**DIMACS Technical Report 2001-29**
**September 2001**

# Adaptive Properties of Pairing Heaps

by

Amr Elmasry
Computer Science Department, Rutgers University
elmasry@paul.rutgers.edu

---

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs, NEC Research Institute and Telcordia Technologies (formerly Bellcore).

DIMACS was founded as an NSF Science and Technology Center, and also receives support from the New Jersey Commission on Science and Technology.

**ABSTRACT**

We show that the pairing heap benefits from the presortedness in the input sequence, and sorts adaptively. In particular, we show that given an input sequence of size $n$ that is already sorted in ascending order from right to left, repeatedly deleting the smallest element, using the pairing heap operation, requires at most $7n$ comparisons. We also show that starting with any initial heap structure that has the property that the corresponding sequence of the heap is sorted in ascending order, at most $7n$ comparisons are required to produce the sorted sequence by repeatedly deleting the smallest element from the heap. This latter result implies an easy proof for Tarjan's sequential access theorem for splay trees, with a better bound of $3.5n$ rotations. Finally, experimental results are conducted, supporting the fact that the pairing heap is adaptive by comparing sorting using the pairing heap with Splaysort and Binomialsort.

**Key-Words:** Algorithms - Data structures - Heaps - Adaptive sorting - Splay trees.

# 1 Introduction

A sorting algorithm is considered adaptive if it benefits from the presortedness in the input sequence. In other words, an adaptive sorting algorithm sorts an already sorted sequence in linear time, and runs in $O(n \log n)$ no matter what the input sequence is. There are several algorithms that have these properties [2] [6]. One of the main measures of presortedness is the number of inversions in the input sequence [7]. The number of inversions is the number of pairs in the wrong order. More precisely, for an input sequence $X$ of length $n$, the number of inversions, $Inv(X)$, is defined

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

An adaptive sorting algorithm is optimal with respect to the number of inversions, if the time required by the algorithm to sort a sequence $X$ of length $n$ is in $O(n \log \frac{Inv(X)}{n} + n)$ [5]. There are several sorting algorithms, optimal with respect to the number of inversions. Among these algorithms is Splaysort and Binomialsort. Splaysort [8], sorting by repeated insertion in a splay tree [9], was proved to be optimal with respect to the number of inversions [1]. Binomialsort [3] is a practically efficient sorting algorithm. The total number of comparisons performed by Binomialsort is at most $2n \log \frac{Inv(X)}{n} + O(n)$.

The pairing heap [4] is a heap-ordered general tree. The values in the heap are stored one value per tree node. The value stored in the parent of a node is smaller than or equal to the value stored in the node itself. We thus find the minimum heap value stored in the tree root. There is no restriction on the number of children a node may have, and the children of a node are maintained in a list of siblings. The basic operation on a pairing heap is the pairing operation, in which two queues are combined by making the root with the larger key value the leftmost child of the other root. Among the operations that can be performed on the pairing heap, we are mainly concerned with the *deletemin* operation. The *deletemin* operation takes place by removing the node at the root and returning its value. The resulting queues are then combined together using the pairing operation to form a single queue. Several variants of this data structure have been suggested, each uses a different way of combining the resulting queues in pairs. In the standard two-pass variant of the pairing heaps, the *deletemin* operation proceeds as follows. In the first pass the children of the deleted root are paired together starting from left to right. In other words, if the number of the queues is odd, the rightmost queue is left unpaired. We refer to this pass as the halving pass. In the second pass, the remaining queues are paired, in order, from right to left. Each queue is paired with the result of the linkings of the queues to its right. We refer to this pass as the right-to-left incremental pairing pass. It has been proven [4] that the amortized cost of the *deletemin* operation for the two-pass variant is $O(\log n)$. In other words, any input of $n$ values can be sorted in $O(n \log n)$ comparisons using this standard implementation of the *deletemin* operation. Another variant of the pairing heap is called the multi-pass variant. In the multi-pass variant, the *deletemin* operation proceeds as follows. The children of the deleted root are paired together from left to right, in what we call a pairing pass. The resulting queues are paired again by another pairing pass, and the process repeats until a single queue is left. The above bound does not apply for the multi-pass variant of the pairing heaps.

The corresponding sequence of a forest of trees, each node of which contains a single value, is defined to be the sequence of values obtained by a pre-order traversal of this forest. Our traversal gives precedence to the trees of the forest in right-to-left order. Also, the precedence ordering of the sub-trees of a given node proceeds from right to left. In this paper, we show that starting with a sequence of $n$ elements sorted in ascending order from right to left and represented as $n$ single nodes, using the pairing heap for sorting takes at most $3.5n$ comparisons for the multi-pass variant and $7n$ comparisons for the two-pass variant. This suggests the fact that the pairing heap benefits from the presortedness in the input sequence. We also show that starting with any heap structure, with the property that the corresponding sequence of the heap is sorted in ascending order, it takes the two-pass variant of the pairing heaps no more than $7n$ comparisons to produce the sorted sequence.

Because of the similarity between the two-pass variant of the pairing heaps and splay trees [9], the above fact directly implies an easy proof for Tarjan's sequential access theorem for splay trees, bounding the number of rotations with at most $3.5n$. The sequential access theorem states that the number of rotations required to access each of the $n$ nodes of an arbitrary initial splay tree once, in symmetric order, is linear. Tarjan [11] proved that the bound for the number of rotations is $10.8n$. Sundar [10] gave an easier proof that uses a potential function technique. His bound for the number of rotations is $15n$.

Experimental results are conducted to measure the performance of the two-pass variant of the pairing heaps, when used for sorting. The performance of the pairing heap is compared with both Splaysort and Binomialsort, indicating that the pairing heap sort is practically efficient as an adaptive sorting algorithm.

We leave it as an open question, whether the pairing heap is optimal with respect to the number of inversions, or with respect to other known measures of presortedness defined in the literature.

## 2 Adaptivity

**Lemma 1** *Given an input sequence of $n$ elements that are sorted in ascending order from right to left, represented as $n$ single nodes. Applying the multi-pass variant of the pairing heaps' deletemin operation repeatedly, until the heap is empty, requires less than* $3.5n$ *comparisons.*

**Proof** Throughout the algorithm, when the value of a node is compared with its right neighbor, the one to the left will have the larger value. Hence, the left node will be linked to the right node as its leftmost child.

After the first multi-pass phase, which requires $n-1$ comparisons, the heap becomes a binomial queue. We call the queue at this moment the initial binomial queue. Recall that a binomial queue is a heap structure that has the following property. A binomial queue of rank $r$ is constructed recursively by making the root of a binomial queue, of rank $r-1$, the leftmost child of another binomial queue of rank $r-1$. A binomial queue of rank 0 consists of a single node. The rank of any node in a binomial queue equals the number of the children of this node. There are $\frac{n}{2^{i+1}}$ nodes of rank $i$ in an $n$ node binomial queue. In the above definitions we assume for simplicity than $n$ is a power of 2. The results hold for any general value of $n$.

The left spine of a node is defined to be the path from that node to the leftmost leaf of the sub-tree defined by this node. In other words, every node on the path is the leftmost child of its predecessor. Consider any node, $x$, in the heap. Let $g_x(t)$ be the length of the left spine of $x$ after the pairing pass $t$. Let $h_x(t)$ be the length of the left spine of the right neighbor of $x$ after the same pairing pass. If $x$ does not have a right neighbor, then $h_x(t)$ is equal to 0. We show by induction on time that for any node, $x$, $g_x(t) - h_x(t)$ is a positive non-decreasing function with time. Let $t_0$ be the number of the last pairing pass of the first multi-pass phase. A property of binomial queues implies that the value of $g_x(t_0) - h_x(t_0)$ is positive. This establishes the base case. Consider any node $w$ and its right neighbor $z$ after the pairing pass $t$, where $t \geq t_0$, such that $w$ is linked to $z$ during the pairing pass $t+1$. The following relations follow:

$$h_w(t) = g_z(t) \tag{1}$$

$$g_w(t+1) = g_w(t) \tag{2}$$

$$h_w(t+1) = h_w(t) - 1 \tag{3}$$

$$g_z(t+1) = g_w(t) + 1 \tag{4}$$

$$h_z(t+1) \leq h_z(t) + 1 \tag{5}$$

Using (2) and (3), then $g_w(t+1) - h_w(t+1) > g_w(t) - h_w(t)$, and the hypothesis is true for the node $w$ at time $t+1$. Using the induction hypothesis for node $w$ at time $t$, then $g_w(t) > h_w(t)$. This relation together with (1) and (4) implies that $g_z(t+1) > g_z(t) + 1$. Using the latter relation and (5), then $g_z(t+1) - h_z(t+1) > g_z(t) - h_z(t)$, and the hypothesis is true for node $z$ at time $t+1$. The hypothesis is then true for all nodes throughout the algorithm.

We distinguish between two types of links. If $g_z(t) - h_z(t) = 1$, we call the link an A-link. If $g_z(t) - h_z(t) \geq 2$, we call the link a B-link. The above analysis indicates that any node may gain at most one child by an A-link. Hence, the number of comparisons accompanying A-links is at most $n$.

We use the accounting method for bounding the number of comparisons accompanying B-links. After the first multi pass phase, we maintain the invariant that the number of credits on a node $x$, after any pairing pass $t$, is $\frac{h_x^2(t)}{2}$. In the initial binomial queue, $h_x(t_0)$ equals the rank of $x$, $r_x$. Let $C$ be the total number of credits needed to keep the invariant hold for the initial binomial queue. Then

$$\begin{aligned} C &= \sum_x \frac{r_x^2}{2} \\ &= \sum_{i=1}^{\log n - 1} \frac{n}{2^{i+1}} \frac{i^2}{2} \\ &< 1.5n \end{aligned}$$

Next, we show that these credits are enough to pay for all the B-links, while maintaining the invariant. Let $d$ be the difference between the sum of the number of credits on $w$ and $z$ before pass $t+1$ and those needed after pass $t+1$. Then

$$d = \frac{h_z^2(t)}{2} + \frac{h_w^2(t)}{2} - \frac{h_z^2(t+1)}{2} - \frac{h_w^2(t+1)}{2}$$

Using (3) and (5), then

$$d \geq h_w(t) - h_z(t) - 1$$

Using (1) together with the fact that, for all the B-links, $g_z(t) - h_z(t) \geq 2$, then $d \geq 1$. This extra credit is used to pay for the comparison between $w$ and $z$.

The lemma follows by adding the $n-1$ comparisons done in the first multi-pass pairing phase, the, at most, $n$ comparisons accompanying the A-links, and the, at most, $1.5n$ comparisons accompanying the B-links. $\square$

**Lemma 2** *Given an input sequence of $n$ elements that are sorted in ascending order from right to left, represented as $n$ single nodes. Applying the two-pass variant of the pairing heaps' deletemin operation repeatedly, until the heap is empty, requires less than $7n$ comparisons.*

**Proof** The arguments in Lemma 1 apply for all the comparisons of the halving passes of the *deletemin* operation. This leads to a total of at most $3.5n$ comparisons, for the halving passes. The number of comparisons of the right-to-left incremental pairing passes is less than or equal to the number of comparisons of the halving passes. Therefore, the total number of comparisons is less than $7n$. $\square$

The two-pass variant of the pairing heaps also performs well if the input sequence is inversely sorted, as illustrated by the following lemma.

**Lemma 3** *Given an input sequence of $n$ elements that are sorted in descending order from right to left, represented as $n$ single nodes. Applying the two-pass variant of the pairing heaps' deletemin operation repeatedly, until the heap is empty, requires less than $\frac{3}{2}n$ comparisons.*

**Proof** After the first halving and right-to-left incremental pairing passes, $\lceil \frac{n}{2} \rceil$ of the nodes will form a chain that represents the right spine of the heap. Each of these nodes will have a single node left child (except possibly for the last node on the path). These two passes involve $n-1$ comparisons. From this point on, only one comparison is performed for every two *deletemin* operations. At most, an extra $\frac{n}{2}$ comparisons are needed to produce the sorted sequence, and the lemma follows. $\square$

**Lemma 4** *Given any initial heap structure of $n$ elements, having the property that the corresponding sequence of the heap is sorted in ascending order. Applying the two-pass variant of the pairing heaps' deletemin operation repeatedly, until the heap is empty, requires less than $7n$ comparisons.*

**Proof** Throughout the algorithm, when a value of a node is compared with another, the one to the left will have the larger value. Hence, the left node will be linked to the right one as its leftmost child.

We start bounding the number of comparisons of the halving passes, and use the fact that the number of comparisons of the right-to-left incremental pairing passes is less than or equal to the number of comparisons of the halving passes. For the purpose of the proof, we will use a coloring scheme to distinguish some nodes from others. The following rules for coloring nodes are only applied on the halving passes:

- Initially, all the nodes are uncolored.
- When an uncolored node becomes the root of a tree, it is colored yellow.
- When a yellow node is linked to another node, this yellow node is colored red.
- When a red node becomes the root of a tree, this node and all its red descendents are colored green.
- When a green node is linked to a yellow node, the yellow node is colored green.

Consider any green node $x$. Let $g_x(t)$ be the number of the green nodes on the left spine of $x$ after the pairing pass $t$. Let $h_x(t)$ be the number of green nodes on the left spine of the right neighbor of $x$ after the same pairing pass. If $x$ does not have a right neighbor, then $h_x(t)$ is equal to 0. The following properties hold because of the way the linking takes place, and because of our coloring rules:

1. Every node is colored yellow exactly once.
2. The roots of the trees are either yellow or green.
3. The father of a green node is also green.
4. For any green node $x$, the value of $g_x(t) - h_x(t)$ is positive and is non-decreasing with time.
5. Any node is colored green at most once.

When a node is colored yellow, it is given one credit. Using property 1, the total number of such credits is $n$. Using property 2, there are four possible types of links: yellow-to-yellow, yellow-to-green, green-to-yellow and green-to-green. For the first three types, the color of a yellow node changes. We use the credit on this node to pay for the link. For bounding the number of the green-to-green links, we apply a technique similar to that of Lemma 1. If a green node is linked to another green node, $x$, while $g_x(t) - h_x(t) = 1$, we call this link a green-to-green A-link. As in Lemma 1, there are at most $n$ green-to-green A-links. If $g_x(t) - h_x(t) \geq 2$, the link is called a green-to-green B-link. We keep the invariant that, after the pairing pass $t$, there are $\frac{h_x^2(t)}{2}$ credits on any green node $x$. As in Lemma 1, if the invariant holds after the pairing pass $t$, the credits on the green nodes will be enough to pay for the green-to-green B-links and to keep the invariant hold after the next pairing pass. To keep the invariant hold, when a red node becomes a root of a tree and together with its red descendents are colored green, some credits are given to these nodes. A credit of $\frac{g_x^2(t)}{2}$ is given to every node, $x$, of this sub-tree. Because the invariant holds for all the other green nodes, these extra credits are enough to keep it hold for all the green nodes. Using properties 3 and 4, this red sub-tree has similar properties to binomial queues. Let $C$ be the total number of all such credits. Using property 5, the total number of green nodes given such credits is at most $n$. Therefore, $C$ is bounded by the credits given to one, $n$ node, binomial queue. This follows from the fact that several binomial queues, with a total of $n$ nodes, can be embedded in a single $n$ node binomial queue. Then

$$\begin{aligned} C &\leq \sum_{i=1}^{\log n - 1} \frac{n}{2^{i+1}} \frac{i^2}{2} \\ &< 1.5n \end{aligned}$$

The number of comparisons involved in all the halving passes is, therefore, at most $3.5n$. The number of comparisons of the right-to-left incremental pairing passes is less than or equal to the number of comparisons of the halving passes. Therefore, the total number of comparisons is at most $7n$. □

Consider the behavior of the two-pass variant of the pairing heaps as in Lemma 4, such that whenever a couple of trees are paired, the root of the left tree is linked to the root of the right tree, and consider the splaying operation applied repeatedly on the leaf on the left spine of the slay tree. There is a great similarity between the behavior of the two structures. It can be shown, with a similar proof to that of Lemma 4, that the number of rotations of the splay trees is at most the number of comparisons performed during the halving passes of the two-pass variant of the pairing heaps.

**Corollary 1** *If the nodes of an arbitrary initial splay tree are accessed, in symmetric order, the total number of rotations involved is at most* $3.5n$.

# 3 Experimental Findings

In our experiments we compare the results of sorting an input sequence that has some presortedness, when applied to the two-pass variant of the pairing heaps, Splaysort and Binomialsort. We are more concerned with the number of comparisons used by the algorithms, rather than the running time.

The input sequence is randomly generated such that the expected and worst case number of inversions is controlled. To generate a sequence of $n$ elements, we start with a sorted sequence and perform two phases of permutations. For a given value of a parameter $k$, we want to permute the sorted sequence to have at most $kn$ inversions. In the first phase, the sorted sequence is broken into consecutive blocks of $\frac{n}{k}$ elements each. From each block we select one element at random, for a total of $k$ elements. These $k$ elements are then randomly permuted. The total number of inversions produced by this process is at most $\frac{nk}{2}$. In the second phase, the sequence is broken into consecutive blocks of $k$ elements each. The elements of each block are randomly permuted, for a total of at most another $\frac{nk}{2}$ inversions. A value of $k = 0$ means that the input sequence is sorted in ascending order. A small value of $k$, with respect to $n$, means that the input sequence is almost sorted. A value of $k$, which is as big as $n$, means that the input sequence is randomly sorted.

For different values of $k$, the number of comparisons performed by each of the three algorithms is recorded and plotted against $\log k$. The experiment is repeated 100 times for each value of $k$ and the average is used. The experiments are performed on two values of $n$, for Figure 1, $n = 1024$, and for Figure 2, $n = 65536$.

The experiments are repeated with the input sequence reversed before being fed to the program. In other words, a value of $k = 0$ would now mean that the input sequence is inversely sorted. A small value of $k$ means that the input sequence is almost inversely sorted. See Figures 3 and 4.

# 4 Concluding Remarks

We gave evidence that the pairing heap is adaptive when used for sorting. We have proven that, if the input is already sorted, the pairing heap sorts in linear time. We have shown that using the pairing heap is a good practical sorting algorithm, by comparing its performance with two optimal adaptive sorting algorithms; Splaysort and Binomialsort. The question about the optimality of the pairing heap, with respect to different measures of presortedness, has not been answered yet. We have also shown that Binomialsort is an efficient, practical adaptive sorting algorithm, and that it outperforms Splaysort for input sequences having a small number of inversions.

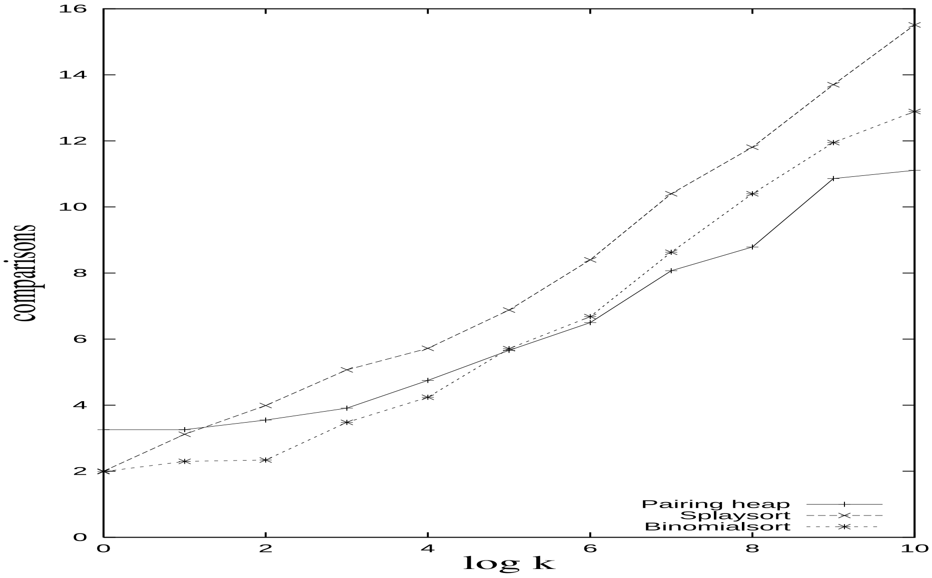


Figure 1: Comparisons per item to sort random sequences X of n=1024 items, plotted as a function of log k, with k varying from 1 to 1024 and $Inv(X) \leq kn$.

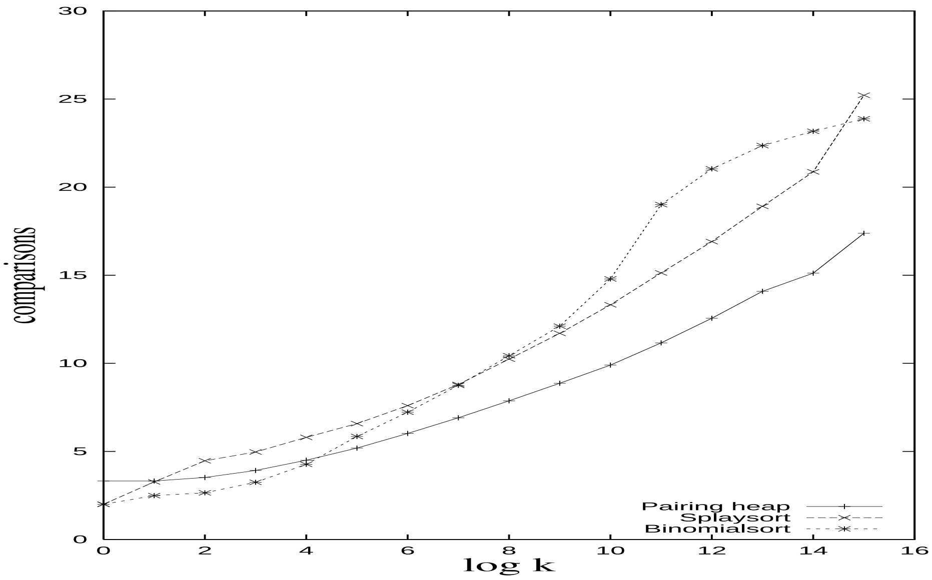


Figure 2: Comparisons per item to sort random sequences X of n=65536 items, plotted as a function of log k, with k varying from 1 to 32768 and $Inv(X) \leq kn$.

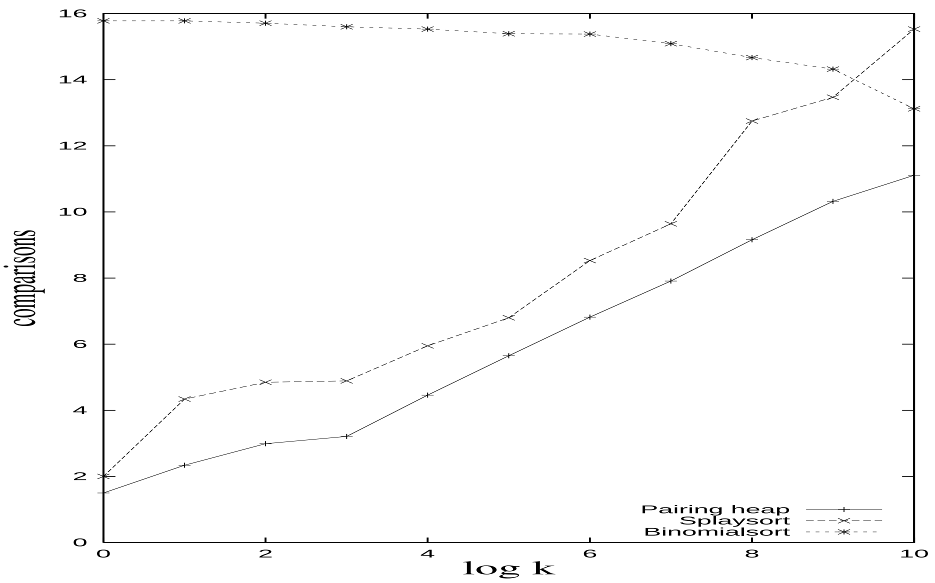


Figure 3: Comparisons per item to sort random sequences X of n=1024 items, plotted as a function of log k, with k varying from 1 to 1024 and $Inv(Rev(X)) \leq kn$, where $Rev(X)$ is the sequence produced by reversing the elements of X.

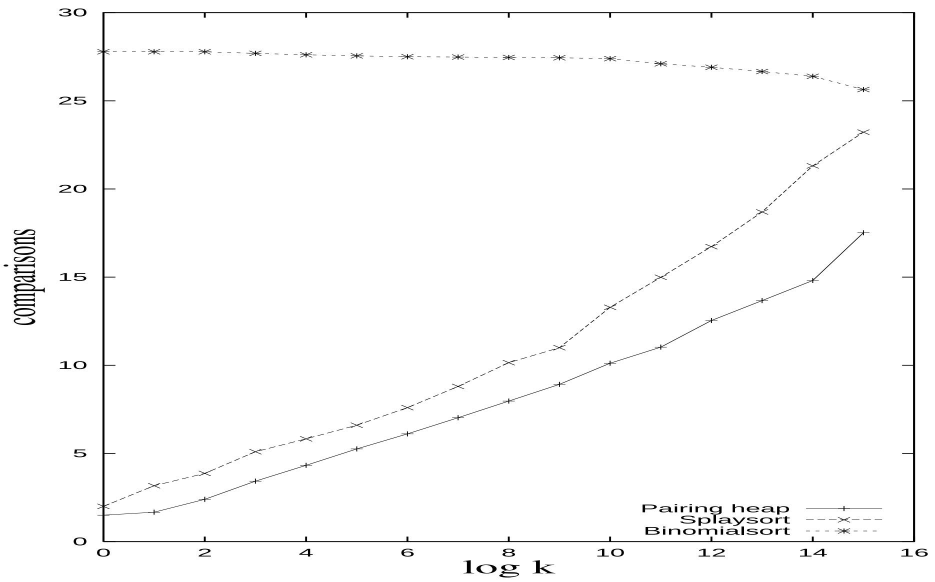


Figure 4: Comparisons per item to sort random sequences X of n=65536 items, plotted as a function of log k, with k varying from 1 to 32768 and $Inv(Rev(X)) \leq kn$, where $Rev(X)$ is the sequence produced by reversing the elements of X.

# References


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

[2] R. Cook and J. Kim. *Best sorting algorithms for nearly sorted lists.* Commun. ACM 23 (1980), 620-624.

[3] A. Elmasry. *Optimal adaptive sorting.* Submitted to SODA (2002).

[4] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. *The Pairing heap: a new form of self_adjusting heap.* Algorithmica 1,1 (1986), 111-129.

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

[6] S. Hertel. *Smoothsort's behaviour on presorted sequences.* Information Processing Letters 16 (1983), 165-170.

[7] D. Knuth. *The art of Computer programming. Vol III: Sorting and Searching.* Addison-wesley, second edition (1998).

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

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

[10] R. Sundar. *Twists, turns, cascades, deque conjecture, and scanning theorem.* FOCS (1989), 555-559.

[11] R. Tarjan, *Sequential access in splay trees takes linear time.* Combinatorica 5 (1985), 367-378.