Fortgeschrittene Datenstrukturen — Vorlesung 6

Schriftführer: Sebastian Schlag

24.11.2011

1 Integer Sorting

The sorting problem of sorting n elements has a tight $\Theta(n \lg n)$ bound in the comparison model. In this lecture, we will cover a special case of the general sorting problem, namely *integer sorting*, in which we have to sort n w-bit integers.

By working in the RAM model, we assume that the word size of our computer is in $\Omega(w)$, so we can address and manipulate a single key in $\mathcal{O}(1)$ time.We will also consider shorter keys of $b \leq w$ bits. In general, we use the notation `SORT(`n`,`w`)` for the complexity of sorting n w-bit integers

Section 1.1 provides an overview of different integer sorting algorithms. Section 1.2 will then cover *signature sort*, which runs in $\mathcal{O}(n)$ expected time for $w \geq \lg^{2+\varepsilon} n \lg \lg n$.

1.1 Overview

As figure 1 shows it is still an open question if there is a linear time sorting algorithm for $w = \omega(\lg n)$ and $w = o(\lg^{2+\varepsilon} n)$. Currently, the best known algorithm covering this range is due to Han and Thorup [4].

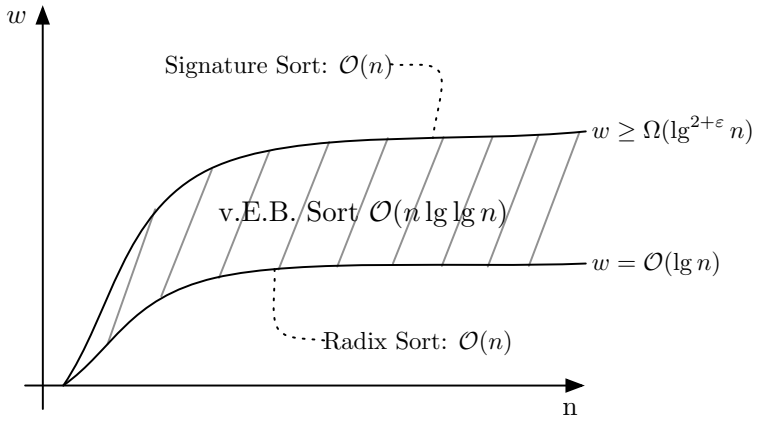

Figure 1: Signature sort, radix sort and v.E.B. sort and their corresponding runtime in respect to n and w

Following is a list of results on the problem of integer sorting:

- *Counting sort* runs in $\text{SORT}(n, w) = \mathcal{O}(n + 2^w)$, which is linear for $w \leq \lg n$.

- *Radix sort* runs in $\text{SORT}(n, w) = \mathcal{O}\left(n \cdot \frac{w}{\lg n}\right)$, which is linear for $w = \mathcal{O}(\lg n)$
- *van Emde Boas sort* runs in $\text{SORT}(n, w) = \mathcal{O}(n \lg w)$, which is $\mathcal{O}(n \lg\lg n)$ for $w = \lg^{\mathcal{O}(1)} n$ and can be improved to $\mathcal{O}\left(n \lg \frac{w}{\lg n}\right)$ (see [5]).
- Han and Thorup [4] present a randomized algorithm for sorting n integers in $\mathcal{O}(n\sqrt{\lg\lg n})$ expected time.

1.2 Signature Sort

The basic idea of signature sort [1] is to break each integer key into chunks and (similar to fusion trees) reduce the bit-length of each chunk such that the resulting integers can be sorted in $\mathcal{O}(n)$ time using *packed sorting* (which is subject of the next lecture).

Let $X_1, \ldots, X_n$ denote the set of integer keys to be sorted and assume $w \geq \lg^{2+\varepsilon} n \lg\lg n$. We begin by conceptually partitioning each key in q *chunks* of size $\frac{w}{q}$ (see fig. 2). During the analysis we will impose *two* requirements on q. At the end of this subsection we will then prove that q can be chosen such that both requirements are fulfilled.

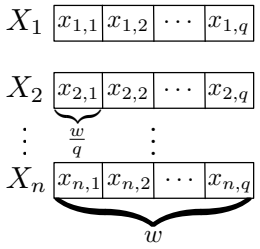

Figure 2: Each w-bit key X_i is broken into q chunks of size $\frac{w}{q}$.

In order to be able to sort these chunks efficiently, we use a universal hash function h_a that maps each chunk to a *unique* hash-value that is significantly smaller than the original chunk. The hash-value $h_a(x_{i,j})$ of a chunk $x_{i,j}$ will be called the *signature* of $x_{i,j}$. As key X_i consists of chunks $x_{i,1}, \ldots, x_{i,q}$, we define the concatenated signature $h_a(X_i)$ of X_i as the integer obtained by concatenating the signatures $h_a(x_{i,1}), \ldots, h_a(x_{i,q})$ (see fig.3).

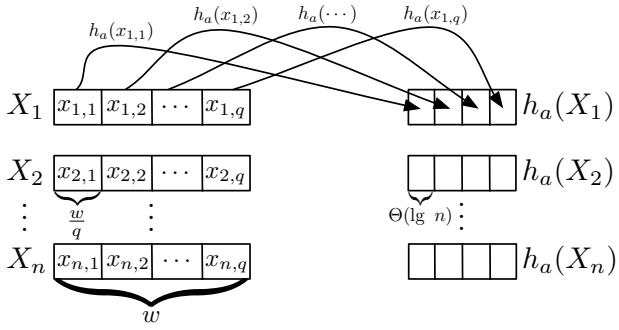

Figure 3: Using h_a to create the signature $h_a(X_i)$ of key X_i and thereby reducing the length of each key from w bits to $\Theta(q \lg n)$ bits.

Because we want the signatures to be *unique*, our hash function has to operate *injectively* on the set of all $n \cdot q$ chunks occuring in the input keys. Furthermore, we require the length of signature $h_a(x_{i,j})$ to be in $\Theta(\lg n)$.

The following class $\mathcal{H}_{k,l}$ of 2-universal hash functions [3] satisfies these requiremens:

$$\mathcal{H}_{k,l} = \{h_a | 0 \leq a \leq 2^k \wedge a \text{ is odd}\} \text{ where } h_a \text{ is defined by:}$$
$$h_a \colon \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^l - 1\}$$
$$h_a(x) = (ax \mod 2^k) \operatorname{div} 2^{k-l}$$

Dietzfelbinger et al. [3] prove (see Lemma 2.6) that given integers k and l with $1 \leq l \leq k$ and a set S of integers in the range $\{0, \ldots, 2^k - 1\}$: If $h_a \in H_{k,l}$ is chosen at random, then

$$\text{Prob}(h_a \text{ is injective on } S) \geq 1 - \frac{|S|^2}{2^l}.$$

Recall that in our case $S = n \cdot q$ is the set of all chunks occuring in the input keys. Without loss of generality we can assume that $q \leq n$ and therefore $|S| \leq n^2$. Thus we can assure that

$$\text{Prob}(h_a \text{ is } \textbf{not} \text{ injective on S}) \leq \frac{1}{n^2}$$

by choosing $l = \Theta(\lg n)$ appropriately (e.g. $l > \lg n^6 = 6 \lg n \in \Theta(\lg n)$). Therefore it is ensured that by choosing h_a uniformly at random, we will find an injective hash function in $\mathcal{O}(1)$ expected time with high probability.

Note that at this point, we have reduced the problem of sorting n w-bit integers to that of sorting n $\Theta(q \lg n)$-bit integers.

We now use *packed sorting* to sort the n length-reduced signatures $h_a(X_1), \ldots, h_a(X_n)$ in $\mathcal{O}(n)$ time (see fig. 4). To be able to use this algorithm to sort n b-bit integers, we have to ensure that

$$b \leq \mathcal{O}\left(\frac{w}{\lg n \lg \lg n}\right).$$

In other words: it has to be possible to pack $\Omega(\lg n \lg \lg n)$ keys into one word w. This imposes our *first* requirement for q, since we want to sort signatures of length $b = \Theta(q \lg n)$.

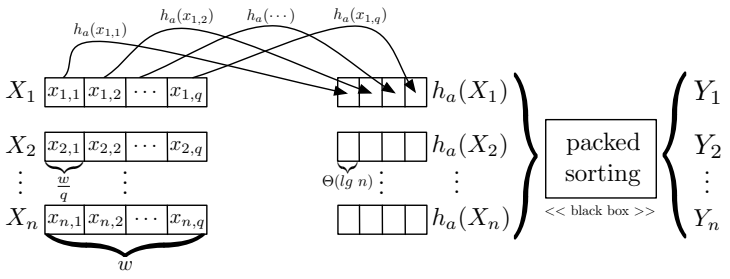

Figure 4: Using packed sorting to sort the signatures $h_a(X_i)$. The result is the sorted sequence $Y_1, \ldots, Y_n$ with $(Y_1 \leq \ldots \leq Y_n)$.

Let $Y_1, \ldots, Y_n$ now be the signatures in ascending order $(Y_1 \leq \ldots \leq Y_n)$ after sorting (see figure 4). Since our only requirement on h_a was to be injective (and not neccesarily to be monotonic),

signatures are now arranged in a differend order than that required by the original sorting problem. To solve this, we construct a trie (interpreting each hashed chunk as a character) by adding the sorted signatures Y_i one-by-one from smallest to largest (see figure 6 for an example):

Assuming that we have already built a trie for $Y_1, \ldots, Y_{i-1}$, we add Y_i as follows:

- At first we compute the *longest common prefix* (LCP) of Y_{i-1} and Y_i, which corresponds to the number of equal signature-chunks in Y_{i-1} and Y_i, in $\mathcal{O}(1)$. This can be done for example by taking the most significant bit of (Y_{i-1} XOR Y_i).
- Then we walk up the rightmost path of the trie beginning at leaf Y_{i-1} in order to find the insertion point of Y_i. There are two possibilities:
  (a) There already is a branching node at the LCP. Then Y_i is simply added as one of its children (see fig. 5 (a)).
  (b) No branching node exists at the LCP position. In that case, we break up the edge after the LCP and add a new branching node. The remainder of that edge (which leads to the subtree that contains Y_{i-1} as the rightmost leaf) as well as a new edge containing the different suffix of Y_i are added as children to the new branching node (see fig. 5 (b)).

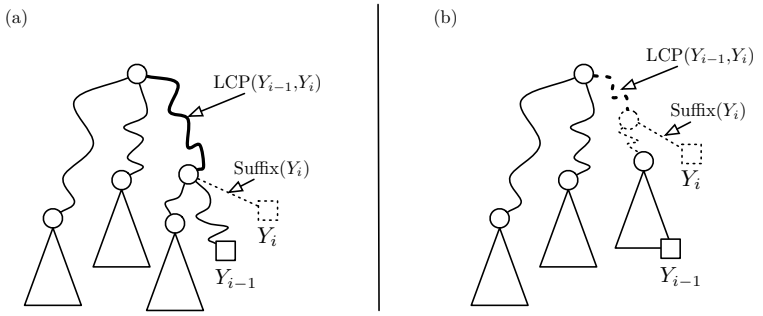

Figure 5: Inserting a new signature Y_i. (a) There already is a branching node at the LCP. Then Y_i is simply added as one of its children. (b) A new branching node is added by breaking up an edge if it does not exist at the correct position. Both the subtree containing Y_{i-1} as its rightmost leaf and Y_i will be added as children of that node. The different suffixes will be used as edge lables.

It is sufficient to consider only the rightmost path when inserting a signature Y_i, because we add all of them in sorted order (small $\rightarrow$ large). Therefore, if $LCP(Y_{i-1}, Y_i) > 0$ the branching node (already existing or not) has to be on the rightmost path. In case of $LCP(Y_{i-1}, Y_i) = 0$ the new signature will be added as a new child of the root node. The trie can be constructed in $\mathcal{O}(n)$ time, because we add every signature only once at a cost of $\mathcal{O}(1)$ for finding the correct insertion point and $\mathcal{O}(1)$ for adding it at that position.

Given this trie, we are now able to acutally sort the original input keys by sorting the edges of each internal node - using the original chunk values instead of the signature-labels as sort keys (see figure 6 (c) for an example). Because the order of an edge is determined by the first symbol on that edge, it is sufficient to use the first chunk as the sort key if an outgoing edge is labled with more than one chunk.

Thus our new sorting problem is to sort tuples of the form *(nodeID,original chunk value, edge index)* - one for each edge in the trie. The edge index is used to keep track of the permutation so that we are able to permute the edges accordingly after sorting.

Because the trie contains $\mathcal{O}(n)$ edges, the reduced problem is to sort $\mathcal{O}(n)$ keys of size:

$$\mathcal{O}\left(\lg n + \frac{w}{q} + \lg n\right) = \mathcal{O}\left(\frac{w}{q}\right), \text{ since } \lg n \in o(w) \text{ by our requirements on } w$$

To do so, we recurse on these $\mathcal{O}\left(\frac{w}{q}\right)$-bit keys, thereby successively reducing their length until it is possible to use *packed sorting* as the base case. As we want the overall sorting time to be linear, we have to ensure that the recursion ends after $\mathcal{O}(1)$ steps. This imposes our *second* requirement for q.

After the recursion we are able to sort the original edges of the trie according to the results of the recursive sort-call by scanning through the result-list of sorted edges and permutating the edges of the trie accodingly. Finally, the sorted sequence of the n input keys can be read off the trie by a final left-to-right scan in $\mathcal{O}(n)$ time.

It remains to be shown that q can be defined such that it fulfills the imposed requirements:

1. $q \lg n \leq \mathcal{O}\left(\frac{w}{\lg n \lg\lg n}\right)$ in order to be able to sort the signatures $h_a(X_1), \ldots, h_a(X_n)$ in $\mathcal{O}(n)$ time with packed sorting.
2. $\frac{w}{q^x} \leq \mathcal{O}\left(\frac{w}{\lg n \lg\lg n}\right)$: After x recursion steps, the keys have to be small enough in order to be sorted with *packed sorting* with $x = \mathcal{O}(1)$.

Lemma 1. $q = \Theta\left(\frac{w}{\lg^2 n \lg\lg n}\right) = \Theta\left(\frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg^2 n \lg\lg n}\right) = \Theta(\lg^{\varepsilon} n)$ *fulfills the above mentioned requirements and leads to a length-reduction by a factor of* $\Theta(\lg n \lg\lg n)$ *in both cases.*

Proof. q fulfills requirement 1:

$$\begin{aligned} & q \lg n \leq \mathcal{O}\left(\frac{w}{\lg n \lg\lg n}\right) \\ \Leftrightarrow\ & \lg^{\varepsilon} n \lg n \leq \mathcal{O}\left(\frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg n \lg\lg n}\right) \\ \Leftrightarrow\ & \lg^{\varepsilon} n \lg n \leq \mathcal{O}(\lg^{1+\varepsilon} n) \\ \Leftrightarrow\ & \lg^{1+\varepsilon} n \leq \mathcal{O}(\lg^{1+\varepsilon} n) \end{aligned}$$

q fulfills requirement 2:

$$\begin{aligned} & \frac{w}{q^x} = \Theta(\lg n \lg\lg n) \\ \Leftrightarrow\ & q^x = \mathcal{O}\left(\frac{w}{\lg n \lg\lg n}\right) = \mathcal{O}\left(\frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg n \lg\lg n}\right) = \mathcal{O}(\lg^{1+\varepsilon} n) \\ \Leftrightarrow\ & x = \mathcal{O}\left(\frac{\lg(\lg^{1+\varepsilon} n)}{\lg q}\right) = \mathcal{O}\left(\frac{\lg(\lg^{1+\varepsilon} n)}{\lg(\lg^{\varepsilon} n)}\right) = \mathcal{O}\left(\frac{1+\varepsilon}{\varepsilon}\right) \end{aligned}$$

After $\mathcal{O}\left(1+\frac{1}{\varepsilon}\right)$ recursion steps, the keys are small enough to be sorted with packed sorting:

$$\begin{aligned} \frac{w}{q^{1+\frac{1}{\varepsilon}}} &\leq \mathcal{O}\left(\frac{w}{\lg n \lg\lg n}\right) \\ \Leftrightarrow \frac{w}{(\lg^{\varepsilon} n)^{1+\frac{1}{\varepsilon}}} &\leq \mathcal{O}(\lg^{1+\varepsilon} n) \\ \Leftrightarrow \frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg^{1+\varepsilon} n} &\leq \mathcal{O}(\lg^{1+\varepsilon} n) \\ \Leftrightarrow \lg n \lg\lg n &\leq \mathcal{O}(\lg^{1+\varepsilon} n) \end{aligned}$$

In both cases, a length-reduction by a factor of $\Theta(\lg n \lg\lg n)$ was sufficient to be able to use *packed sorting* in order to sort in $\mathcal{O}(n)$ time:

- Using hashing to create signatures reduced the size of a key by a factor of $\Theta\left(\frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg^{1+\varepsilon} n}\right) = \Theta(\lg n \lg\lg n)$
- After a total of $\mathcal{O}\left(1+\frac{1}{\varepsilon}\right)$ recursion steps, the size of a key is also reduced by a factor $\mathcal{O}\left(\frac{w}{q^{1+\frac{1}{\varepsilon}}}\right) = \Theta\left(\frac{\lg^{2+\varepsilon} n \lg\lg n}{\lg^{1+\varepsilon} n}\right) = \Theta(\lg n \lg\lg n)$

□

To sum up, let us recall the different steps of signature sort:

1. Partition each key into $q = \Theta(\lg^{\varepsilon} n)$ chunks of size $\frac{w}{q} = \Theta(\lg^2 n \lg\lg n)$.
2. Create a signature of length $\Theta(q \cdot \lg n)$ for each key in $\mathcal{O}(n)$ time using an injective hash function h_a chosen uniformly at random from the 2-universal class of hash functions $\mathcal{H}$. Since $\mathcal{H}_{k,l}$ is 2-universal, we are able to find such h_a in $\mathcal{O}(1)$ expected time.
3. Sort these signatures in $\mathcal{O}(n)$ time using *packed sorting*.
4. Build a trie on the sorted signatures in $\mathcal{O}(n)$ time.
5. Recursivly sort the edges of that trie with the original chunk values as input keys. After $\mathcal{O}(1+\frac{1}{\varepsilon})$ recursion steps *packed sorting* will be used as the base case.
6. Permute the edges according to the results from the previous step in $\mathcal{O}(n)$.
7. A left-to-right scan through the trie now yields the sorted sequence of the input keys and takes $\mathcal{O}(n)$ time.

Therefore, if $w \geq \lg^{2+\varepsilon} n \lg\lg n$ for some fixed $\varepsilon > 0$, we can sort n w-bit integers in linear expeced time.

If we had chosen to recurse in the first place (on the chunks of the keys) the number of keys to recurse on would have been $n \cdot q$, while the "trie technique" was able to keep their number linear in n. By recursing on the original chunks, the running time would follow the recursion

$$SORT(n, w) = SORT(nq, \frac{w}{q}) = \mathcal{O}(n \cdot w),$$

which is even worse than radix sort!

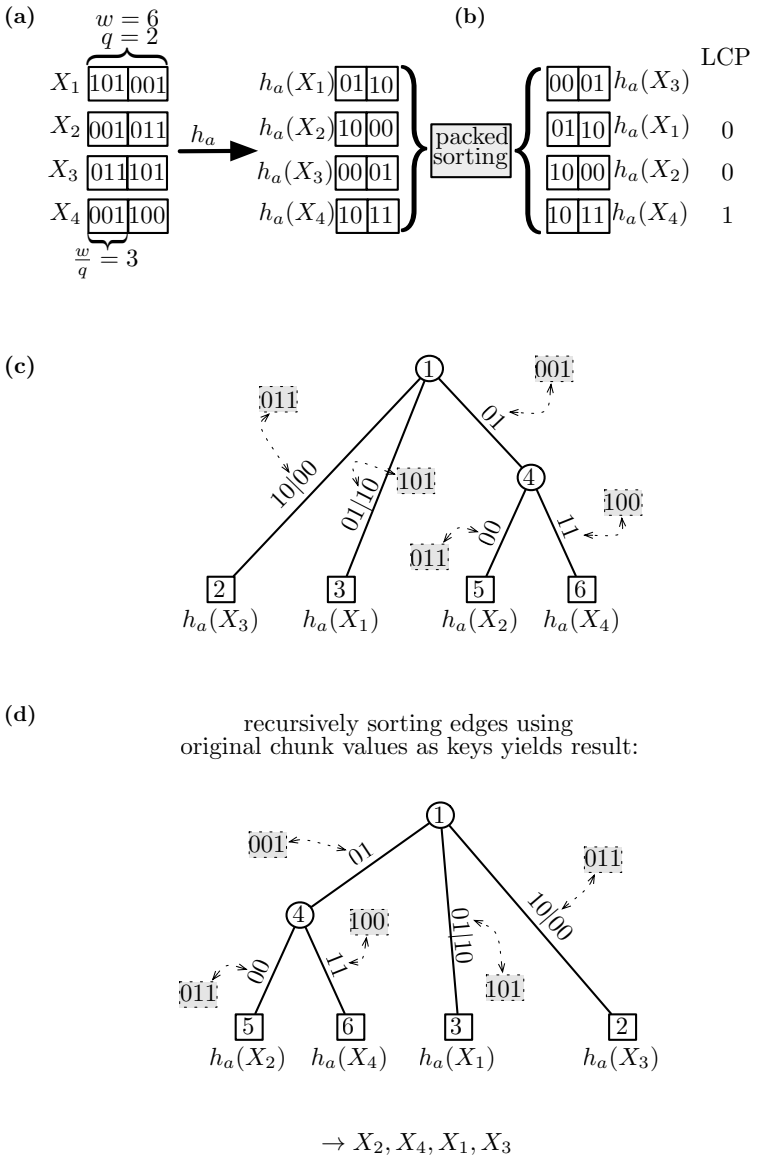

Figure 6: Example of signature sort steps: (a) Breaking keys into chunks and hash them to create signatures. (b) Use packed sorting to sort signatures. (c) Build trie based on sorted signatures. Grey boxes show the original chunk value corresponding to the signature chunk. (d) Recusivly sort edges of the tie using the original chunk values as sort keys. Afterwards, the sorted sequence of the input keys can be read off the trie.

References

[1] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, 7141(7141), 1995. URL `http://dl.acm.org/citation.cfm?id=225173`.

[2] Erik Demaine. Lecture Notes on Advanced Data Structures - Lecture 13: Integer Sorting. *Theoretical Computer Science*, pages 1–7, 2005. URL `http://courses.csail.mit.edu/6.897/spring05/lec/lec13.pdf`.

[3] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25: 19–51, October 1997. ISSN 0196-6774. doi: 10.1006/jagm.1997.0873. URL `http://dl.acm.org/citation.cfm?id=264631.264633`.

[4] Yijie Han and Mikkel Thorup. Integer sorting in O (n (log log n)) expected time and linear space. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 135–144. IEEE, 2002. ISBN 0-7695-1822-2. doi: 10.1109/SFCS.2002.1181890.

[5] David G. Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theor. Comput. Sci.*, 28:263–276, 1984.