1. Use the recursion to account for the fact that hashes might not preserve order.

   - recursion: stops after $O(\log \frac{1}{\delta}) = O(1)$ calls.
   - within key lengths $O(\log \sigma + \frac{w}{\log w})$.

   - use packed policy for base case.

2. Check if output sorted and restart if not $O(\log \sigma)$ times.

3. DO NOT CHECK INJECTIVITY IMMEDIATELY.
Example

Sort \((1, 011), (1, 101), (1, 001), (2, 011), (2, 100), (1, 001), (1, 101), (1, 011), (2, 100)\)

\[ \text{rebuild tree is} \]

\[ \text{final result} \]

\[ X_2 < X_4 < X_3 < X_1 \]

So why didn't we use recursion in the first place (on the chunks of the keys)? The reason is that there are too many bad chunks; \( n \geq \frac{w}{4} \). With the "tree" technique, their number linear in \( n \). With the original chunks, the running time would follow the recursion

\[ T(n, w) = T\left(n \frac{w}{4}, \frac{w}{4}\right) = O(n \cdot w) \]

Even worse than radix sort!

3.3 Packed Sorting

We now show how to sort \( \frac{b}{k} \)-bit keys in \( O(n) \) time, provided that the word size of the computer is at least \( 2(b+1)\log n \log \log n \). The idea is to pack \( k = 2(b+1)\log n \log \log n \) keys into one computer word, sort them with bit-wise in \( O(k) \) time, and then merge the \( \frac{n}{k} \) sorted lists in additional \( O(n) \) time.

\[ \frac{b}{k} \text{ bit (pack for sort)} \]
Suppose we have a black box for merging a pair of words consisting of \(k\) sorted elements each into one word consisting of \(2k\) sorted elements in \(O(\log k)\) time (instead of \(O(k)\) time with "normal" merging), and another black box for sorting \(k\) words of \(k\) elements in \(O(k)\) time (instead of \(k \log k\) with "normal" merge sort).

Then we can merge 2 sorted lists of \(r\) sorted words into one sorted list of \(2r\) sorted words in \(O(r \log k)\) time by:

1. merging the first words of each list in \(O(r)\)
2. outputting the first half of the result
3. replacing the word containing the maximum key with the second half of the result
4. removing the other merged word (not containing the max)
5. if both lists are nonempty, go to 1.
6. outputting all remaining words in order.

This algorithm spends \(O(\log k)\) time per word in the output, so the running time is \(O(r \log k)\).

Then we can also merge all \(r^k\) words in \(O(r)\) time by a standard merge-sort with the \(rlgk\) subroutine as a merger.

\[
\frac{r^k \cdot \log k}{\frac{r^k}{2} \cdot \log k} = \frac{r^k}{2^k} \cdot \frac{\log k}{\log 2k} = \frac{r^k}{2^k} \cdot \log_2 \frac{k}{2k} = \frac{r^k}{2^k} \cdot \log_2 \frac{1}{2} = \frac{r^k}{2^k} \cdot \log_2 \frac{1}{2} = \frac{r^k}{2^k} \cdot \frac{1}{2} = \frac{r^k}{2^{k+1}}
\]

Total: \(\frac{r^k}{2^{k+1}}\)
Here, the total time is \( O\left(\frac{\log n}{\log k} \cdot \frac{\log k}{k} + \frac{n}{k}\log k\right)\)
\(= O\left(\frac{n}{\log k} \log k \log n + n\right)\)
\(= O(n)\) with \(k = \log n \log n\).

3.4. **Bitonic Sorting**

The base case (sorting a word consisting of \(k\) keys) is handled with a bit-parallel version of bitonic sorting.

**Def.** A sequence \(x_1, \ldots, x_n\) is called bitonic if it is a cyclic shift of the concatenation of an increasing and a decreasing sequence.

\[3, 4, 5, 7, 6\]
\[5, 7, 6, 3, 4\]
\[5, 6, 7, 3, 4\]
\[6, 5, 7, 3, 4\]

Bitonic sequences are easy to sort in parallel:

**Procedure**: \(\text{sort}(x_1, \ldots, x_n)\):

For \(i = 1\) to \(n/2\)

If \((x_i > x_{i+n/2})\), swap \(x_i\) with \(x_{i+n/2}\)

Sort \((x_1, \ldots, x_{n/2})\) and \((x_{n/2+1}, \ldots, x_n)\) recursively in parallel

**Example**: \(x = 8, 3, 3, 4, 5, 7, 3, 8\)

```plaintext
1 3 3 4 5 7 3 8
5 3 3 4 5 7 3 8
3 3 5 4 8 7 5 8
3 3 4 5 7 3 8 9
```
The correctness of this algorithm follows from the following 2 facts:

1) both \( \min (x_1, x_{n/2+1}), \ldots, \min (x_{n/2}, x_n) = : L \)
   and \( \max (x_1, x_{n/2+1}), \ldots, \max (x_{n/2}, x_n) = : R \)
   are bitonic.

2) all elements in \( L \) are smaller than those in \( R \).

This can be proved by looking at all possibilities of where the half falls into, e.g.

The idea for sorting any sequences (not necessarily bitonic) is now to first move the bitonic part, sort the first half increasing and the second decreasing, and then use the bitonic-sort to finish.

(Note the running time of \( O(\log^2 n) \) in the sequential case.)
3.5. **Block Box 1: Merging Two Sorted Words**

We need to show how to merge two sorted words of \( k = \log n \) binary elements into one sorted word consisting of \( 2k \) elements. To adopt the ideas from bitonic sorting, we first reverse the first word (in a bit-parallel manner) and then concatenate the two words to get a bitonic sequence. This sequence can be sorted with a bit-parallel version of bitonic sorting — the result is the merged word!

To reverse the keys in a word, observe that \( \text{rev}(LR) = \text{rev}(R) \cdot \text{rev}(L) \). Hence we can do the following.

\[
\begin{align*}
\text{AND} & \quad \text{L} \quad \text{R} \\
\text{L} & \quad \text{R} \\
\text{L} \quad & = \quad \text{R} \\
\end{align*}
\]

and likewise with \( R \) to get

\[
\begin{align*}
\text{AND} & \quad \text{R} \quad \text{L} \\
\text{R} & \quad \text{L} \\
\text{R} \quad & = \quad \text{L} \\
\end{align*}
\]

Bit this can be done in parallel by

\[
\begin{align*}
\text{AND} & \quad \text{RL} \quad \text{LR} \quad \text{RR} \quad \text{LL} \\
\text{RL} & \quad \text{RR} \\
\text{RL} \quad & = \quad \text{RR} \\
\end{align*}
\]

And so on until all elements are reversed; this happens after \( \log \) \( k \) iterations.

We now come to the last step of the merging algorithm: the bit-parallel version of bitonic sorting.
To this end, separate the reverse keys from the first word by 1-bits, and the keys from the second word by 0-bits. Then a subtraction yields a parallel comparison (like with fusion trees):

\[
A = 1A-key1 1A-key2 1A-key3 1 \ldots 1A-key_k 0 \ldots 0
\]

\[
B = 0B-key1 0B-key2 0B-key3 0 \ldots 0B-key_k 0 \ldots 0
\]

\[
A - B = 1b 1b 1b 1b 1b 1b 1b \ldots 1b 1b
\]

\[
0 \iff B-key > A-key
\]

Now the aim is to construct a word containing exactly the smaller keys (and another one containing exactly the larger ones). That is, we need a mask \( M \) with 1's at exactly those positions where \( A - B \) has 1-separators, and \( \text{AND} M \) with \( B \). This gives those keys from \( B \) that are smaller than their counterparts in \( A \):

\[
(A-B) \oplus (A-B) \times B = M = 01001110110101000000000000000000 = \text{SMALL-B}
\]

\[
B \text{ AND } M = 010B-key2 \ldots 0B-key_3 \ldots 01 = \text{SMALL-B}
\]

For the smaller keys from \( A \), we do the following:

\[
A - B = 0 \ldots 11111 \ldots 0
\]

\[
M = 111110101010011111111
\]

\[
M' = 010110110110110110111
\]

\[
M'' = M' \text{ AND } M = 01100000001000000000000000 = \text{SMALL-A}
\]

\[
A \text{ AND } M'' = 0A-key1 01000000000000000000000000000000 = \text{SMALL-A}
\]

\[
\text{SMALL-A OR SMALL-B} = 0A-key1 0B-key2 0B-key_3 01 \ldots 0A-key_k = \text{SMALL-}\text{ALL}
\]

We perform a symmetric calculation to obtain a word LARGER.
containing those keys that are larger than their counter part.

For the next round, the goal is to compare the elements from SMALLS and LARGES separately:

Like with swapping, this can be done in parallel:

And so on. In total, we need $2^k$ rounds, each taking $O(1)$ time. Hence we can merge in $O(2^k)$ time.

3.6 Block Box 2: Sorting a Packed Word

We can simulate merge sort within a word:

Pressing on the last level worse like block box 1', but only on two words of half the size $= O(lg(k))$ time.
Likewise, traversing at any level can be implemented
like black box 1, with the appropriate bits
washed. Hence, the running time follows the recursion
\[ T(k) = O(g(k)) + 2T\left(\frac{k}{2}\right) \]
\[ = O(k). \]