6.851: Advanced Data Structures Spring 2012

Lecture 14 — April 10, 2012

Prof. Erik Demaine *Scribes: Malvika Joshi (2017), Jeet Mohapatra (2017), Henry Wu (2017), Leon Bergen (2012), Andrea Lincoln (2012), Tana Wattanawaroon (2012), Haitao Mao (2010),Eric Price (2007)*

1 Overview

Today we are going to cover integer sorting. The main topics are:

- Reductions between sorting and priority queues
- Survey of integer sorting
- Bitonic Sequences
- Logarithmic Merge Operation
- Packed Sorting
- Signature Sort Algorithm

The reduction and survey will serve as motivation for signature sort. Bitonic Sequences will be used to build Logarithmic Merge which will in turn be used to build Packed Sorting which will in finally be used to build Signature Sort.

2 Priority Queues

Thorup [7] showed that if we can sort n w-bit integers in $O(nS(n, w))$, then we have a priority queue that can support the insertion, deletion, and find minimum operations in worst case $O(S(n, w))$. To get a constant time priority queue, we need linear time sorting.
OPEN: linear time sorting

3 Survey of Integer Sorting

	Complexity	Simplifications
Comparison Sorting	$O(n \lg n)$	
Counting Sort	$O(n + 2^w)$	$O(n)$ for $w = \lg n$ only
Radix Sort	$O(n \frac{w}{\lg n})$	$O(n)$ for $w = O(\lg n)$
vEB Sort [6]	$O(n \lg \lg w)$	$O(n \lg \lg n)$ for $w = \lg^{O(1)} n$
	improved: $O(n \lg \frac{w}{\lg n})$	
Signature Sort [2]	$O(n)$ for $w = \Omega(\lg^{2+\epsilon} n)$	
Han [4]	$O(n \lg \lg n)$ in AC^0	
Han & Thorup [5], [6]	$O(n\sqrt{\lg \frac{w}{\lg n}})$ randomized	$O(n\sqrt{\lg \lg n})$ for all w

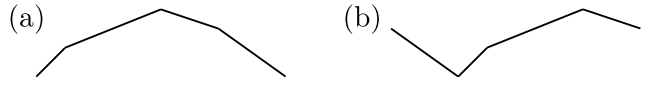
We can sort in linear time if $w = O(\lg n)$ using radix sort. If $w = \Omega(\lg^{2+\epsilon} n)$, then signature sort works and sorts in linear time. The optimal sorting algorithm for $w = \omega(\lg n)$ and $w = o(\lg^{2+\epsilon} n)$ is an open problem. The best we have done in the general case is a randomized algorithm in $O(n\sqrt{\log \frac{w}{\log n}})$ time by Han, Thorup, Kirkpatrick, and Reisch.
OPEN: $O(n)$ time sort for all w

4 Sorting for $w = \Omega(\log^{2+\varepsilon} n)$

The signature sort was developed in 1998 by Andersson, Hagerup, Nilsson, and Raman [2]. It sorts n w-bit integers in $O(n)$ time when $w = \Omega(\log^{2+\varepsilon} n)$ for some $\varepsilon > 0$. This is a pretty complicated sort, so we will build the algorithm from the ground up. First, we give an algorithm for sorting bitonic sequences using methods from parallel computing. Second, we show how to merge two words of $k \leq \log n \log \log n$ elements in $O(\log k)$ time. Third, using this merge algorithm, we create a variant of MERGESORT called *packed sorting*, which sorts n b-bit integers in $O(n)$ time when $w \geq 2(b+1) \log n \log \log n$. Fourth, we use our packed sorting algorithm to build signature sort.

4.1 Bitonic Sequences

A *bitonic sequence* is a cyclic shift of a monotonically increasing sequence followed by a monotonically decreasing sequence. When examined cyclically, it has only one local minimum and one local maximum.



Cyclic shifts preserve the binoticity of a sequence.

To sort a bitonic sequence `btcseq`, we run the algorithm given below. Assume $n = \text{len}(\texttt{btcseq})$ is even.

Algorithm 1 Bitonic sequence sorting algorithm

```
btcsort(btcseq):
for i from 0 to n/2-1:
if btcseq[i]>btcseq[i+n/2]:
swap(btcseq[i], btcseq[i+n/2])
btcsort(btcseq[0:n/2-1])
btcsort(btcseq[n/2:n-1])
```

Bitonic sequence sorting maintains two invariants: (a) The sequence `btcseq` contains multiple sections, where each section is a substring of that sequence and also a bitonic sequence itself. (Each level of `btcsort` splits a bitonic sequence into two sections)

(b) When considering two sections of `btcseq`, an element in the left section is always smaller than an element in the right section.

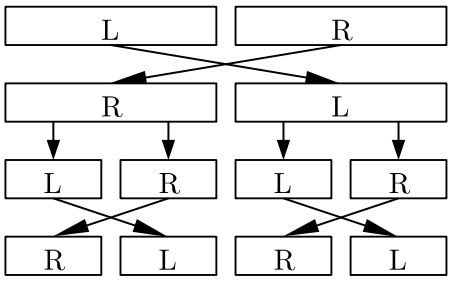
After $O(\log n)$ rounds, `btcseq` is broken into n sections. Hence, due to invariant (b), the sorting is complete.

For more information about sorting bitonic sequences, including a proof of the correctness of this algorithm, see [3, Section 27.3].

4.2 Logarithmic Merge Operation

The next step is to merge two sorted words, each containing k b-bit elements. First, we concatenate the first word with the reverse of the second word, getting a bitonic sequence.

To efficiently reverse a word, we mask out the leftmost $\frac{k}{2}$ elements and shift them right by $\frac{k}{2}b$, then mask out the rightmost $\frac{k}{2}$ elements and shift them left by $\frac{k}{2}b$. Taking the OR of the two resulting words leaves us with the original word with the left and right halves swapped.
We can now recurse on the left and right halves of the word, giving us the recursion $T(k) = T(\frac{k}{2}) + O(1)$, so the whole algorithm takes $T(k) = O(\log k)$ time.
The key here is to perform each level of the recursion in parallel, so that each level takes the same amount of time. This list reversal is illustrated in the figure below.



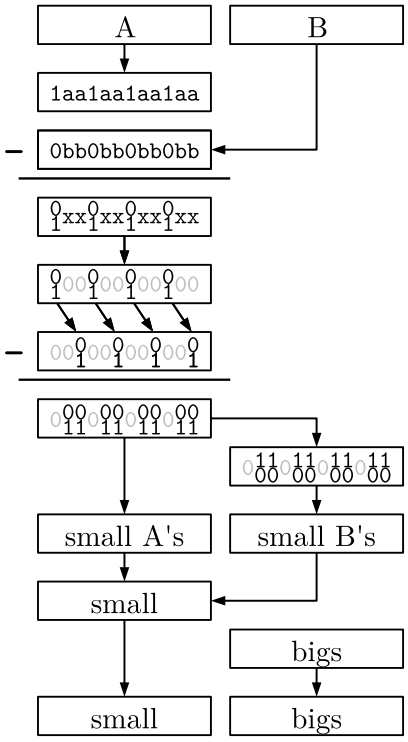
The first two steps in the recursion for reversing a list.

The two words may now be concatenated by shifting the first word left by kb and taking its OR with the second word.

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, once again giving us the recursion $T(k) = 2T(\frac{k}{2}) + O(1) \Rightarrow T(k) = O(\log k)$ time. Thus we need a constant-time operation which will perform the desired swapping.

Assume we have an extra 0 bit before each element packed into the word. We use this spare bit to help bitmask our word.
We will mask out the left half of the elements into a word set this extra bit to 1 for each element, then mask out the right half of the elements into a second and shift them left by $\frac{k}{2}b$. After we subtract the second word from the first, a 1 will appear in the extra bit iff 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. We can combine these together using bitwise operations to get the desired list.

See the diagram below for clarification.



Parallel swap operation in bitonic sorting.

Using these bit tricks, we end up with our desired constant time all-swap operation. This leads to the following theorem:

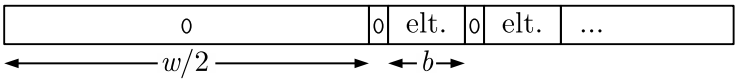
Theorem 1. *Suppose two words each contain k b-bit elements. Then we can merge these words in $O(\log k)$ time.*

Proof. As already noted, we can concatenate the first word with the reverse of the second word

in $O(\log k)$ time. We can then apply the bitonic sorting algorithm to these concatenated words. Since this algorithm has a recursion depth of $O(\log k)$, and we can perform the required swaps in constant time, our merge operation ends up with time complexity $O(\log k)$. □

4.3 Packed Sorting

In this section we will present packed sorting [1], which sorts n b-bit integers in $O(n)$ time for a word size of $w \geq 2(b+1)\log n \log\log n$. This bound for w allows us to pack $k = \log n \log\log n$ elements into one word, leaving a zero bit in front of each integer, and $\frac{w}{2}$ zero bits at the beginning of the word.



Structure for packing b-bit integers into a w-bit word.

The n elements can be packed into the words in $O(n)$ time. To sort these elements using packed sorting, we'll use the following merging sub-routines:

1. Merge two sorted words in $O(\lg k)$ time: This is shown in the previous section using bitonic sorting.
2. Sort k elements packed in a single word: This can be done with a merge sort, where we split the word into two words containing $k/2$ elements each and merge them using (1) after recursively sorting them.

   The time taken has the recurrence $T(k) = 2T(\frac{k}{2}) + O(\lg k) \Rightarrow T(k) = O(k)$.
3. Merge two sorted lists of r sorted words into one sorted list of $2r$ sorted words:

   We assume that each word contains exactly k elements already sorted. This is done in a similar manner as in a standard merge sort and we use (1) to speed it up. In each step of the merge, we take the left most-words from each of the two lists and merge them using (1). The first word that is a result of this merge will contain the smallest k elements, so we output that. However, the position of the second word from the result is unclear, so we put it back into one of the lists and repeat. To determine where to put back the second word, we examine the maximum element in it, and add the it to the beginning of the list that initially contained this maximum element.
   We repeat this until all elements have been output, resulting in $O(r \log k)$ overall.

Now, we can use merge sort on the words using, (3) to merge two sorted lists of words. When we get to the base case of 1 word, we use (2) to sort it.

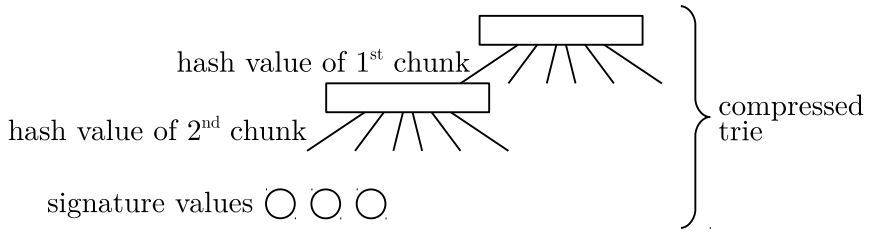
Theorem 2. *Let word size $w \geq 2(b+1)\log n \log\log n$. Then packed sorting sorts n b-bit integers in $O(n)$ time.*

Proof. The sort defined in Step 4 of the algorithm above has the recursion $T(n) = 2T(\frac{n}{2}) + O(\frac{n}{k}\log k)$ and the base case $T(k) = O(k)$. The recursion depth is $O(\log \frac{n}{k})$, and each level takes $O(\frac{n}{k}\lg k) = O(\frac{n}{\lg n})$ time, so altogether they contribute a cost of $O(n)$. There are also $\frac{n}{k}$ base cases, each taking $O(k)$ time, giving a total runtime of $O(n)$, as desired. □

4.4 Signature Sort Algorithm

We use our packed sorting algorithm to build our seven step signature sort. We assume that the word size $w \geq \log^{2+\varepsilon} n \log\log n$ and that the size of numbers is $w' \leq \log^{2+\varepsilon} n$. The signature sort will sort n w'-bit integers in $O(n)$ time. We will start by breaking each integer into $\lg^{\varepsilon} n$ equal-size chunks, which will then be replaced by $O(\lg n)$-bit signatures via hashing. These smaller signatures of the integers now satisfy $w \geq b \log n \log\log n$ and thus can be sorted with packed sorting in linear time. However, we are not yet done as hashing does not preserve order. In order to deal with this we will build a compressed trie of these sorted signatures, which we will then sort according to the values of the original chunks. This will allow us to recover the sorted order of the integers. The signature sort will proceed as follows:

1. Break each integer into $\lg^{\varepsilon} n$ equal-size chunks. (Note the distinguishment from a fusion tree, which has chunks of size $\lg^{\varepsilon} n$)
2. Replace each chunk by a $O(\lg n)$-bit hash (static perfect hashing is fine). By doing this, we end up with n $O(\lg^{1+\varepsilon} n)$-bit *signatures*. One way we can hash is to multiply by some random value x, and then mask out the hash keys. This will allow us to hash in linear time. Now, our hash does not preserve order, but the important thing is that it does preserve identity.
3. Sort the signatures in linear time with packed sorting, shown below.
4. Now we want to rescue the identities of the signatures. Build a compressed trie over the signatures, so that an inorder traversal of the trie gives us our signatures in sorted order. The compressed trie only uses $O(n)$ space.



A trie for storing signatures. Edge represent hash values of chunks; leaves represent possible signature values.

   To do this in linear time, we add the signatures in order from left to right. Since we are in the word RAM, we can compute the Longest Common Prefix with $(i-1)^{\text{st}}$ signature by taking the most significant 1 bit of the XOR. Then, we walk up the tree to the appropriate node, and charge the walk to the decrease in the rightmost path length of the trie. The creation of the new branch is constant time, so we get linear time overall. This process is similar to the creation of a Cartesian tree.
5. Recursively sort the edges of each node in the trie based on their actual values. This is a recursion on (node ID, actual chunk, edge index), which takes up $(O(\log n), O(\frac{w'}{\log^{\varepsilon} n}), O(\log n))$ space. The edge indices are in there to keep track of the permutation. So we see that the above steps have reduced the size of the integers to be sorted from w' to $\frac{w'}{\log^{\varepsilon} n} + O(\log n)$. Repeating this we see that after a constant $\frac{1}{\varepsilon}+1$ levels of recursion, we will have $b = O(\log n + \frac{w'}{\log^{1+\varepsilon} n}) = O(\frac{w}{\log n \log\log n})$, so we can use packed sorting as the base case of the recursion.

6. Next we permute the child edges to get back the actual sorted order.

7. Do an inorder traversal of the trie to get the desired sorted list from the leaves.

Putting these steps together, we get:

Theorem 3. *Let word size be $w \geq \log^{2+\varepsilon} n \log\log n$ and integer size $w' \leq \log^{2+\varepsilon} n$. Then signature sort will sort n w'-bit integers in $O(n)$ time.*

Proof. Breaking the integers into chunks and hashing them (Steps 1 and 2) takes linear time. Sorting these hashed chunks using packed sort (Step 3) also takes linear time. Building the compressed trie over signatures takes linear time (Step 4), and sorting the edges of each node (Step 5) takes constant time per node for a linear number of nodes. Finally, scanning and permuting the nodes (Step 6) and the in-order traversal of the leaves of the trie (Step 7) will each take linear time, completing the proof. □

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.

[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] Y. Han: *Deterministic Sorting in $O(n \log\log n)$ Time and Linear Space*, J. Algorithms 50(1): 96-105, 2004.

[5] Y. Han, M. Thorup: *Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space*, FOCS 2002: 135-144.

[6] D.G. Kirkpatrick, S. Reisch: *Upper Bounds for Sorting Integers on Random Access Machines*, Theoretical Computer Science 28: 263-276 (1984).

[7] M. Thorup: *Equivalence between Priority Queues and Sorting*. FOCS 2002: 125-134 (2002).