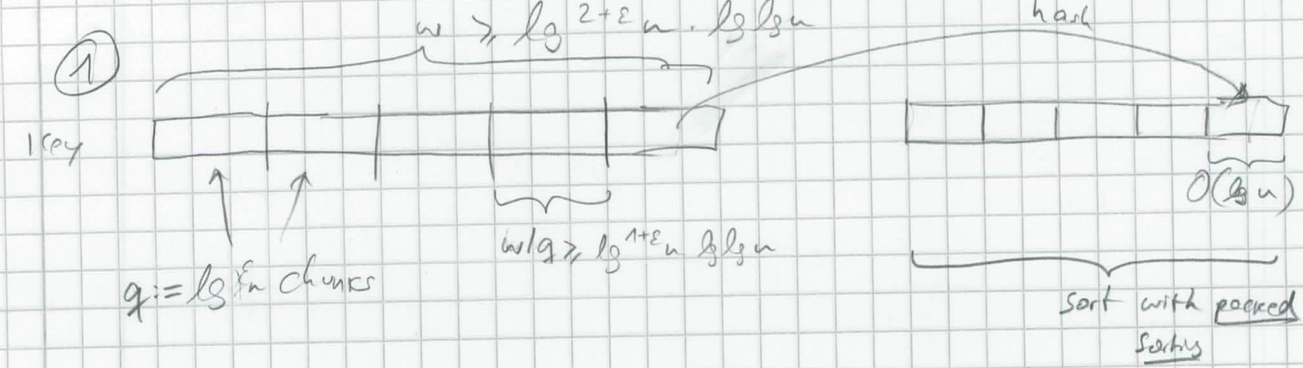
Zusammenfassung Teil 1 von Sign Sort:

(1)



$\Rightarrow$ hash nq elements to table of size $2^{O(\lg n)} = n^{O(1)} = m$

with univ. h.f. h

$$\mathbb{P}[h(x) = h(y)] \leq \frac{1}{m}$$

$$\Rightarrow \mathbb{P}[\text{single collision}] \leq \binom{nq}{2} \cdot \frac{1}{m} \leq \frac{n^2 q^2}{m}$$

$$\text{set } m \leftarrow n^4 \quad \Rightarrow \mathbb{P}[\text{single collision}] \leq \frac{n^2 \cdot \lg^{2\varepsilon} n}{n^4}$$

$$= \frac{\lg^{2\varepsilon} n}{n^2}$$

$$\Rightarrow \mathbb{P}[h \text{ injective}] \leq 1 - \mathbb{P}[\text{single collision}]$$

$$\approx 1 - \frac{1}{n^2} \quad (\text{"high"})$$

DO NOT CHECK INJECTIVITY IMMEDIATELY

(2) Use trie + recursion to account for the fact that hashes might not preserve order

recursion stops after $O(1 + \frac{1}{\varepsilon}) = O(1)$ calls with key lengths $O(\underbrace{\lg n}_{O(w)} + \underbrace{\frac{w}{\lg^{1+\varepsilon} n}}_{\leq \frac{w}{\lg n \lg\lg n}})$.

$\Rightarrow$ use Packed Sorting for base case

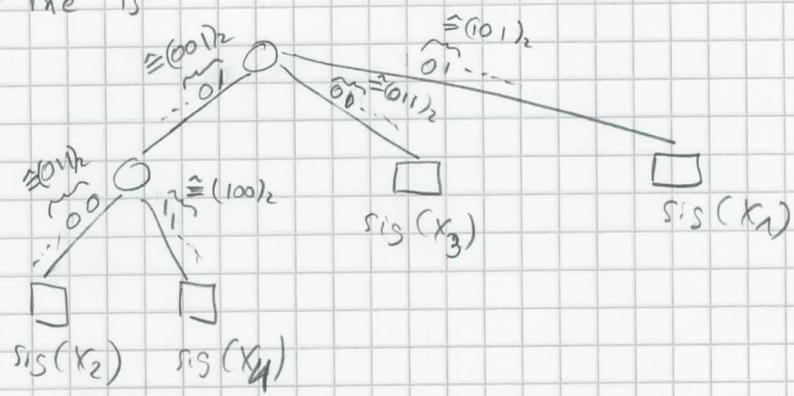
(3) check if output sorted and restart if not (only $O(1)$ trials).

⑥ **Example**

Sort $\overbrace{(1,011),\ (1,101),\ (1,001)}^{\text{root}},\ \overbrace{(2,011),\ (2,100)}^{\text{rightmost child of root}}$

$(1,001)\ ,\ (1,011)\quad (1,101)\quad (2,011)\quad (2,100)$

$\Rightarrow$ rebuilt trie is

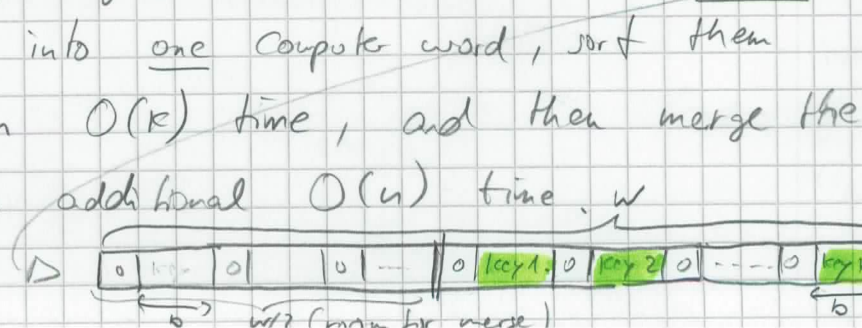


$\Rightarrow$ final result $\boxed{x_2 \leq x_4 \leq x_3 \leq x_1}$

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 such chunks: $n \cdot q$ many, whereas the "trie-technique" keeps their number linear in n. With the original chunks the running time would follow the recursion
$$T(n,w) = T\left(nq, \frac{w}{q}\right) = O(n \cdot w),$$
even worse than radix sort!

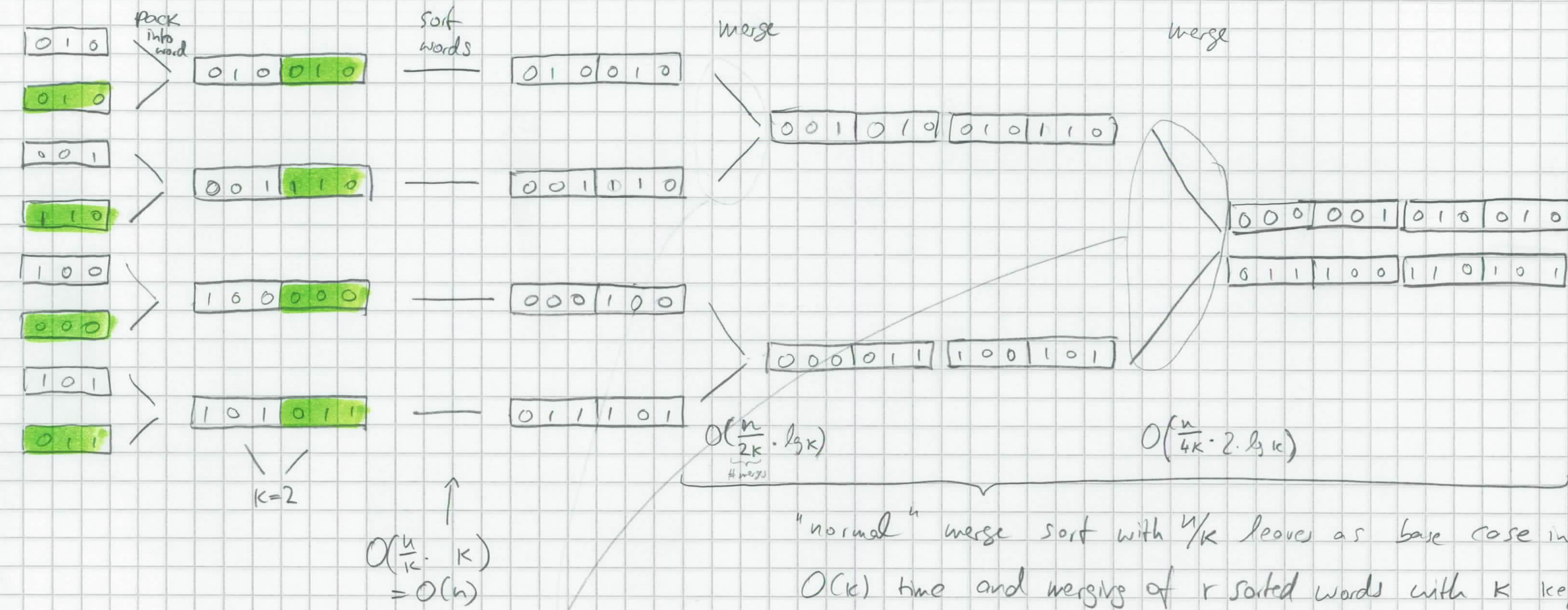
3.3 Packed Sorting

We now show how to sort n $\overbrace{\text{keys}}^{b\text{-bit}}$ in $O(n)$ time, provided that the word size of the computer is at least $2(b+1)\lg n \lg\lg n$. The idea is to <u>pack</u> $k := \lg n \lg\lg n$ keys into <u>one</u> computer word, sort them with bit-tricks in $O(k)$ time, and then merge the $\frac{n}{k}$ sorted lists in additional $O(n)$ time.



Example keys = 010, 010, 001, 010,

$b=2$, $n=8$, $k=2$



"normal" merge sort with n/k leaves as base case in $O(k)$ time and merging of r sorted words with k keys each in $O(r \lg k)$ time.

	1st list		2nd list		output	back to 1st/2nd list	
1) merge	001 010	with	000 011	⇒	000 001	010 011	(2nd)
2) merge	010 110	with	010 011	⇒	010 010	011 110	(1st
3) merge	011 110	with	100 101	⇒	011 100	110 101	(1st
4) merge	110 101	with	$\varnothing$	⇒	110 101		

7

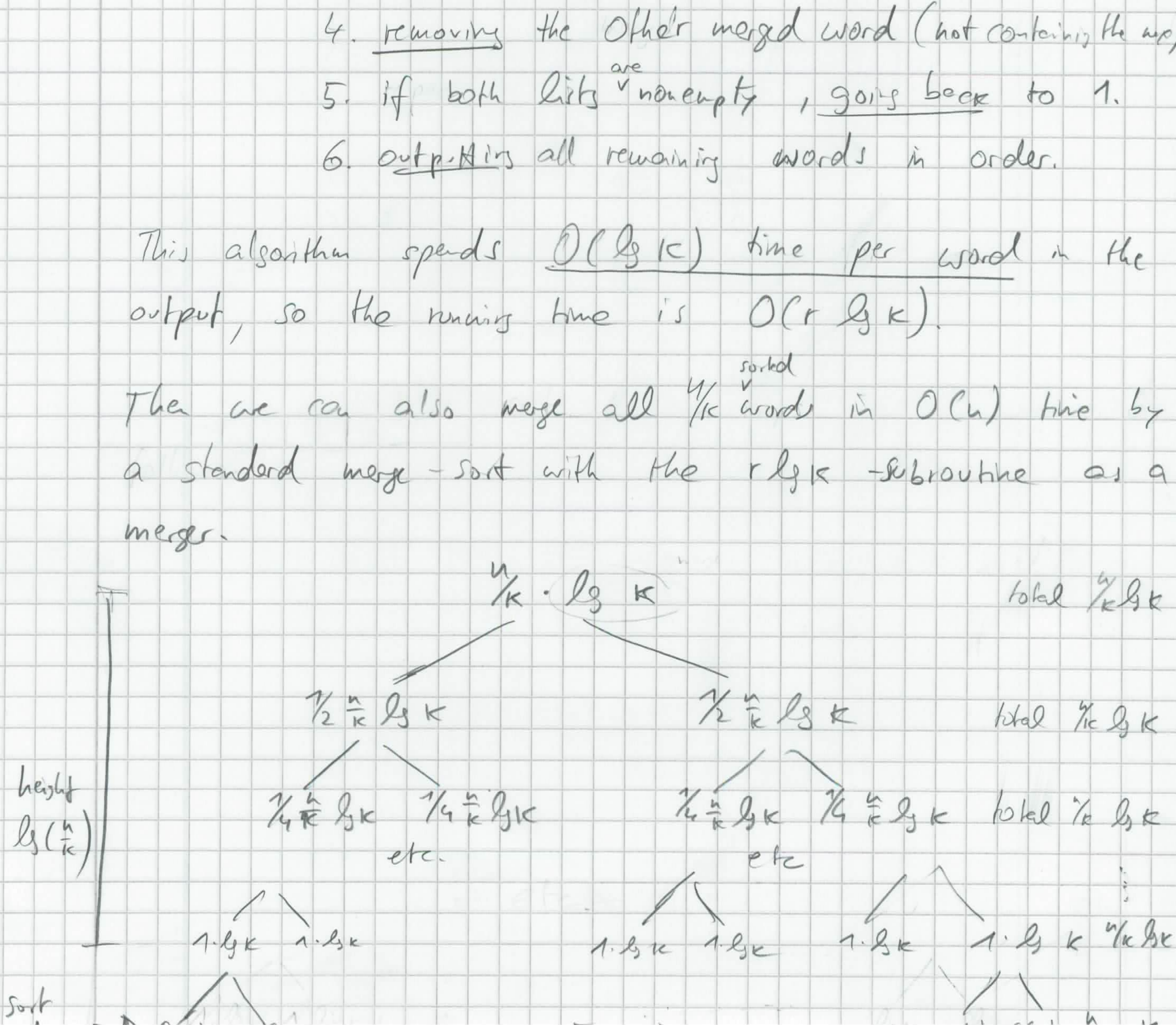
⑧ Suppose we have a black box for merging a pair of words consisting of k sorted elements each into one word consisting of $2k$ sorted elements in $O(\lg k)$ time (instead of $O(k)$ time with "normal" merging), and another black box for sorting 1 word of k elements in $O(k)$ time (instead of $k \lg k$ with "normal" merge sort).

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

1. merging the first words of each list in $O(\lg k)$
2. outputting the first half of the result
3. replacing the merged 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, going back to 1.
6. outputting all remaining words in order.

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

Then we can also merge all n/k sorted words in $O(n)$ time by a standard merge-sort with the $r \lg k$-subroutine as a merger.



Hence, the total time is $O(\underbrace{\lg \frac{n}{k}}_{\text{height}} \cdot \underbrace{\frac{n}{k} \lg k}_{\text{merge per level}} + \underbrace{\frac{n}{k} \cdot k}_{\text{sort leaves}})$

$= O(\frac{n}{k} \lg k \lg n + n)$

$= O(n)$ with $k = \lg n \lg \lg n$.

3.4. Bitonic Sorting

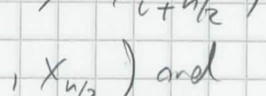
The base case (sorting a packed word consisting of k keys) is handled with a bit-parallel version of bitonic sorting.

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

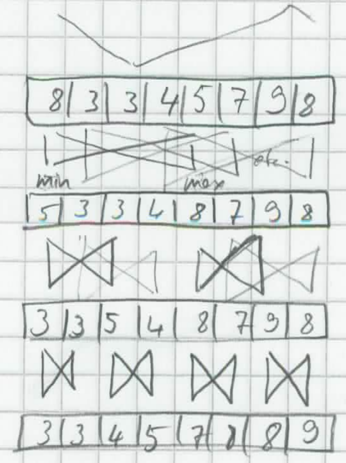
3 4 5 7 6	5 7 6 3 4	5 6 7 3 4	6 5 7 3 4
bitonic	bitonic	bitonic	not bitonic

Bitonic sequences are easy to sort in parallel:

procedure sort $(x_1, \dots, x_n)$:
  for $i = 1$ to $n/2$
    if $(x_i > x_{i+n/2})$ swap x_i with $x_{i+n/2}$
  sort $(x_1, \dots, x_{n/2})$ and $(x_{n/2+1}, \dots, x_n)$ recursively in parallel



Example: $x =$

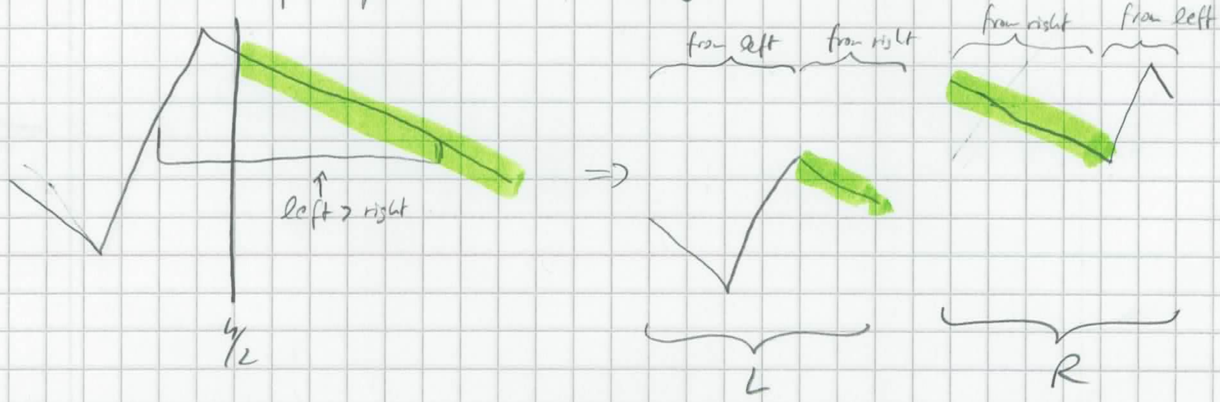


The correctness of this algorithm follows from the following 2 facts:

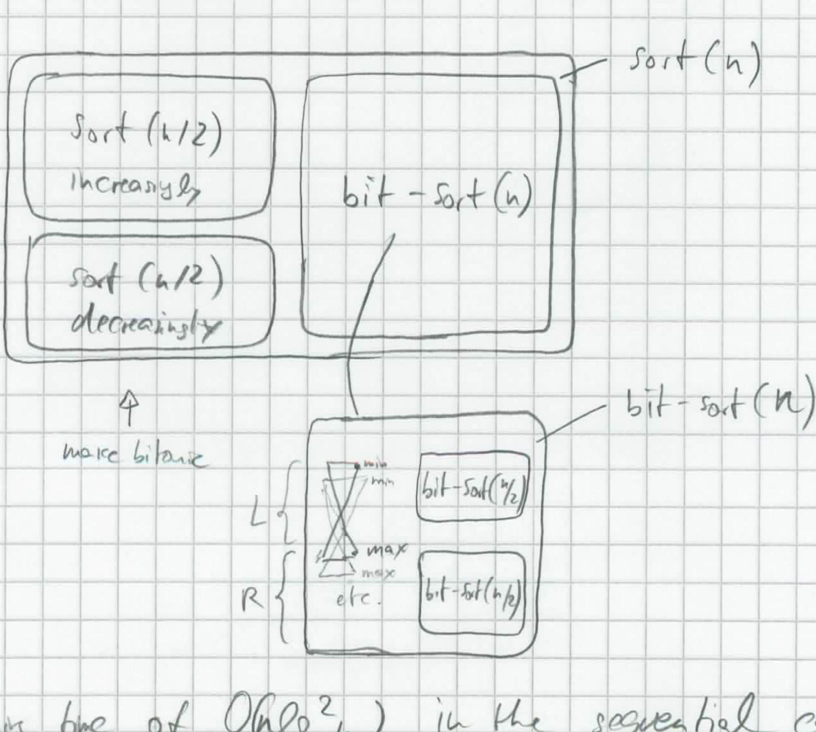
1) both $(\min(x_1, x_{n/2+1}), \ldots, \min(x_{n/2}, x_n)) =: L$
and $(\max(x_1, x_{n/2+1}), \ldots, \max(x_{n/2}, x_n)) =: R$
are bitonic

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 make them bitonic by sorting the first half increasingly and the second decreasingly, and then use the bitonic-sorter to finish.

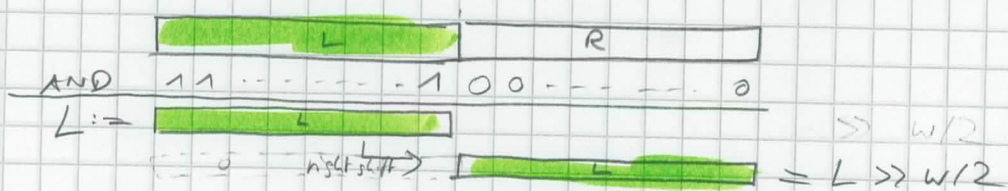


[Note the running time of $O(n \lg^2 n)$ in the sequential case.]

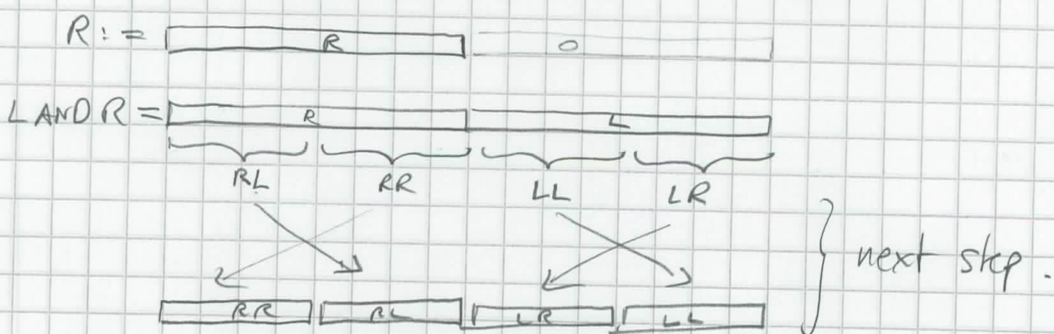
3.5. Black Box 1: Merging Two Sorted Words.

We need to show how to merge two sorted words of $k = \lg n \lg\lg n$ elements into one sorted word consisting of $2k$ elements. To adapt the ideas from bitonic sorting, we first reverse the first word (in a bit-parallel way) 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!

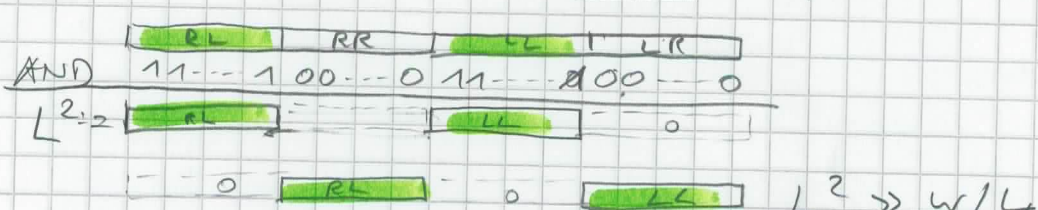
To reverse the keys in a word, observe that $rev(LR) = rev(R) \circ rev(L)$. Hence we can do the following.



and likewise with R to get



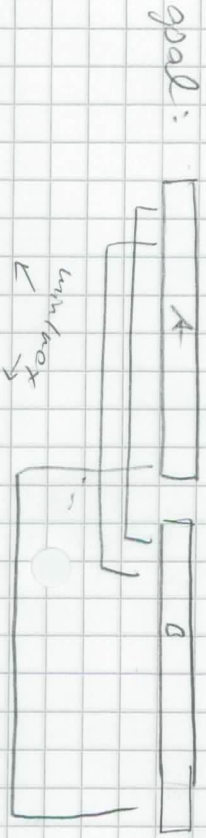
But this can be done in parallel by



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

We now come to the last step of the merging algorithm: the bit-parallel version of bitonic sorting.

(12) To this end, separate the reversed keys from the first word A by 1-bits, and the keys from the second word B by 0-bits. Then a subtraction yields a parallel comparison (like with fusion trees):



$$\overbrace{\hspace{6em}}^{w/2}\qquad\overbrace{\hspace{2em}}^{w/2}$$

$A =$	1	A-key 1	1	A-key 2	1	A-key 3	1	----	1	A-key k	0 ---- 0
$B =$	0	B-key 1	0	B-key 2	0	B-key 3	0	---	0	B-key k	0 ---- 0
$A-B =$	1/0	???	1/0	???	1/0	???	1/0	----	1/0	???	------

(B-key 1 spans b bits.)

↑ 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 AND M with B. This gives those keys from B that are smaller than their counterparts in A:

$A-B =$	0	- - -	1	- - -	1	- - -	1/0	- - -	0	- - -	
$(A-B)-[(A-B)\gg(b-1)] = M =$	0	0 - - - 0	0	1 1 - - - 1	0	1 - - - 1	0	- - -	0	0 - - - 0	$=$
B AND $M =$	0		0	B-key 2	0	B-key 3	0	- - -	0		$=:$ SMALL-B

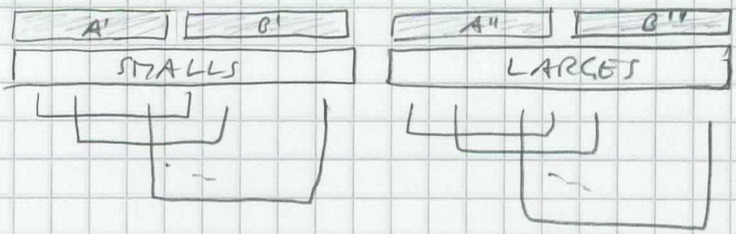
For the smaller keys from A, we do the following:

$A-B =$	0	------	1		1	------	1/0		0	-	
$\neg M =$	1	1 - - - 1	1	0 - - - 0	1	0 - - - 0	1	- - -	1	1 - - - 1	
$M' =$	0	1 - - - 1	0	1 - - - 1	0	1 - - - 1	0	- - -	0	1 - - - 1	
$M'' = \neg M$ AND $M' =$	0	1 - - - 1	0	0 - - - 0	0	0 - - - 0	0	- - -	0	1 - - - 1	
A AND $M'' =$	0	A-key 1	0		0		0	- - -	0	A-key k	$=:$ SMALL-A

SMALL-A OR SMALL-B $=$	0	A-key 1	0	B-key 2	0	B-key 3	0	---	0	A-key k	$=:$ SMALLS

We perform an symmetric calculation to obtain a word LARGES containing those keys that are larger than their counterpart.

For the <u>next round</u>, the goal is to compare the elements from SMALLS and LARGES seperately:



Like with swapping, this can be done <u>in parallel</u>:

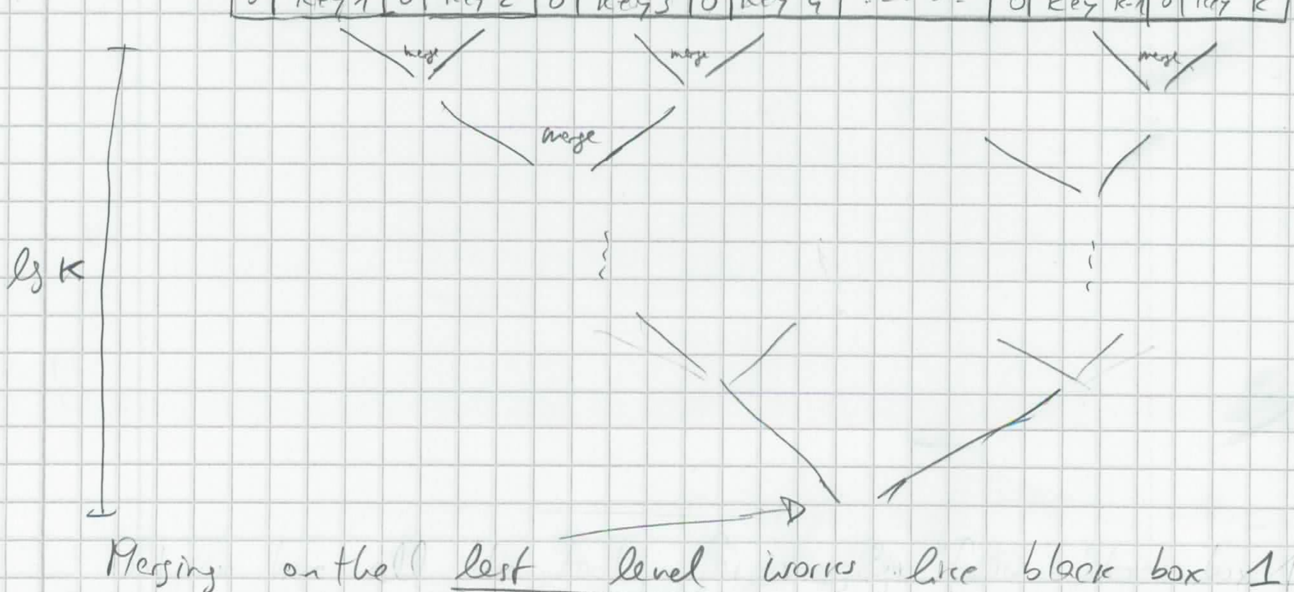
	w/4					w/4	w/4					w/4
mask & set 1's:	1	a'-key 1	---	1	a'-key k/2	0 ---- 0	1	a''-key 1	---	1	a''-key k/2	0 ---- 0
mask & shift:	0	b'-key 1	---	0	b'-key k/2	0 --- 0	0	b''-key 1	---	0	b''-key k/2	0 --- 0
subtract:	1/0	???	--	1/0	???	0 --- 0	1/0	???	--	1/0	???	0 --- 0

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

3.6 Black Box 2: Sorting a Packed Word

We can <u>simulate merge sort</u> within a word:

w/2												
0	key 1	0	key 2	0	key 3	0	key 4	------	0	key k-1	0	key k



Merging on the <u>last level</u> works like black box 1, but only on two words of half the size $\Rightarrow O(\lg(k))$ time.

(14) Likewise, merging on any level can be implemented like black box 1, with the appropriate bits masked. Hence, the running time follows the recursion

$$T(k) = O(\lg k) + 2\,T(k/2)$$
$$= O(k).$$