

1 Overview

Although sorting isn't a data structure, it uses many of the same ideas. The sorting problem, in which we wish to sort n elements according to a given ordering, has a tight $O(n \lg n)$ bound in the comparison model. In the integer sorting problem, we consider elements which are w -bit integers. By making this assumption and working in the word RAM model, we have these results for sorting:

The big open question is whether we can get $O(n)$ for all word sizes in this model.

- Comparison sort: $O(n \lg n)$
- Counting sort: $O(n + u) = O(n + 2^w)$ ($= O(n)$ for $w = \lg n$)
- Radix sort: $O(n \cdot \frac{w}{\lg n})$ ($= O(n)$ for $w = O(\lg n)$)
- van Emde Boas: $O(n \lg w)$. For $w = \lg^{O(1)} n$, this is $O(n \lg \lg n)$. This can be improved to $O(n \lg \frac{w}{\lg n})$, making it better than the previous methods (Kirkpatrick and Reisch [7]).
- Signature sort: $O(n)$ for $w = \Omega(\lg^{2+\varepsilon} n)$. Combined with the previous result for small w , this gives sorting in $O(n \lg \lg n)$ time for all word sizes. See Andersson, Hagerup, Nilsson, and Raman [2].
- Han [5]: $O(n \lg \lg n)$ deterministic, but only on the AC^0 RAM (signature sort is randomized).
- Han and Thorup [6]: $O(n \sqrt{\lg \lg n})$ randomized. Actually, one can achieve $O(n \sqrt{\lg \frac{w}{\lg n}})$, improving the result of [7].

We will prove the result in [2]. Combining this result with van Emde Boas gives an $O(n \lg \lg n)$ upper bound for all values of w . It is also worth noting that the hardness of integer sorting is concentrated in a narrow interval for w , between $\lg^{1+\varepsilon} n$ and $\lg^2 n$. At the ends of the interval, there is a relatively quick fall-off in the running time until it becomes linear.

2 Signature Sort

The *signature sort* in [2] allows us to sort in $O(n)$ time for $w \geq (\lg^{2+\varepsilon} n) \lg \lg n = \lg^{2+\varepsilon'} n$. We break each integer into $\lg^\varepsilon n$ equal-sized chunks, encoding each of these chunks in $O(\lg n)$ bits with a universal hash function. The result will be n *signatures* with $b = O(\lg^{1+\varepsilon} n)$ bits each. The hash codes for different chunks will be different with high probability.

The general strategy of the algorithm is as follows:

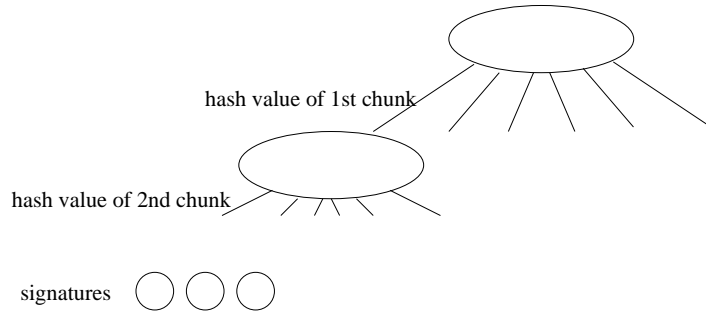


Figure 1: A trie of signatures. Each edge represents the hash value of a chunk, and the leaves are possible signature values.

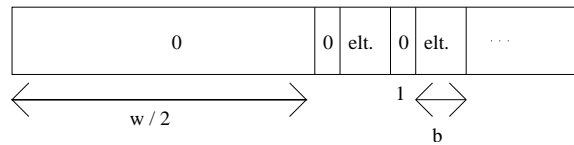


Figure 2: Packing b -bit integers into a w -bit word.

- Sort the signatures in linear time. This is now possible because they are significantly smaller than a word. We develop *packed sorting*, which takes $O(n)$ time to sort n integers of b bits each, given a word size of $w = \Omega(b \lg n \lg \lg n)$.
- Build a compressed trie over the signatures: see Figure 1.. This can also be done in linear time, using the same techniques as converting a suffix array into a suffix trie.
- Recursively sort the edges of each node in the trie based on the actual values, not the hashed values. That is, we sort (node ID, actual chunk, edge index) tuples, where the *actual chunk* is the original chunk value (of $\frac{w}{\lg^\varepsilon n}$ bits). Since this is the only interesting part of the sort, after $O(1/\varepsilon) = O(1)$ recursions the number of bits has been reduced by a $\lg n \lg \lg n$ factor. But then we can use packed sorting, which takes linear time.
- Once we have sorted the edges based on actual value, we can sort the edges at each node in the trie.
- Perform an in-order traversal of the trie, outputting the leaves. This is the sorted list of integers.

The only remaining part of this algorithm is packed sorting.

3 Packed Sorting

Packed sorting, due to Albers and Hagerup [1], can sort n integers of b bits in $O(n)$ time, given a word size of $w \geq 2(b + 1) \lg n \lg \lg n$. We can therefore pack $\lg n \lg \lg n$ elements into one word in memory. We leave one zero bit between each integer, and $w/2$ zero bits in the high half of the word; see Figure 2.

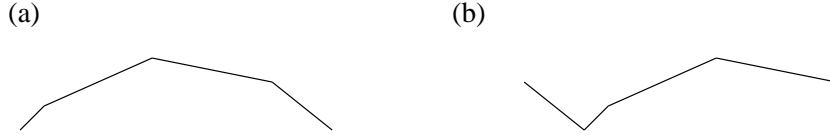


Figure 3: **(a)** A bitonic sequence. **(b)** A cyclic shift of (a) is also a bitonic sequence.

We use an adapted version of mergesort to sort the elements. We have four main operations that allow us to do this:

1. Merge a pair of sorted words with $k \leq \lg n \lg \lg n$ elements into one sorted word with $2k$ elements. In Section 3.1, we show how to do this in $O(\lg k)$ time.
2. Merge sort $k \leq \lg n \lg \lg n$ elements, yielding a packed word with elements in order. Using (1) for the merge operation, this takes time $T(k) = 2T(\frac{k}{2}) + O(\lg k)$. Using the master theorem or drawing the recursion tree shows the leaves dominate the running time, so $T(k) = O(k)$.
3. Merge two sorted lists of r words, each word containing $k = \lg n \lg \lg n$ sorted elements, into one sorted list of $2r$ sorted words. We do this by removing the first word of each list and merging them using (1). The first half of the resulting word can be output, since its k elements are necessarily the smallest of all those remaining. We then mask the second half of the word, which contains the larger k elements. This word is placed at the beginning of the list which formerly contained the maximum element in the word, maintaining the sortedness of the lists. We take $O(\lg k)$ time to output a word, so the merge operation takes total time $O(r \lg k)$.
4. Merge sort with (3) as the merge operation and (2) as the base case, yielding a recurrence of $T(n) = 2T(\frac{n}{2}) + O(\frac{n}{k} \lg k)$, where $k = \lg n \lg \lg n$. There are $\lg \frac{n}{k} = O(\lg n)$ internal levels in the recursion tree, each taking total time $O(\frac{n}{k} \lg k) = O(\frac{n}{\lg n})$. So internal levels contribute a cost of $O(n)$. The $\frac{n}{k}$ leaves each take $O(k)$ time, so the total cost of the leaves is also $O(n)$.

3.1 Merging Words

We will use bitonic sorting networks and bit tricks to merge two words together. A *bitonic sequence* is one for which a cyclic shift will result in a sequence which increases monotonically and then decreases monotonically; see Figure 3. A bitonic sequence can be sorted by putting all pairs $A[i]$ and $A[i + \frac{n}{2}]$ in the correct order for $i = 1, 2, \dots, \frac{n}{2}$, and then recursively sorting the first and second halves of the data. Each step uses $\frac{n}{2}$ comparisons and potential swaps, and the recursion has depth $O(\lg n)$. A proof of correctness can be found in [3, Chapter 27].

We can use a bitonic sort to merge two words of k elements. We first reverse the second word and then concatenate the two words, leaving a bitonic sequence. Reversing a word can be done by masking out the leftmost $\frac{k}{2}$ elements and shifting them right by $\frac{k}{2}b$, and similarly masking out the rightmost $\frac{k}{2}$ elements and shifting them left by $\frac{k}{2}b$. Taking the OR of the two resulting words will give a word with the left and right halves of the original word swapped. We now recursively reverse the left and right halves of the word *in parallel*, so that each level of recursion takes $O(1)$ time. After $\lg k$ recursions we reach the base case where there is only one element to be reversed, so the total time to reverse a word of k elements is $O(\lg k)$. The two words may now be concatenated by shifting the first word left by kb and taking its OR with the second word. See Figure 4.

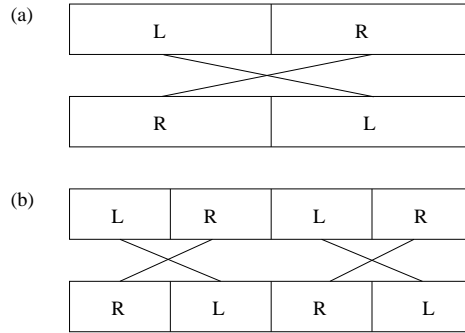


Figure 4: **(a)** The first recursion in reversing a list. **(b)** Next, the two halves are each divided into left and right halves which are swapped and recursively sorted.

All that remains is to run the bitonic sorting algorithm on the elements in our new word. To do so, we must divide the elements in two halves and swap corresponding pairs of elements which are out of order. Then we can recurse on the first and second halves in parallel, performing $\lg k$ total recursions. Thus we need a constant-time operation which will perform the desired swapping.

Recall that we left an extra 0 bit before each element when we packed them into a word. We will mask the left half of the elements and set this extra bit to 1 for each element, then mask the right half of the elements and shift them left by $\frac{k}{2}b$. If we subtract the second word from the first, a 1 will appear in the extra bit if and only if the element in the corresponding position of the left half is greater than the element in the right half. Thus we can mask the extra bits, shift the word right by $b - 1$ bits, and subtract it from itself, resulting in a word which will mask all the elements of the right half which belong in the left half and vice versa. Similarly, negating this word will mask all elements which belong in their current position. Simple shifts and OR operations will then produce the desired result, a word containing $2k$ sorted elements. See Figure 5.

We therefore have a constant-time operation which performs the desired operation from bitonic sorting. Recursively sorting both halves in parallel will yield $\lg k$ levels of recursion, leading to the $O(\lg k)$ time for operation (1). Packed sorting in $O(n)$ time immediately follows, as does our $O(n)$ time result for $w = \Omega(\lg^{2+\epsilon} n)$.

4 Priority Queues

If there is an $O(nS(n, w))$ sorting algorithm, there is an $O(S(n, w))$ worst-case priority queue (with insert, delete, find-min) (see Thorup [8]). It is conjectured by Demaine and Pătraşcu that one can also get the meld operation (Mendelson, Tarjan, Thorup, and Zwick [9]) and $O(1)$ decrease-key.

References

- [1] Susanne Albers, Torben Hagerup: *Improved Parallel Integer Sorting without Concurrent Writing*, Inf. Comput. 136(1): 25-51, 1997.
- [2] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, *Sorting in Linear Time?*, J. Comput. Syst. Sci. 57(1): 74-93, 1998.

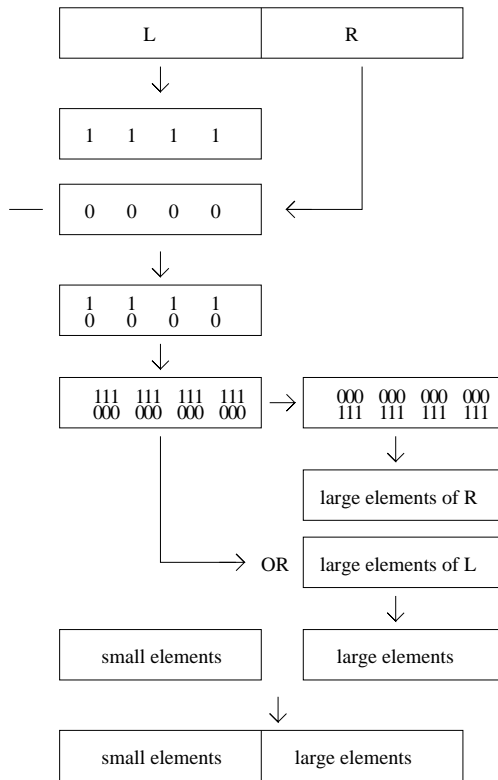


Figure 5: Sorting pairs of corresponding elements in the left and right halves of a word. Extra bits are set in the left half, the right half of the word is shifted and subtracted, and a mask is created from the result. The large elements are then masked out of both halves using this mask and its negation. In a similar process, the small elements are found, and the two are finally appended together.

- [3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*, Second Edition, The MIT Press and McGraw-Hill Book Company 2001.
- [4] H.N. Gabow, J.L. Bentley, R.E. Tarjan: *Scaling and Related Techniques for Geometry Problems*, STOC 1984: 135-143.
- [5] Y. Han: *Deterministic Sorting in $O(n \log \log n)$ Time and Linear Space*, J. Algorithms 50(1): 96-105, 2004.
- [6] Y. Han, M. Thorup: *Integer Sorting in $O(n\sqrt{\log \log n})$ Expected Time and Linear Space*, FOCS 2002: 135-144.
- [7] D.G. Kirkpatrick, S. Reisch: *Upper Bounds for Sorting Integers on Random Access Machines*, Theoretical Computer Science 28: 263-276 (1984).
- [8] M. Thorup: *Equivalence between Priority Queues and Sorting*. FOCS 2002: 125-134 (2002).
- [9] R. Mendelson, R. Tarjan, M. Thorup, and U. Zwick. *Melding Priority Queues*. Proceedings of 9th SWAT, 2004.