Improving Sorting Algorithms via Information Theoretic Analysis

Qiyuan Zhao 518030910289

Yanwei Jiang 518030910274

Xuekai Liu 518030910309

Shanghai Jiao Tong University

Abstract

Entropy describes the degree of confusion of a set, while information entropy represents the amount of information that can be obtained in each step. As an ancient and important problem, the essence of sorting problem is to realize a process from disorder to order. Therefore, the use of information theory can provide a feasible reference for the improvement of sorting problem. In this paper, the original heapsort algorithm is improved, and proposes two fast heapsort algorithms. The experimental results show that the fast heapsort (rearranged) algorithm has excellent performance and strong stability in the case of large amount of data. At the same time, this paper also proposes an optimization idea for the fast sorting algorithm.

Keywords: information entropy, sorting algorithms, heapsort, quicksort.

Contents

1 Introduction

1.1 Invention and Importance of Sorting Algorithms

In our daily life, sorting is a frequently-used method. Such as sorting students by their grades, sorting goods of a supermarket by price.

Similarly, in science occasion, we usually analyse datasets that contain millions of independent items. In most cases, they need to be arranged in a certain order, and it is a challenge to sort so many data. So, many sorting algorithms are invented. The *speed* of sorting algorithm is very important. An outstanding algorithm can perform hundreds of times better than a terrible one. So when dealing with millions of data, an algorithm's speed is vital to help us to analyse, or for another program to process them. Then we would like to introduce some typical algorithms.

1.2 Typical Algorithms and Their Speed

When sorting a small amount of items in life, we usually:

1. Select two items, and sort them.
2. Then select a new item, compare it with sorted ones.
3. Place it in its position that it should be.
4. Repeat this process until all is sorted.

This naive method is called *insertion sort*. Managing cards in a card game is an example for insertion sort.

Insertion sort is a natural but not a good algorithm. In some best cases, we need to compare n times and swap 0 times. But that is fairly rare. In most cases, we need to compare and swap less than n^2 times. The time complexity is $O(n^2)$.

This is just an introduction. There are still many typical algorithms, such as *bubble sort*, *shell sort* and *selection sort*. But they can not cope with large array. We have three frequently-used excellent algorithms: *quicksort*, *merge sort* and *heapsort*. The time complexity of them is $\Omega(n \log n)$. They are fast enough to sort a large data set (more than 1 billion

on normal PC and even larger on a better computer). And we will focus on quicksort and heapsort. They are what we are interested in.

1.3 Purpose of This Report

After learning *Information Theory*, we re-researched the sorting algorithms we've learned in *Data Structure* last semester by applying information entropy.

Our purpose is:

1. Proved that $\Omega(n \log n)$ is the lower bound of sorting algorithms.
2. Improve quicksort and heapsort based on some existing work.
3. Prove or explain their advantages and compare them with the original algorithms.

2 Literature review

As an ancient and very important problem, sorting problem has been studied for a long time. Kislitsyn proved in 1968 [6] that any comparison-based algorithm requires $\log(e(P))$ comparisons in the worst case, where $e(P)$ is the number of linear extensions of P. This is called the "information theory lower bound (ITLB)", which we will also talk about later. But almost 5 years later, Fredman [4] showed that sorting can be achieved with $\log(e(P)) + 2n$ comparisons, where $n = |P|$. So it is clear that the ITLB is nearly sharp unless $e(P)$ is quite small. But both of their work only state the ideal lower bound. No one gives a feasible implementation method.

With the emergence of a variety of sorting algorithms, people began to think about how to make the time efficiency of the algorithm close to the theoretical value. Paul Hsieh [2] did a research on the running time of heapsort and quicksort in different compiling environments, which showed that the total CPU tally is different from the number of comparisons made. Then David MacKay [1] thought the reason of that heapsort is not as fast as quicksort in general is that it makes comparisons whose outcomes do not have equal prior probability. He explained this based on the idea of expected information content. Recently, a Chinese

searcher called Guoqing Qi [5] analyzed the efficiency of the insertion sort and merging sort by means of entropy, and pointed out that when the length of the series is larger than 4, any sorting algorithm based-on direct comparison requires more comparison than the lower bound. But as for heapsort and quicksort, almost none of them gave a reasonable and effective improvement method until now.

3 Preliminaries

In this section, we will model our problem. We will also give necessary definitions and lemmas.

3.1 Basic Definitions

For convenience, we will focus on the following *sorting problem*:

Input: a sequence of n *distinct* numbers $a_1, a_2, \cdots, a_n$

Output: a permutation (reordering) $b_1, b_2, \cdots, b_n$ of the input sequence where $b_1 < b_2 < \cdots < b_n$

Since the output that we want is the permutation of the input sequence, we can put it another way:

Input: a sequence of n *distinct* numbers $a_1, a_2, \cdots, a_n$

Output: a permutation of $\{1, 2, \cdots, n\}$, denoted as $\{i_1, i_2, \cdots, i_n\}$, such that $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$

We will use this definition in proving the information theoretic lower bound.

And the sorting algorithms we will discuss are *comparison-based*. That is, the sorted order they determine is based only on comparisons between the input numbers. We do a *comparison* between a_i and a_j by querying whether $a_i < a_j$ and then getting the answer.

3.2 The Information Theoretic Lower Bound

Here, we will prove the information theoretic lower bound of comparison-based sorting algorithms.

We use X^n to denote the result sequence $\{i_1, i_2, \cdots, i_n\}$. Obviously the size of its sample space is $|\Omega| = n!$.

Denote the (discrete) information entropy of X^n as $H(X^n)$. Then we have the following lemma, which shows the upper bound of $H(X^n)$.

Lemma 1. $H(X^n) \leq \log(n!)$.

Proof. The size of the sample space of X^n is $|\Omega| = n!$. Then it follows the conclusion in information theory straightforward. □

Note. We will basically focus on the cases where X^n obeys a *uniform distribution*. That is, for any possible permutation p, $\Pr(X^n = p) = \frac{1}{n!}$.

If we sorted the sequence under some information, we would uniquely determine the value of X^n, which implies that its entropy vanishes.

Lemma 2. Let the information needed to sort the input sequence be Y. Then $H(X^n|Y) = 0$.

Proof. Given Y, we can uniquely determine X^n. So $\exists! p, \Pr(X^n = p|Y) = 1$. Thus $H(X^n|Y) = 0$. □

Intuitively, in one comparison, the information that it contains is at most 1 bit, as the answer can be represented with 1 bit (0 for "no" and 0 for "yes"). So it seems reasonable to state that we need at least $\log_2(n!)$ comparisons. But actually, we can prove it in a formal way.

Lemma 3. Let Y be the information that we have obtained in previous comparisons. Then denote the answer of whether $a_i < a_j$ be $\mathrm{cmp}(a_i, a_j)$, where $1 \leq i \leq n, 1 \leq j \leq n$. Then

$$H(X^n|Y) - 1 \leq H(X^n|Y, \mathrm{cmp}(a_i, a_j)) \leq H(X^n|Y)$$

Proof. Since conditions reduce entropy, $H(X^n|Y, \text{cmp}(a_i, a_j)) \leq H(X^n|Y)$.

For another inequality, let $\Pr(a_i < a_j) = p$. Let the number of possible permutations under the condition Y be m. Then there are pm permutations where i is front of j and $(1-p)m$ permutations where i is behind j.

Since X^n obeys a uniform distribution, we have

$$\begin{aligned} H(X^n|Y, \text{cmp}(a_i, a_j)) &= pH(X^n|Y, a_i < a_j) + (1-p)H(X^n|Y, a_i > a_j) \\ &= p\log_2(pm) + (1-p)\log_2[(1-p)m] \\ &= \log_2 m - H(p) \end{aligned}$$

Since $H(p) \leq 1$ and $H(X^n|Y) = \log_2 m$, we have $H(X^n|Y) - 1 \leq H(X^n|Y, \text{cmp}(a_i, a_j))$.

□

Combining the three lemmas above, we have the following theorem.

Theorem 1. (The information theoretic lower bound) The numbers of comparisons that are needed to sort a sequence with length n is at least $\log_2(n!)$.

Note. Some textbooks also use the *decision tree model* to prove this lower bound. (Ref: Data Structure and Algorithm Analysis, 4ed)

With this lower bound, we can prove the asymptotic time complexity of an optimal comparison-based sorting algorithm. It only requires some basic algebraic transformations.

Theorem 2. The asymptotic time complexity of an optimal comparison-based sorting algorithm is $\Omega(n \log n)$.

Proof. When n is sufficiently large,

$$\begin{aligned} \log_2(n!) &= \sum_{i=1}^{n} \log_2 i \\ &\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log_2 i \\ &\geq \lceil \frac{n}{2} \rceil \log_2 \lceil \frac{n}{2} \rceil \\ &\geq \frac{n}{2} \log_2 \frac{n}{2} \\ &= \frac{n}{2} \log_2 n - \frac{n}{2} \end{aligned}$$

So $\log_2(n!) = \Omega(n \log n)$. □

4 Improving Heapsort

In this section, we implement an improvement proposed by MacKay and try to find out its advantages in a information-theoretic view.

4.1 The Original Algorithm

Heapsort can be seen as an improved selection sort. It uses an auxiliary data structure, called *heap*, to extract the minimum in the unsorted part efficiently.

A heap is essentially an array object, but we can view it as an almost complete binary tree, by taking the left and right child of element indexed i as elements indexed $2i$ and $2i+1$ respectively.

For a max(min)-heap, the *heap property* holds: for any element in the heap, it is larger(smaller) than its children. Thus the maximum(minimum) is at the root.

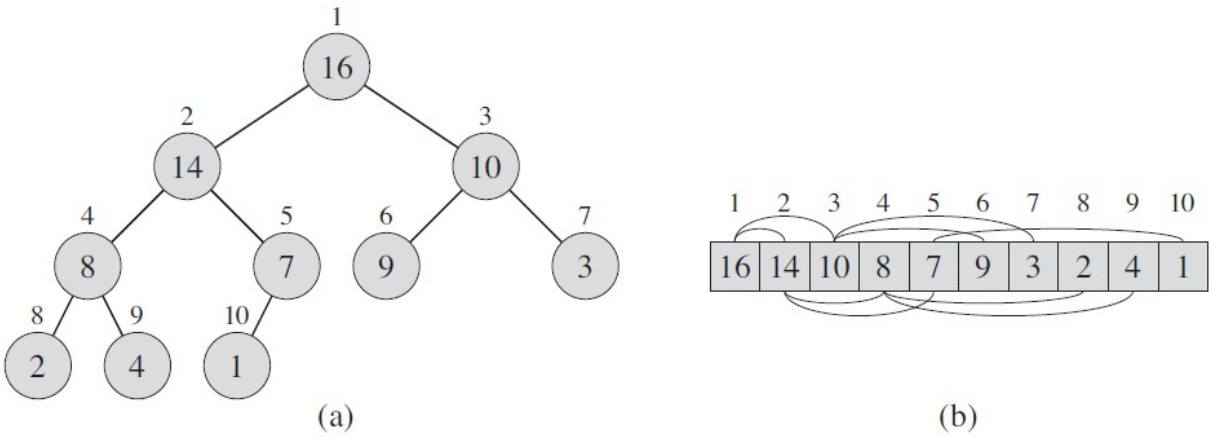

Figure 1: A max-heap viewed as (a) a binary tree and (b) an array. Picture credit: An Introduction to Algorithms, 3ed.

Heapsort consists of three main procedures: building the input sequence into a heap, popping out the root of the heap, and maintaining the heap property.

Here, we will make improvements based on Williams' method for maintaining. In Williams' method, once the root is popped out, the element with the largest index will be moved to the root, and then the maintaining procedure starts. The whole process can be

described by the pseudocode below.

Input: a sequence A

Result: sorted sequence A (in-place sort)

```
1 build_max_heap(A)
2 for i = len(A) downto 2 do
3 |   swap(A[1], A[i])
4 |   A.heap_size := A.heap_size - 1
5 |   percolate_down(A, 1)
6 end
```

Algorithm 1: Heapsort (Williams' method)

For more details, one can refer to the papers where the algorithm is proposed or some algorithm books like [3].

4.2 Analysis and Improvements

The algorithm above can achieve the information theoretic lower bound asymptotically, but the maintaining method is somehow weird. Swapping the largest-indexed element with the root seems to disrupt the orderliness.

So to make it more natural, MacKay proposed an improved version of heapsort, called

"fast heapsort". His idea can be written as follows.

Input: a heap A and a hole indexed at id

Output: the position id where maintainment finishes

Result: a maintained heap A (in-place maintainment)

```
1  if id is a leaf node then
2      delete id from the heap
3      return id
4  else
5      if id has one child then
6          let ch be the child
7      else
8          compare the two sub-heap roots directly below id
9          denote the larger one as A[ch]
10     end
11     A[id] := A[ch]
12     creating a hole at ch
13     maintain(A, ch)
14 end
```

Algorithm 2: The `maintain` Function

Input: a sequence A

Output: a sorted sequence B, also a permutation of A

```
1  build_max_heap(A)
2  for i = len(A) downto 1 do
3      B[i] := A[1]
4      remove A[1], resulting in a hole
5      maintain(A, 1)
6  end
```

Algorithm 3: Fast Heapsort

Here is an example that fast heapsort beats the original one in the sense of numbers of comparisons made. And we use it to demonstrate how fast heapsort works.

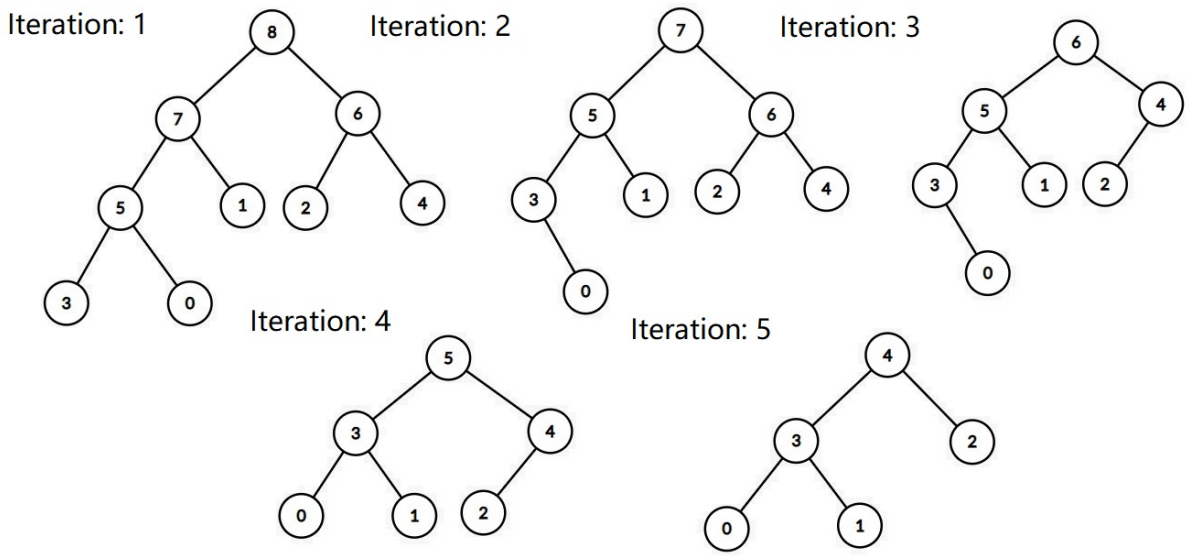

Figure 2: A example. Created with the graph editor at `csacademy.com`.

Although he did not make a more specific explanation for this method, we may understand it with the famous principle: *chaos increases entropy*. An similar example is that shuffling cards will increase the entropy of the order of cards, because it adds chaos into the order.

As we can see, in fast heapsort, the heap may not be an almost complete binary tree. To fix this, we can introduce a sifting-up operation (used in inserting an element into a heap)

at the end of one iteration. This rearranged version is described below.

```
   Input: a sequence A
   Result: a sorted sequence A (in-place)
 1 build_max_heap(A)
 2 for i = len(A) downto 2 do
 3 |   tmp := A[1]
 4 |   remove A[1], resulting in a hole
 5 |   pos := maintain(A, 1)
 6 |   if pos < i then
 7 |   |   A[pos] := A[i]
 8 |   |   sift_up(A, pos)
 9 |   end
10 |   A[i] := tmp
11 end
```

Algorithm 4: Fast Heapsort (Rearranged)

4.3 Implementation

We implemented the original and rearranged fast heapsort with C++. We also implemented the original heapsort with C++. The code is this repository.

4.4 Result

We tested our implementation with random test data. We generate input sequences with a random number generator. The code is this repository.

The result is shown in the picture as follows.

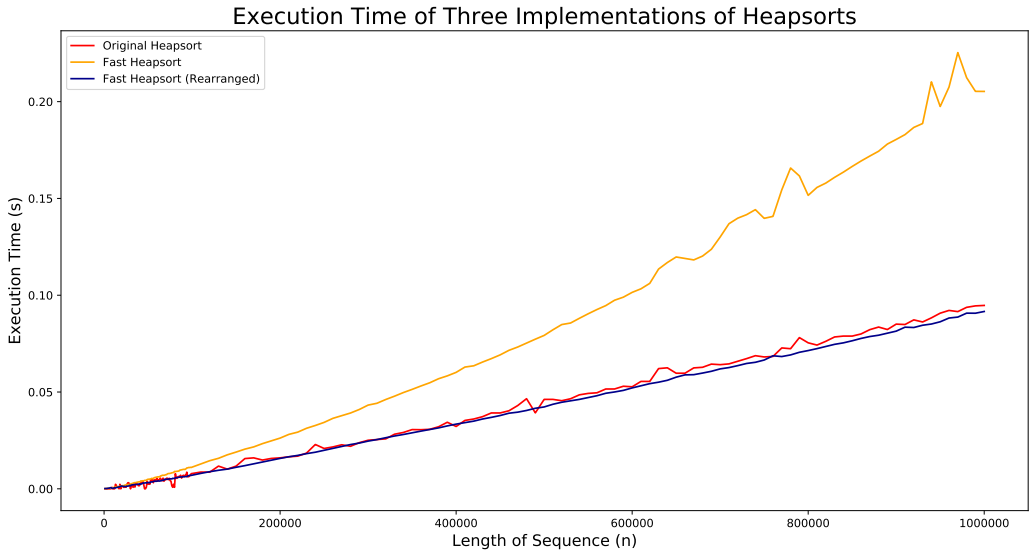

Figure 3: The performance of three implementations under normal test-cases.

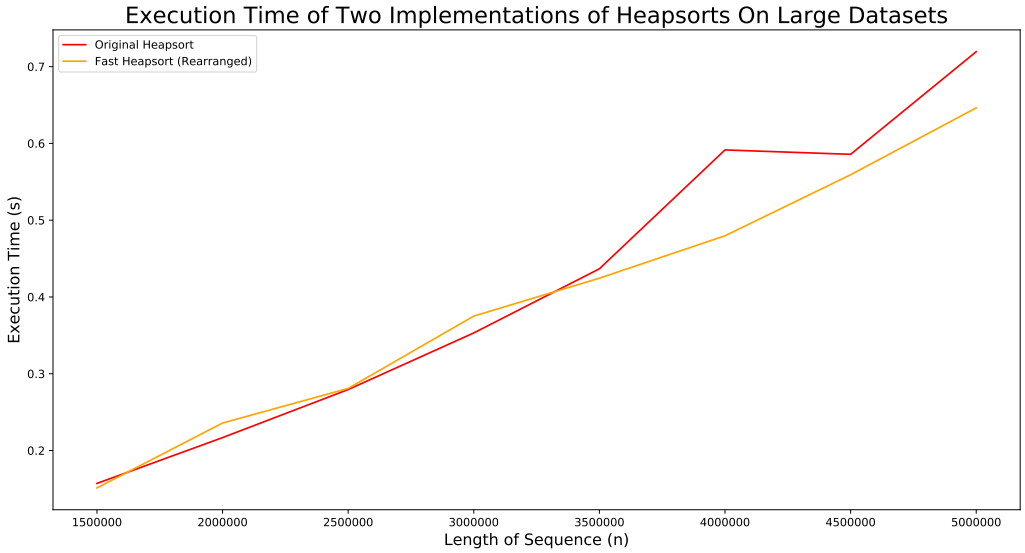

Figure 4: The performance of three implementations under large test-cases.

5 Improving Quicksort

In this section, we will briefly discuss how to improve the quicksort based on some information theoretic analysis.

5.1 The Original Algorithm

Quicksort is an in-place algorithm with expected running time $\Theta(n \log n)$. It is quite efficient for most input sequences, despite the fact that it has a worst-case running time of $\Theta(n^2)$.

Quicksort applies the classical divide-and-conquer strategy in sorting. In each recursive case, a special element called *pivot* is chosen, and then the sequence is partitioned into three parts: the pivot, the sub-sequence on its left where the elements are all smaller than it, and the sub-sequence on its right where the elements are all larger than it. Then the two sub-sequences will be recursively sorted.

The pseudocode of quicksort is described as follows.

Input: a sequence A, the left bound of sub-sequence l, and the right bound r

Output: an index mid where the pivot is located

Result: elements in the sequence with indices $[l, mid-1]$ are smaller than $A[q]$, and elements in the sequence with indices $[mid+1, r]$ are larger than $A[q]$

```
1  pivot := A[r]
2  i := p − 1
3  for j = l : r − 1 do
4      if A[j] ≤ pivot then
5          i := i + 1
6          swap(A[i], A[j])
7      end
8  end
9  swap(A[i + 1], A[r])
10 return i + 1
```

Algorithm 5: The `partition` Function

Input: a sequence A, the left bound of sub-sequence l, and the right bound r

Result: elements in the sequence with indices $[l, r]$ are sorted (in-place)

```
1 if l < r then
2     pos := partition(A, l, r)
3     quicksort(A, l, pos − 1)
4     quicksort(A, pos + 1, r)
5 end
```

Algorithm 6: The `quicksort` Function

Input: a sequence A

Result: a sorted sequence A (in-place)

```
1 quicksort(A, 1, len(A))
```

Algorithm 7: Quicksort

5.2 Analysis and Improvement

The most critical part in quicksort is the choice of pivot. A good pivot can greatly reduce the entropy, while a bad pivot can almost do nothing helpful.

As pointed out in [1], a bad pivot will make the partition unbalanced, and thus reduce the information gain in each iteration. So we can focus on how to find a good pivot.

One classical way to prevent a bad pivot selection is the three-median. We randomly choose three elements in the sequence, and choose the median as the pivot. We may generalize this approach by choosing more elements and find their median.

Another possible improvement is to try some heuristic method. For example, when the partition is going to be much too biased (we can set a threshold for this), we can choose a new pivot from the partitioned numbers. We can try some different thresholds and choose the best one.

6 Discussion

6.1 General Definition of Sorting Algorithms Performance

The evaluation of an algorithm is mainly based on the following indicators:

- Time complexity. The time complexity of the algorithm refers to the amount of computation required to execute the algorithm.
- Spatial complexity. The spatial complexity of the algorithm refers to the memory space consumed by the algorithm. Compared with time complexity, the analysis of space complexity is much simpler.

It is worth noting that the same algorithm runs at different speeds in different compilation environments. Paul's [2] experiment shows that although the time complexity of heapsort is greater than that of quicksort, heapsort maybe much faster than quicksort in some environment. Obviously, the original intention of the improvement is to get faster running speed under specific conditions. Therefore, in addition to analyzing the time com-

plexity based on information entropy, paying attention to the actual running speed of the improved algorithm is also important.

6.2 Applicable Conditions and Stability

The percolating-down operation of original heapsort always wastes a lot of time if the number of data is very large. But the rearranged fast heapsort can solve this problem to a certain degree. According to the results, the rearranged fast heapsort has a short execution time than the others when the length of sequence is larger than 100000. But the simple fast heapsort does not show any advantage in execution time. So if the amount of data to be sorted is very large, the rearranged fast heapsort is a better choice.

It can be seen from the graph that the rearranged fast heapsort has a smoother curve with little fluctuation. So it is clear that this new algorithm is more stable than the original one for different length sequences.

6.3 Advantages and Disadvantages

The theoretical lower limit of sorting algorithm is demonstrated from the knowledge of information theory. Taking information entropy as reference, an improved heapsort algorithm is designed, and achieve the expected outcome. A feasible reference idea is proposed for the improvement of the fast sorting algorithm.

In the process of program testing, only a specific laptop is used, so the final results cannot represent the performance of the algorithm in general. For the improvement of quicksort algorithm, there is not enough time to propose a specific solution but only an idea.

7 Conclusion

We applied information theory to prove that sorting algorithm have a lower bound of time complexity: $\Omega(n \log n)$. And we discussed how heapsort and quicksort can be improved with information theory. Also, we did experiments and showed that improved version of heapsort is steadiest all the time, and fastest when n is larger than 100000.

References

[1] Heapsort, quicksort, and entropy.

[2] Sorting revisited.

[3] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. 1990.

[4] Michael L Fredman. How good is the information theory bound in sorting. *Theoretical Computer Science*, 1(4):355–361, 1976.

[5] Qi Guo Qing. Application of the concept of entropy in the research of the sorting algorithms. *Journal of Dalian Maritime University*, 2002.

[6] S S Kislitsyn. A finite partially ordered set and its corresponding set of permutations. *Mathematical Notes*, 4(5):798–801, 1968.