $O(n\sqrt{\log\log n})$ expected time sorting from [10], we get that a sequence X of n keys can be sorted in expected time $O(n\sqrt{\log\log(Osc(X)/n)})$.

6 Final remarks

An anonymous reviewer has pointed out an alternative proof of Theorem 1. The main insight is that for a sequence of n keys with qn inversions, there is a comparison-based linear time algorithm that splits the keys into a sequence of groups of size at most q such that no key in a group is smaller than any key in a previous group. In other words, all that remains to sort the entire key set is to multisort the groups.

The groups can be maintained incrementally, inserting keys in the order they appear in the input. Initially we have one group consisting of the first q keys. We maintain the invariant that all groups contain between $q/2$ and q keys. For each group, the minimum key and the group size is maintained. The group of a key x is found by linear search for the largest minimum key no larger than x. The total time for these searches is $O(n)$ since each search step can be associated with at least $q/2$ inversions. If the size of the group in which x is to be inserted is q, we split the group (including x) into two groups, using a linear time median finding algorithm. The time for these splittings is constant amortized per inserted key.

References

1. S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136:25–51, 1997.
2. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comp. Syst. Sc.*, 57:74–93, 1998. Announced at STOC'95.
3. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32nd STOC*, pages 335–342, 2000.
4. P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31st STOC*, pages 295–304, 1999.
5. G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th ISAAC*, volume 2518 of *LNCS*, pages 219–228. 2002.

6. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.

7. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994.

8. L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th STOC*, pages 49–60. 1977.

9. Y. Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608, 2002.

10. Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *Proc. 43rd FOCS*, pages 135–144, 2002.

11. C. Levcopoulos and O. Petersson. Adaptive heapsort. *J. Algorithms*, 14(3):395–413, 1993.

12. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.

13. R. Sedgewick. *Quicksort*. Garland Pub. Co., New York, 1980.

14. M. Thorup. Equivalence between priority queues and sorting. In *Proc. 43rd FOCS*, pages 125–134, 2002.