Integer Sorting
on the word-RAM

Uri Zwick
Tel Aviv University

May 2015

Last updated: June 30, 2015

Integer sorting

Memory is composed of w-bit words.

Arithmetical, logical and shift operations on w-bit words take $O(1)$ time.

How fast can we sort an array of length n?

Comparison based algorithms

$\Theta(n \log n)$

Time bounds dependent on w

$\Theta(n \log w)$ - van Emde Boas trees

Time bounds independent of w

$O(n \log n / \log\log n)$ - Fredman-Willard (1993)

$O(n \log\log n)$ - Andersson et al. (1998)

$O\left(n\sqrt{\log\log n}\right)$ - Han-Thorup (2002)

Some of these algorithms are *randomized*

Fundamental open problem

Can we sort in $O(n)$ time,
for any $w \geq \log n$???

Sorting – three variants

$Sort(n,w,b)$ –
Return a *permutation* that (stably) sorts
n b-bit keys, each stored in a separate word,
on a machine with w-bit words.

$Sort'(n,w,b)$ –
(Stably) sort n w-bit words according to b-bit keys.
(The keys are the first/last b bits of each word.)

$Sort''(n,w,b)$ –
Sort n b-bit keys, each stored in a separate word,
on a machine with w-bit words.

(We will not be very precise in using these names…)

Sorting – three variants

Exercise: Reduce $Sort''(n, w, b)$ to $Sort'(n, w, b)$ with no extra work.

Exercise: Reduce $Sort'(n, w, b)$ to $Sort(n, w, b)$ with only $O(n)$ extra work.

Exercise: Reduce $Sort(n, w, b)$ to $Sort''(n, w, b + \log n)$ with only $O(n)$ extra work.

Exercise: Reduce $Sort(n, w, b)$ to $Sort''(n, w, b)$ with only $O(n)$ extra work.
(Hint: hashing.)

Two techniques

Range reduction

Reduce $Sort(n, w, b)$ to $Sort(n, w, b/2)$ using only $O(n)$ extra work.

Packed sorting

Solve $Sort''(n, w, b)$ by packing w/b keys in each word. In $O(1)$ time we can perform simple operations on w/b keys.

(Word-level parallelism)

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

2	8	7	1
4	5	9	1
6	5	7	2
1	3	0	1
2	4	7	2
3	5	5	5
7	0	2	2
8	3	9	4
4	8	4	4
3	5	3	6

2	8	7	1
4	5	9	1
1	3	0	1
6	5	7	2
2	4	7	2
7	0	2	2
8	3	9	4
4	8	4	4
3	5	5	5
3	5	3	6

To sort according to a “digit” use bucket or count sort.

Slides from undergrad course

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

2	8	7	1
4	5	9	1
1	3	0	1
6	5	7	2
2	4	7	2
7	0	2	2
8	3	9	4
4	8	4	4
3	5	5	5
3	5	3	6

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

After the i-th pass, numbers are sorted according to the *least* significant i digits

2	8	7	1
4	5	9	1
1	3	0	1
6	5	7	2
2	4	7	2
7	0	2	2
8	3	9	4
4	8	4	4
3	5	5	5
3	5	3	6

1	3	0	1
7	0	2	2
3	5	3	6
4	8	4	4
3	5	5	5
2	8	7	1
6	5	7	2
2	4	7	2
4	5	9	1
8	3	9	4

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

1	3	0	1
7	0	2	2
3	5	3	6
4	8	4	4
3	5	5	5
2	8	7	1
6	5	7	2
2	4	7	2
4	5	9	1
8	3	9	4

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

1	3	0	1
7	0	2	2
3	5	3	6
4	8	4	4
3	5	5	5
2	8	7	1
6	5	7	2
2	4	7	2
4	5	9	1
8	3	9	4

7	0	2	2
1	3	0	1
8	3	9	4
2	4	7	2
3	5	3	6
3	5	5	5
6	5	7	2
4	5	9	1
4	8	4	4
2	8	7	1

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

7	0	2	2
1	3	0	1
8	3	9	4
2	4	7	2
3	5	3	6
3	5	5	5
6	5	7	2
4	5	9	1
4	8	4	4
2	8	7	1

Backward/LSD Radix sort

Stably sort according to “digits”.
Starting from least significant digit.

7	0	2	2
1	3	0	1
8	3	9	4
2	4	7	2
3	5	3	6
3	5	5	5
6	5	7	2
4	5	9	1
4	8	4	4
2	8	7	1

1	3	0	1
2	4	7	2
2	8	7	1
3	5	3	6
3	5	5	5
4	5	9	1
4	8	4	4
6	5	7	2
7	0	2	2
8	3	9	4

Function `count-sort`(A, B, n, R)

for $i \leftarrow 0$ `to` $R - 1$ **do**

$\quad C[i] \leftarrow 0$

for $j \leftarrow 0$ `to` $n - 1$ **do**

$\quad C[A[j].key] \leftarrow C[A[j].key] + 1$

for $i \leftarrow 1$ `to` $R - 1$ **do**

$\quad C[i] \leftarrow C[i] + C[i - 1]$

for $j \leftarrow n - 1$ `downto` 0 **do**

$\quad C[A[j].key] \leftarrow C[A[j].key] - 1$

$\quad B[C[A[j].key]] \leftarrow A[j]$

(Adapted from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, Third Edition, 2009, p. 195)

Backward/LSD Radix Sort
in the word-RAM model

Sort according to *least significant* $\log n$ bits.

Stably sort according to remaining $w - \log n$ bits.

Least significant $\log n$ bits can be sorted in $O(n)$ time using bucket sort or count sort.

Total running time is $O\left(n \cdot \frac{w}{\log n}\right)$.

Running time is $O(n)$ if $w = O(\log n)$.

We shall revisit Radix Sort later…

“Ultra” Radix Sort
[Kirkpatrick-Reisch (1984)]

First attempt:

Split each w-bit word into two $w/2$-bit parts.

Sort according to the *low* part.

Scan the array sequentially and append each item into a bucket indexed by *high*.

(Each bucket is sorted.)

Use a hash table to maintain the non-empty buckets.

Sort the indices of the non-empty buckets.

Go over the non-empty buckets, in sorted order, and concatenate the sorted lists of the buckets.

“Ultra” Radix Sort
[Kirkpatrick-Reisch (1984)]

First attempt:

The algorithm is correct.

But, to sort an array of w-bit words, we need to sort two arrays of the same length with $w/2$-bit words.

We cannot do that more than a constant number of times.

To fix the problem we use a trick similar to the one used in van Emde Boas trees.

“Ultra” Radix Sort
[Kirkpatrick-Reisch (1984)]

Working version:

Scan the items in arbitrary order and
throw them into buckets according to $high$.

In each bucket, indexed by $high$, keep only the item(s) with the smallest low value found so far.

Put all non-minimal items in a list L.

For each non-empty bucket, add $(0, high)$ to L.

(The length of the L is at most n.)
($(0, high)$ replaces the minimal item in bucket $high$.)

"Ultra" Radix Sort
[Kirkpatrick-Reisch (1984)]

Working version (continued):

Sort L according to low.

Extract from L a sorted list of the non-empty buckets.
(When we encounter the first $(0, high)$ pair,
check whether bucket $high$ is non-empty.)

Append each remaining item
$(high, low)$ in L into bucket $high$.

Use the sorted list of non-empty buckets to
concatenate the buckets in the appropriate order.

Use hashing to maintain the buckets in both phases.

“Ultra” Radix Sort

[Kirkpatrick-Reisch (1984)]

Complexity

$T(n,w)$ – time to sort n items according to a w-bit key

$$T(n,w) = T\left(n, \frac{w}{2}\right) + O(n)$$

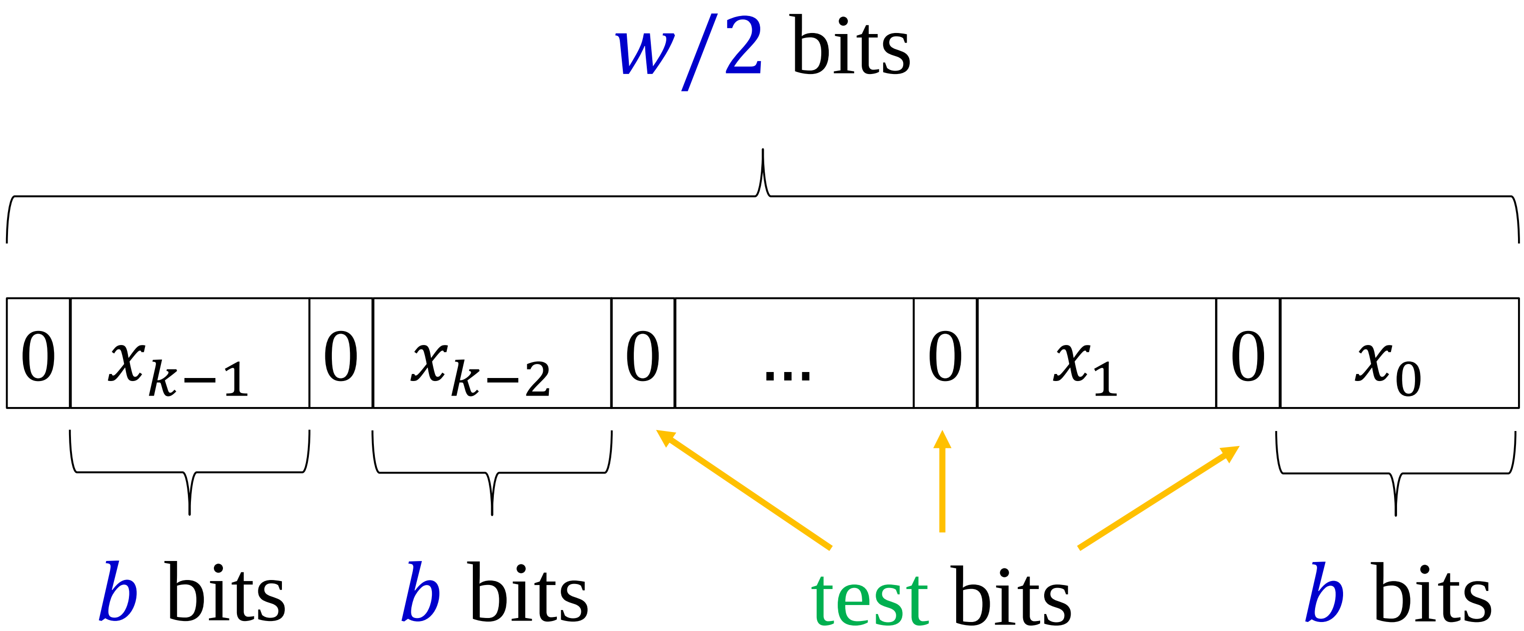
$$T(n,w) = O(n \log w)$$

Matching van Emde Boas trees…

In $O(n \log\log n)$ time, we can reduce w to $\mathrm{w}/(\log n)^2$.

We can then pack $\log^2 n$ keys is each word!

Packed representation



We can easily take two such words and produce a word that contains $2k$ keys.

Packed representation

Useful constants:

w bits

C_1	1	00…0	1	00…0	1	...	1	00…0	1	00…0
C_2	0	$2k-1$	0	$2k-2$	0	...	0	1	0	0

Exercise: How quickly can we construct these constants?

Packed representation

Useful operation: $CopyTestBits$

w bits

1	00…0	1	00…0	1	...	0	00…0	1	00…0

0	11…1	0	11…1	0	...	0	00…0	0	11…1

$$CopyTestBits(A) \equiv A - (A \gg b)$$

Packed Sorting

[Paul-Simon (1980)] [Albers-Hagerup (1997)]

$Sort''(n, w, b)$ –

Sort n b-bit keys on a machine with w-bit words.

Partition the items into groups of size $k = \left\lfloor \frac{w}{2(b+2)} \right\rfloor$.

Sort each group naïvely.

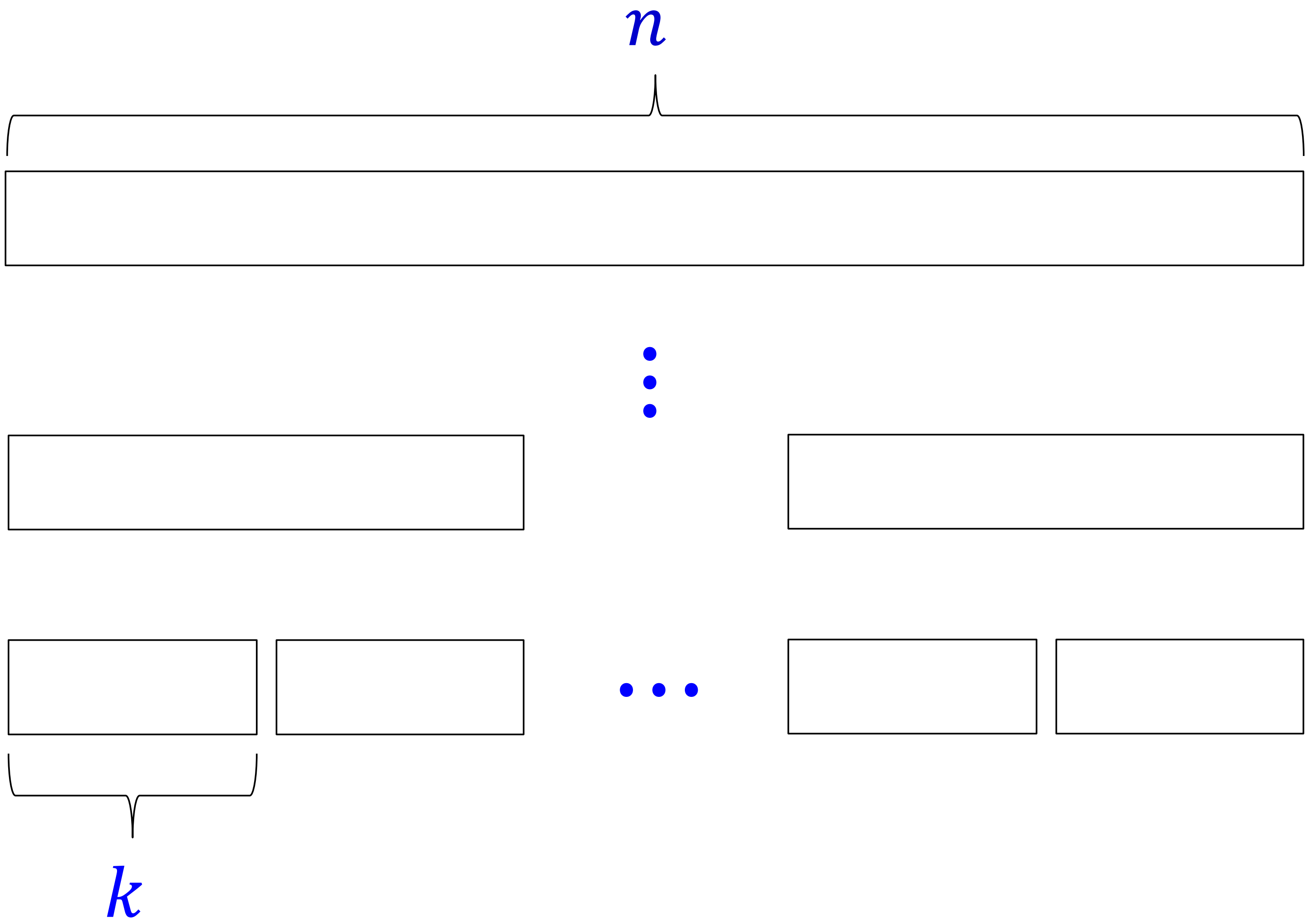
Pack each group into a single word.

Time required for this preliminary step is

$$O\left(\frac{n}{k} \cdot k \log k\right) = O(n \log k) .$$

This is $O(n \log \log n)$, if $k = (\log n)^c$.

(Packed) Merge Sort



Packed Merge Sort

[Paul-Simon (1980)] [Albers-Hagerup (1997)]

Merge packed sorted sequences of length k to sorted sequences of length $2k$, and then of length $4k$, etc., until a single sorted sequence of length n is obtained.

As a basic operation, use the merging of two sorted sequences of length k.

We shall implement this basic operation in $O(\log k)$ time, by simulating a bitonic sorting network.

Packed Merge Sort

[Paul-Simon (1980)] [Albers-Hagerup (1997)]

Standard merge sort take $O(n \log n)$ time.

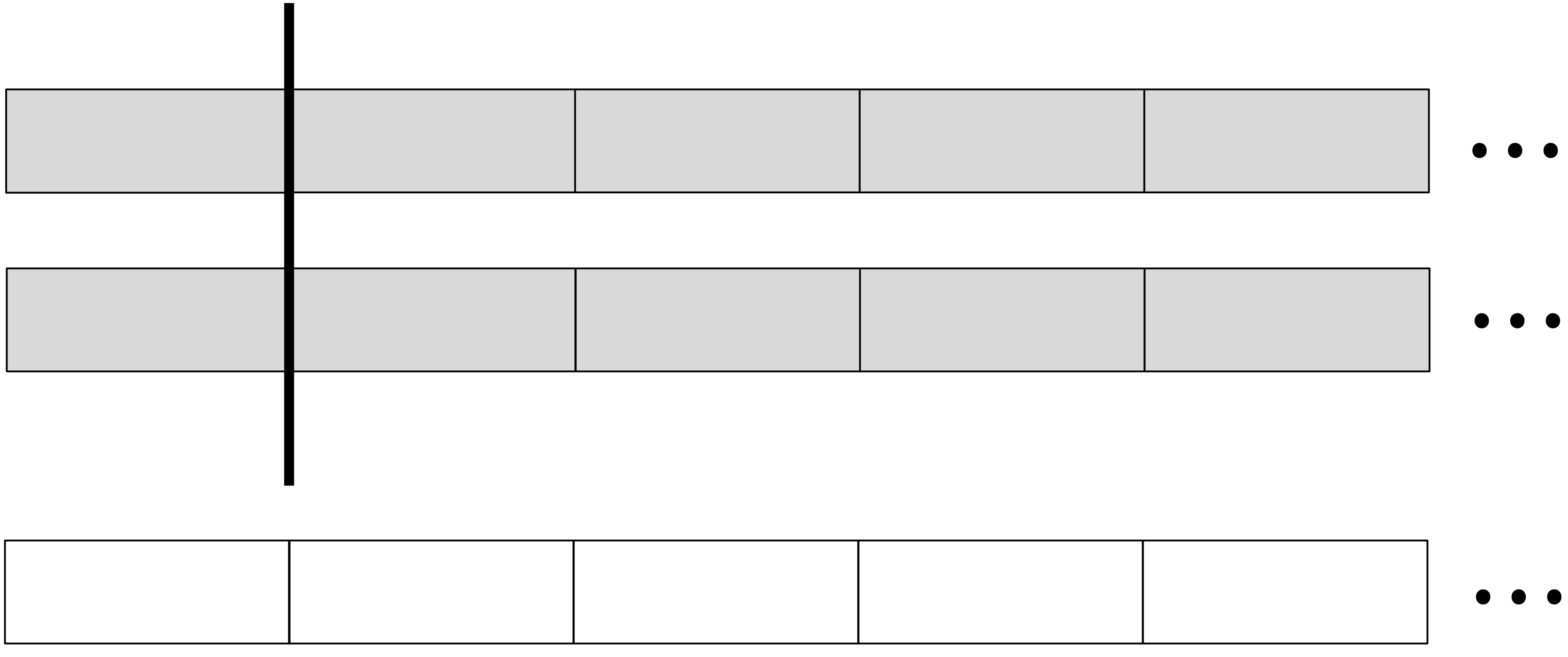
We save a factor of $O(k/\log k)$.

Thus, the running time is $O\left(\frac{n \log n \log k}{k}\right)$.

For $k = \log^2 n$, the running time is $o(n)$.

Packed Merge Sort

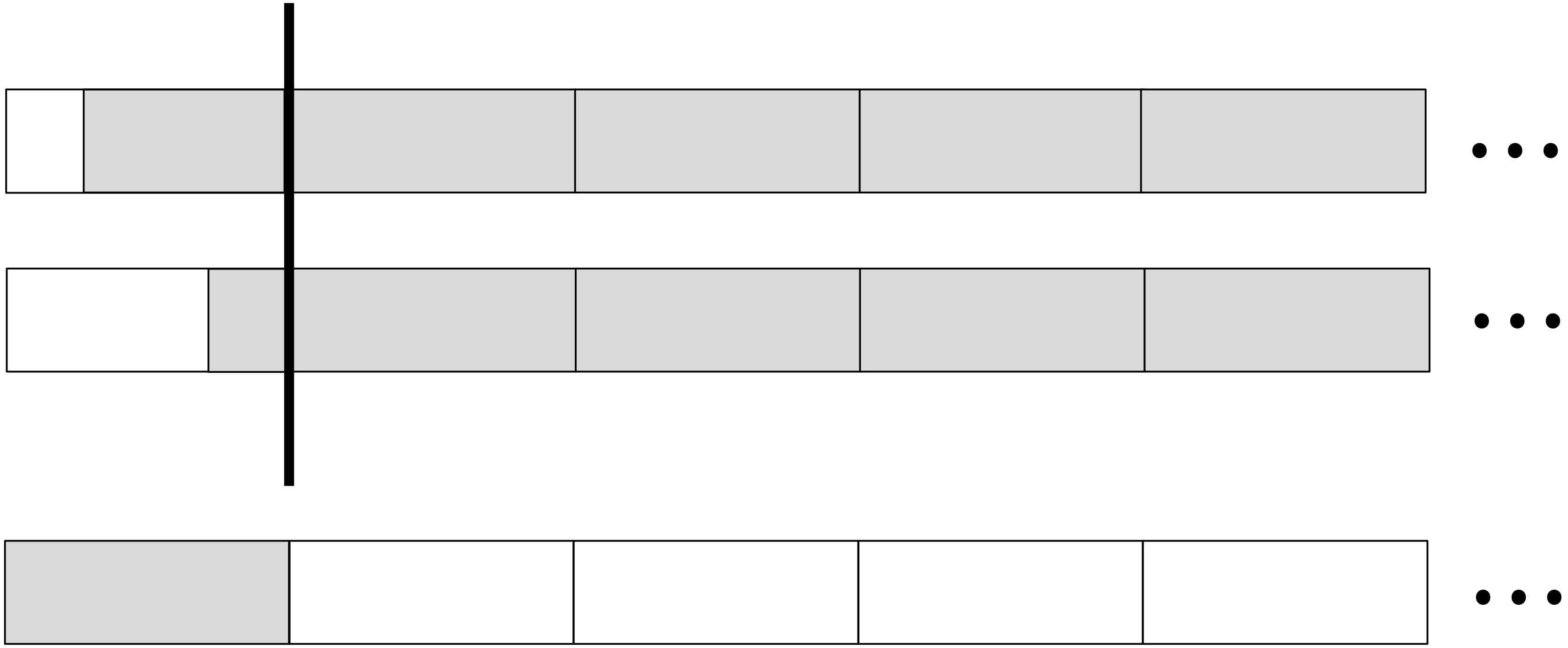
[Paul-Simon (1980)] [Albers-Hagerup (1997)]



Merge the smallest k items from both sequences.
The smallest k items go to the output sequence.

Packed Merge Sort

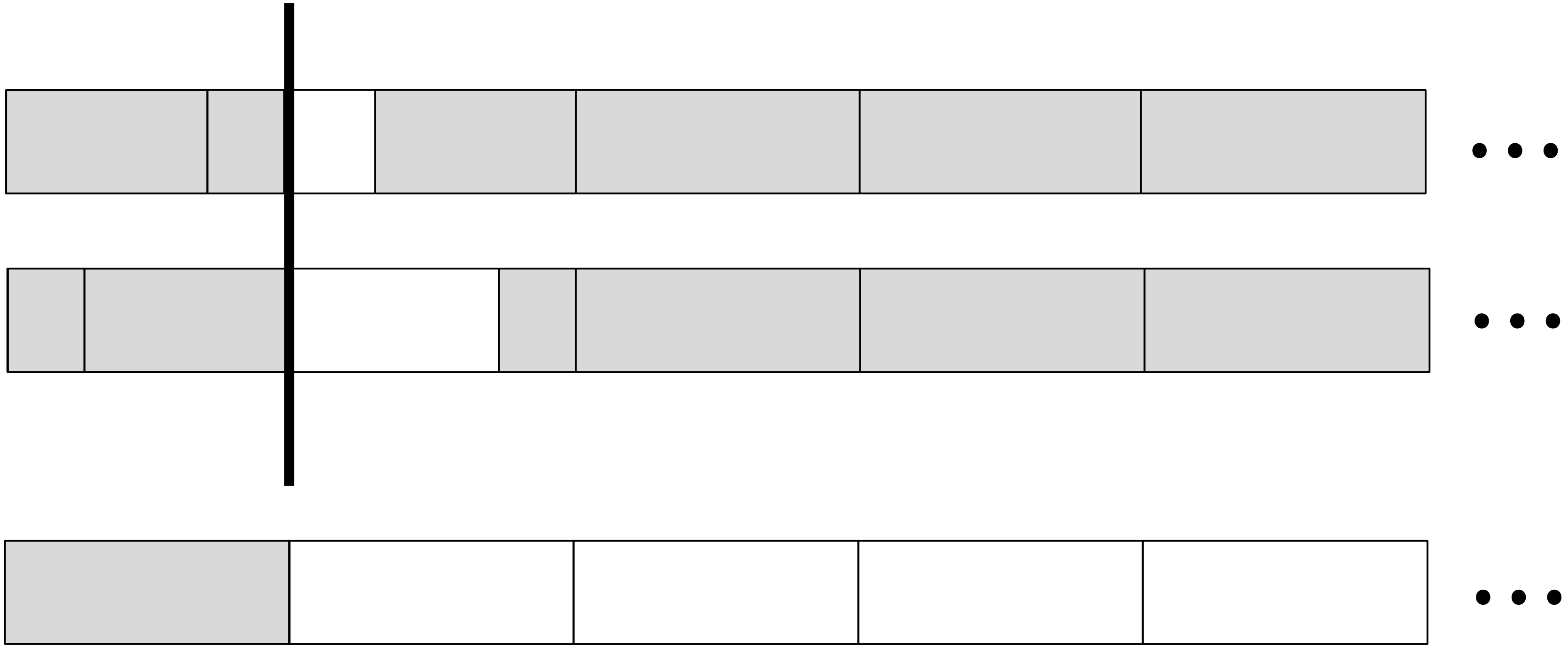
[Paul-Simon (1980)] [Albers-Hagerup (1997)]



Merge the smallest k items from both sequences.
The smallest k items go to the output sequence.

Packed Merge Sort

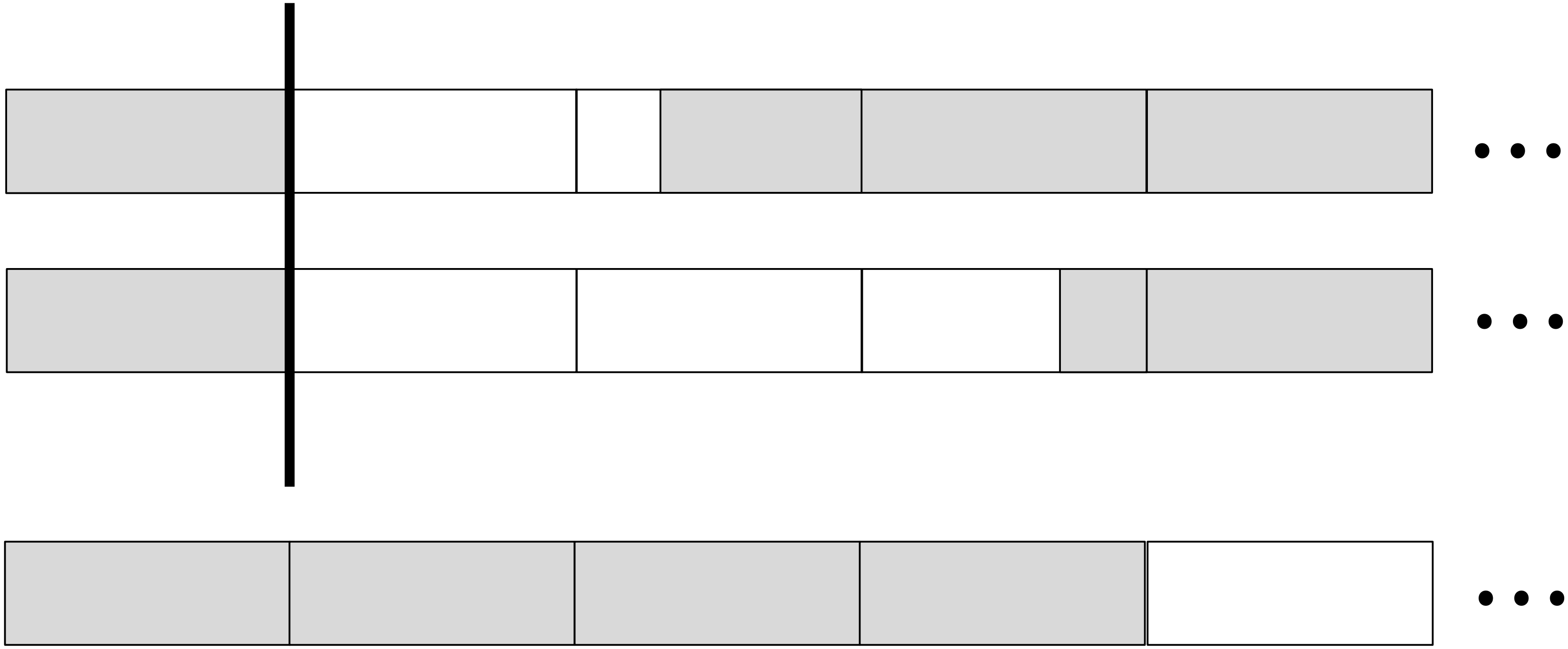
[Paul-Simon (1980)] [Albers-Hagerup (1997)]



Merge the smallest k items from both sequences.
The smallest k items go to the output sequence.

Packed Merge Sort

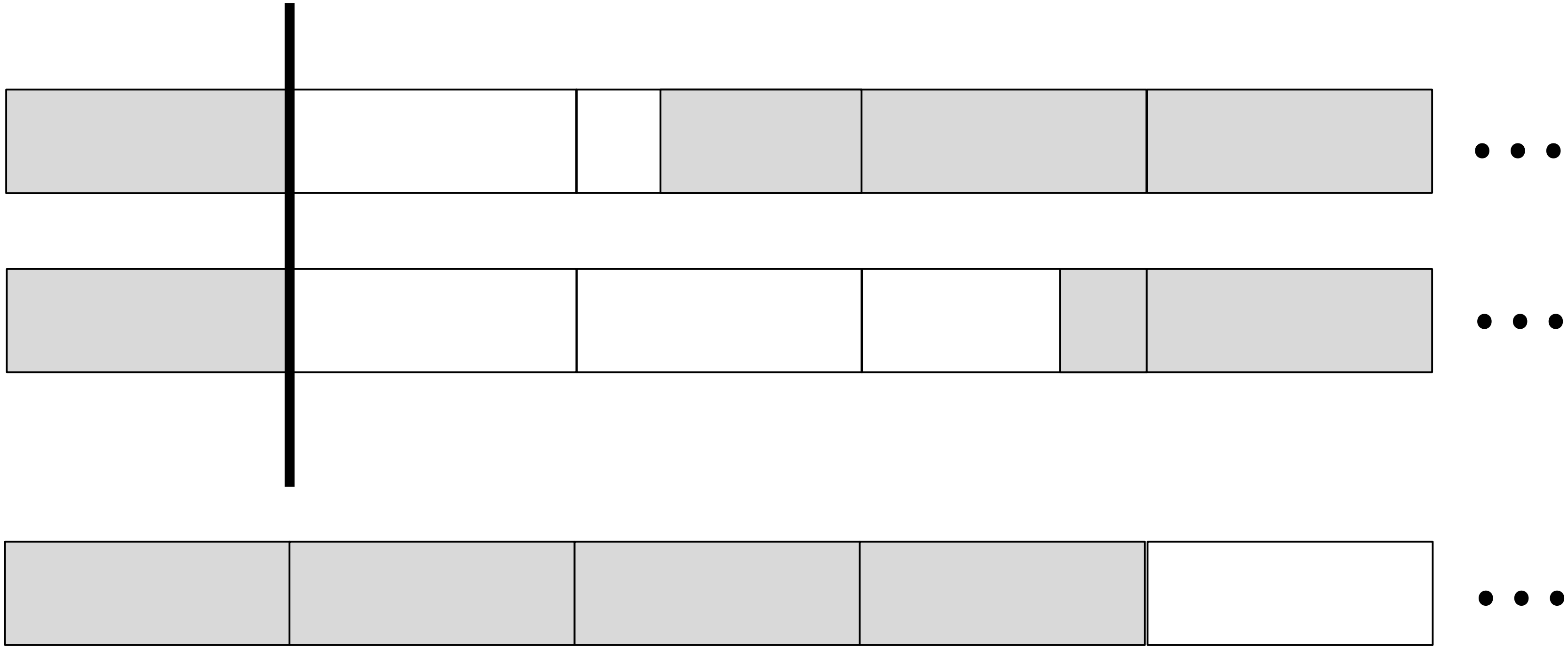
[Paul-Simon (1980)] [Albers-Hagerup (1997)]



Small technical detail: We need to know how many of the k smallest items came from each sequence.

Packed Merge Sort

[Paul-Simon (1980)] [Albers-Hagerup (1997)]



Simple solution: Add a bit to each key, telling where it is coming form. Count number of keys coming from each sequence. (But, how do we count?)

Batcher's bitonic sort

To merge two packed sorted sequence of k keys each, we use Batcher's bitonic sort.

We need to *reverse* one of the sequences and *concatenate* it to the other sequence.

Suppose that k is a power of 2.

Bitonic sort is composed of $1 + \log k$ iterations.

In iteration $i = \log k, \ldots, 0$, we need to compare/swap items whose indices differ only in their i-th bit.

One step of bitonic sort (1)

Compare/swap items that whose indices differ in the i-th bit
(In the example $i = 1$.)

0	x_7	0	x_6	0	x_5	0	x_4	0	x_3	0	x_2	0	x_1	0	x_0

Extract items whose indices have a 1 in their i-th bit,
and items whose indices have a 0 in their i-th bit

0	x_7	0	x_6	0	0000	0	0000	0	x_3	0	x_2	0	0000	0	0000

0	0000	0	0000	0	x_5	0	x_4	0	0000	0	0000	0	x_1	0	x_0

One step of bitonic sort (2)

Shift the first word 2^i fields to the right, and set their test bits to 1

0	x_7	0	x_6	0	0000	0	0000	0	x_3	0	x_2	0	0000	0	0000

0	0000	0	0000	0	x_5	0	x_4	0	0000	0	0000	0	x_1	0	x_0

One step of bitonic sort (3)

Shift the first work 2^i fields to the right,
and set their test bits to 1

1	x_7	1	x_6	1	0000	1	0000	1	x_3	1	x_2

0	0000	0	0000	0	x_5	0	x_4	0	0000	0	0000	0	x_1	0	x_0

Subtract

0	0000	0	0000	1	...	0	...	0	0000	0	0000	0	...	1	...

$x_5 \leq x_7$ $x_4 > x_6$ $x_1 > x_3$ $x_0 \leq x_2$

One step of bitonic sort (4)

1	x_7	1	x_6	1	0000	1	0000	1	x_3	1	x_2

0	0000	0	0000	0	x_5	0	x_4	0	0000	0	0000	0	x_1	0	x_0

Subtract

0	0000	0	0000	1	...	0	...	0	0000	0	0000	0	...	1	...

$x_5 \leq x_7$ $x_4 > x_6$

$x_1 > x_3$ $x_0 \leq x_2$

Collect winners and losers

0	0000	0	0000	0	x_7	0	x_4	0	0000	0	0000	0	x_1	0	x_2

0	0000	0	0000	0	x_5	0	x_6	0	0000	0	0000	0	x_3	0	x_0

One step of bitonic sort (5)

Shift the winners 2^i fields to the left

0	0000	0	0000	0	x_7	0	x_4	0	0000	0	0000	0	x_1	0	x_2

0	0000	0	0000	0	x_5	0	x_6	0	0000	0	0000	0	x_3	0	x_0

One step of bitonic sort (6)

Shift the winners 2^i fields to the left

0	x_7	0	x_4	0	0000	0	0000	0	x_1	0	x_2

0	0000	0	0000	0	x_5	0	x_6	0	0000	0	0000	0	x_3	0	x_0

Combine them together again:

0	x_7	0	x_4	0	x_5	0	x_6	0	x_1	0	x_2	0	x_3	0	x_0

The i-th step is over!

Packed Bitonic Sort
[Albers-Hagerup (1997)]

$Z \leftarrow X \vee Reverse(Y)$

for $i \leftarrow \log k$ downto 1:

$$M \leftarrow CopyTestBits\Big(\big(C_2 \ll (b - i)\big) \wedge C_1\Big)$$
$$A \leftarrow (Z \wedge M) \gg (2^i(b + 1))$$
$$B \leftarrow Z - (Z \wedge M)$$
$$M' \leftarrow CopyTestBits\Big(\big((A \vee C_1) - B\big) \wedge C_1\Big)$$
$$C \leftarrow (B \wedge M') \vee \big(A - (A \wedge M')\big)$$
$$D \leftarrow (A \wedge M') \vee \big(B - (B \wedge M')\big)$$
$$Z \leftarrow C \vee (D \ll (2^i(b + 1))$$

Packed Bitonic Sort

[Albers-Hagerup (1997)]

$Z \leftarrow X \vee Reverse(Y)$

for $i \leftarrow \log k$ downto 1:

$$M \leftarrow CopyTestBits\Big(\big(C_2 \ll (b - i)\big) \wedge C_1\Big)$$
$$A \leftarrow (Z \wedge M) \gg ((b + 1) \ll i)$$
$$B \leftarrow Z - (Z \wedge M)$$
$$M' \leftarrow CopyTestBits\Big(\big((A \vee C_1) - B\big) \wedge C_1\Big)$$
$$C \leftarrow (B \wedge M') \vee \big(A - (A \wedge M')\big)$$
$$D \leftarrow (A \wedge M') \vee \big(B - (B \wedge M')\big)$$
$$Z \leftarrow C \vee (D \ll ((b + 1) \ll i)$$

Reversing the fields in a word

Similar to the implementation of bitonic sort.

For $i = 0,1,\ldots,\log k$, in any order,
swap fields with a 0 in the i-th bit of their index
with fields 2^i positions to the left.

We already know how to do it.

Exercise: Show that this indeed reverses the fields.

Packed Merge Sort

We began by splitting the n input numbers into groups of size $k = \Theta(w/b)$, naively sorting them, and then packing them into words.

This is good enough for obtaining an $O(n \log \log n)$-time algorithm, but the naïve sorting is clearly not optimal.

Exercise: Show that n integers, each of $\frac{w}{\log n \log \log n}$ bits, can be sorted in $O(n)$ time.

Integer Sorting in $O(n \log \log n)$ time
[Andersson-Hagerup-Nilsson-Raman (1998)]

Putting everything together, we get a *randomized* $O(n \log \log n)$-time sorting algorithm for any $w \geq \log n$.

How much space are we using?

If the *recursion stack* in managed carefully, the algorithm uses only $O(n)$ space.

Are we using *multiplications*?

Yes! In the hashing.

Non-AC^0 operations are required to get $O(1)$ search time

Sorting strings/multi-precision integers

We have n strings of arbitrary length.

Each character is a w-bit word.

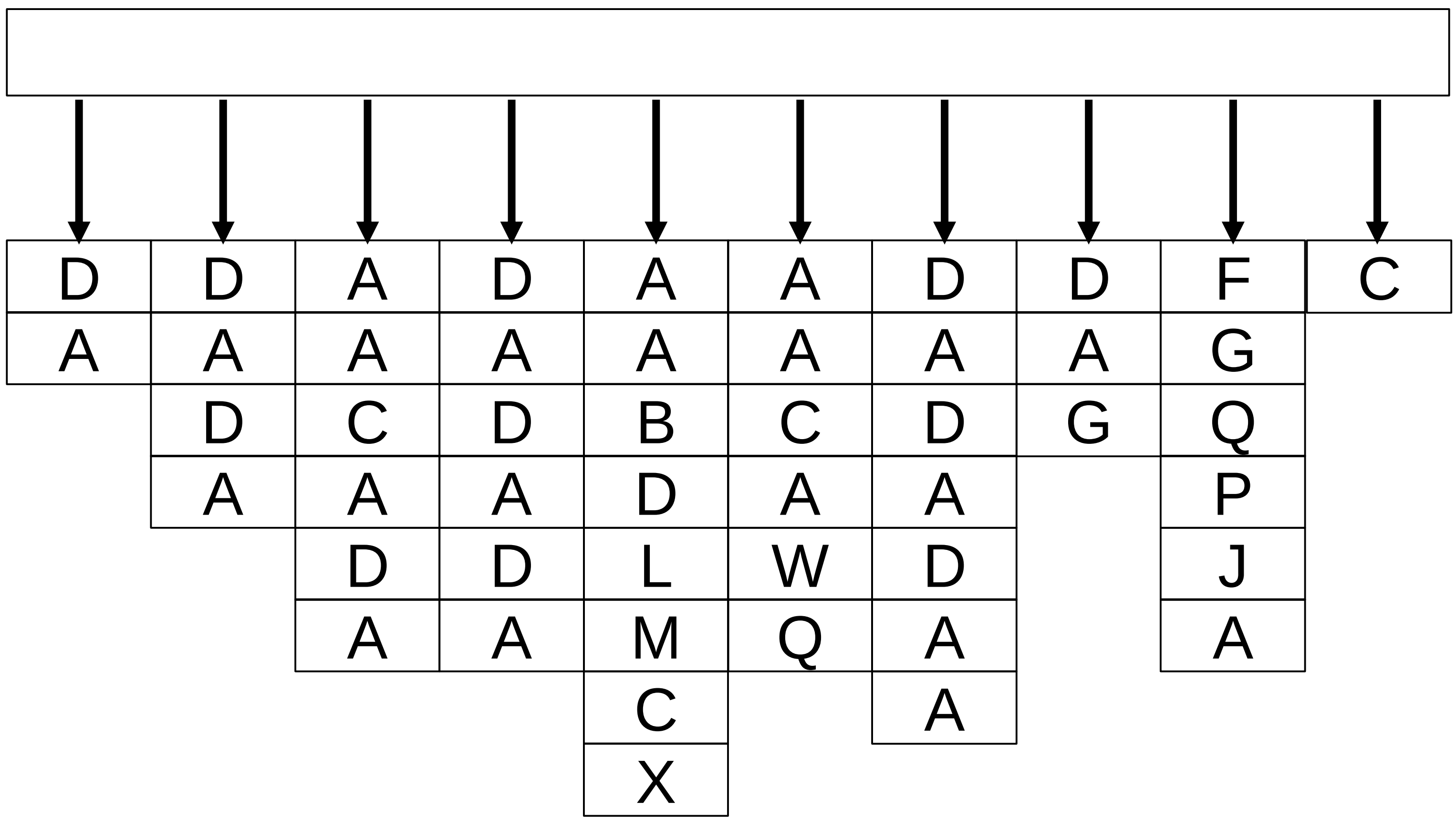
We want to sort them lexicographically.

Let N be the number of characters that *must* be examined to determine the order of the strings.

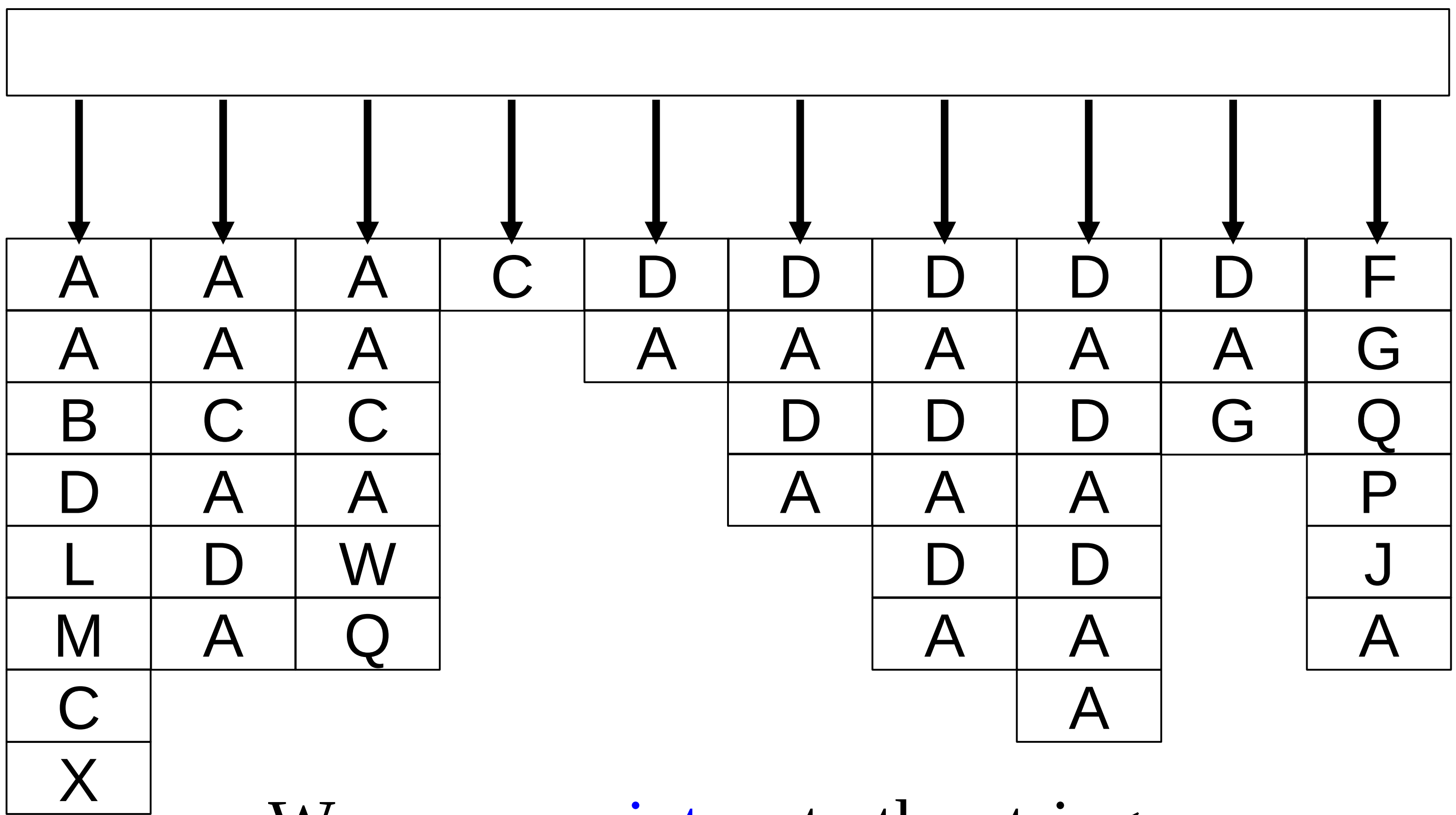
The problem can be reduced in $O(N+n)$ time to the problem of sorting n characters!

We get an $O(N + n \log\log n)$-time algorithm.

Sorting strings/multi-precision integers

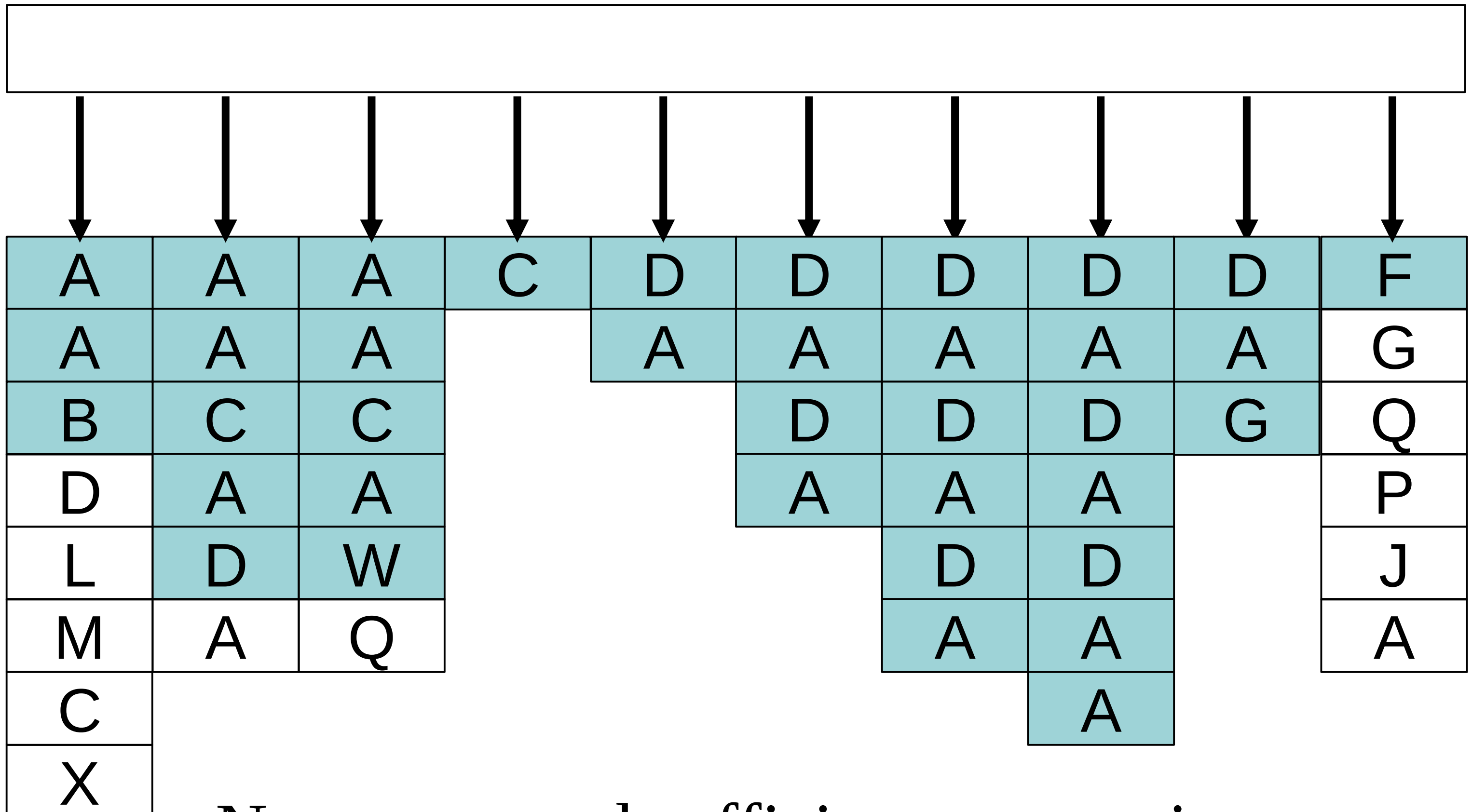


Sorting strings/multi-precision integers



We move pointers to the strings,
not the strings themselves.

Sorting strings/multi-precision integers



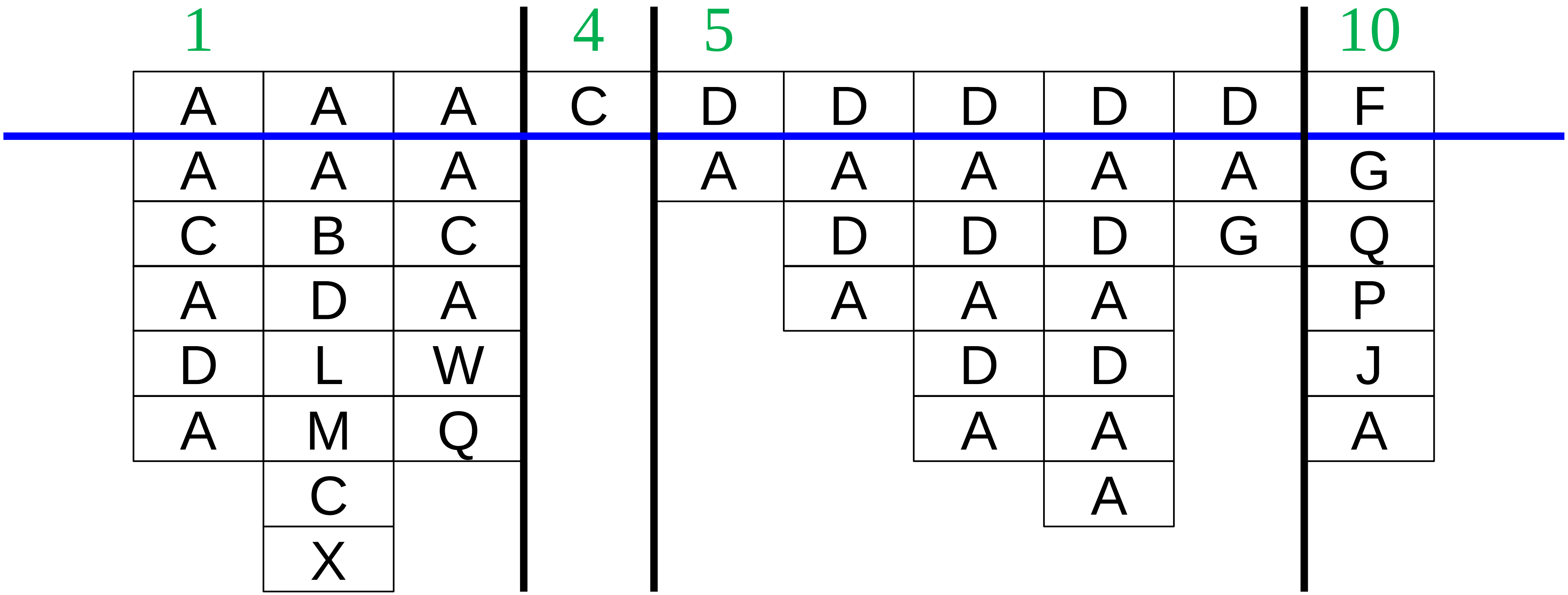
Necessary and sufficient to examine the *N* distinguishing characters.

Forward Radix Sort
[Andersson-Nilsson (1994)]

D	D	A	D	A	A	D	D	F	C
A	A	A	A	A	A	A	A	G	
	D	C	D	B	C	D	G	Q	
	A	A	A	D	A	A		P	
		D	D	L	W	D		J	
		A	A	M	Q	A		A	
				C		A			
				X					

After the i-th pass, the strings are sorted according to the first i characters.

Forward Radix Sort
[Andersson-Nilsson (1994)]

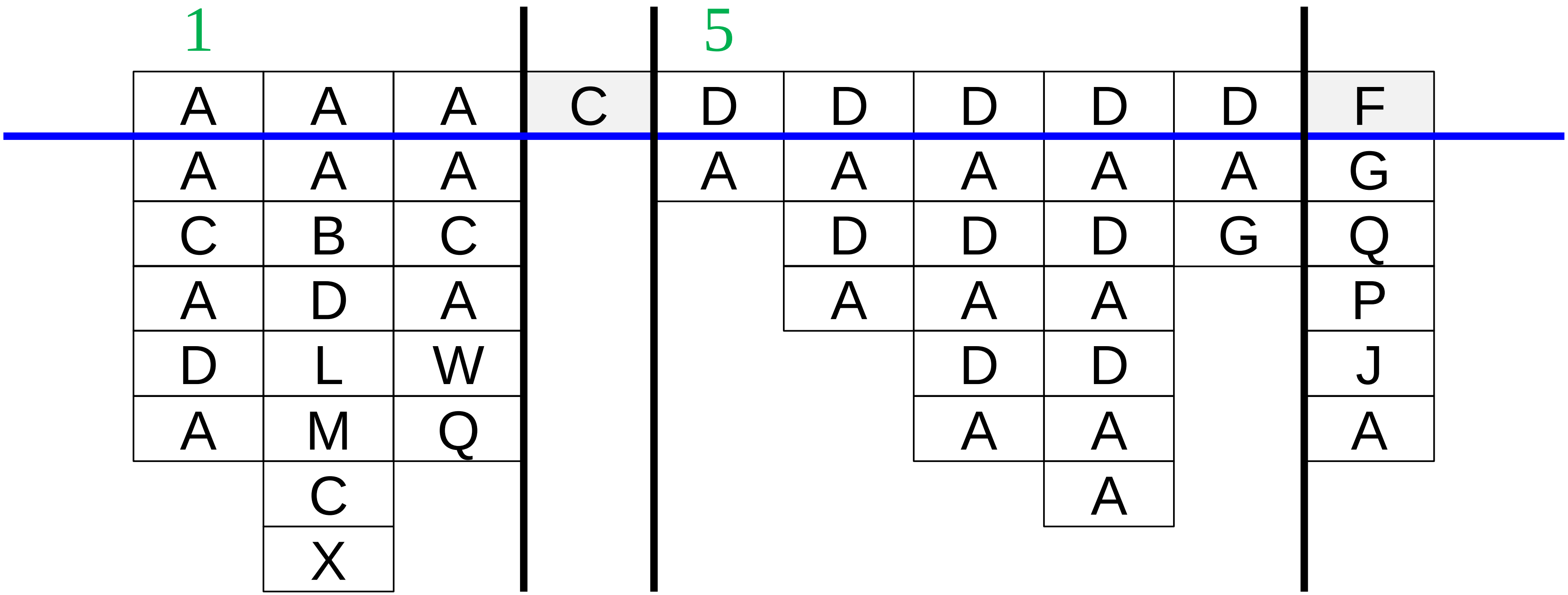


The strings are partitioned into groups.

We keep the starting/end positions of each group.

Groups are active or inactive.

Forward Radix Sort
[Andersson-Nilsson (1994)]

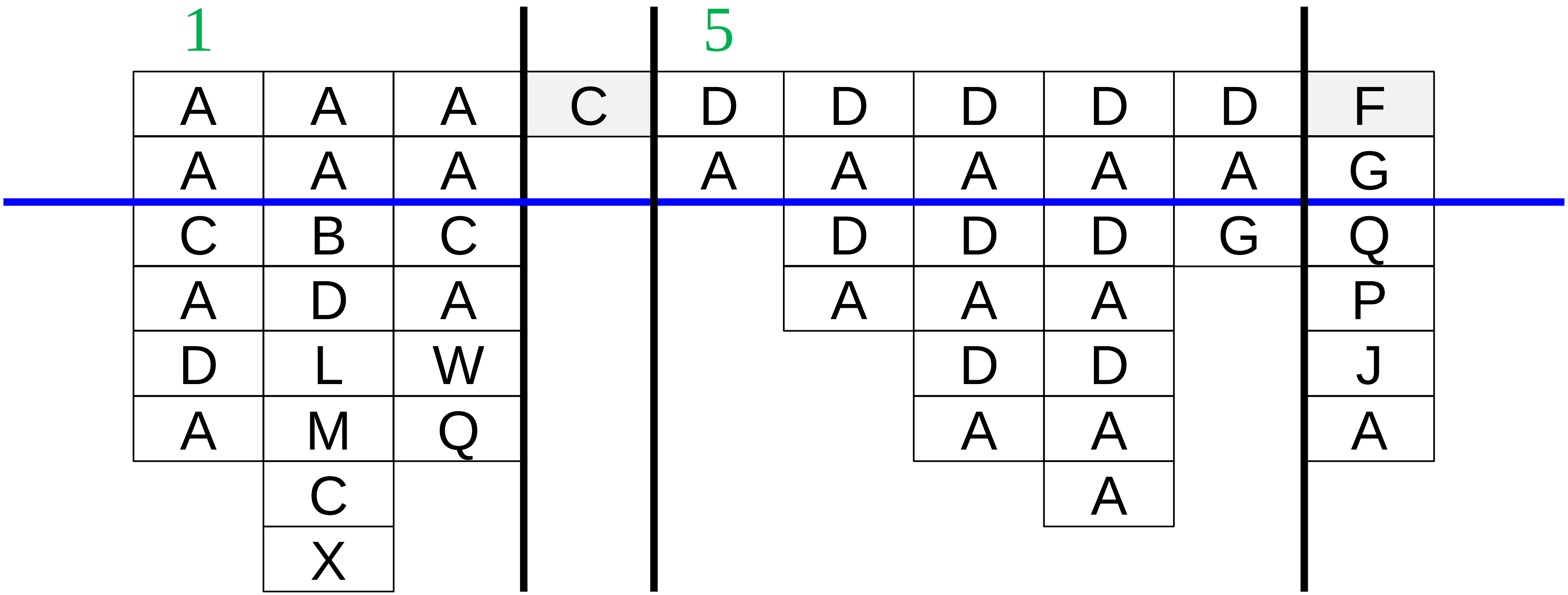


The strings are partitioned into groups.

We keep the starting position of each group.

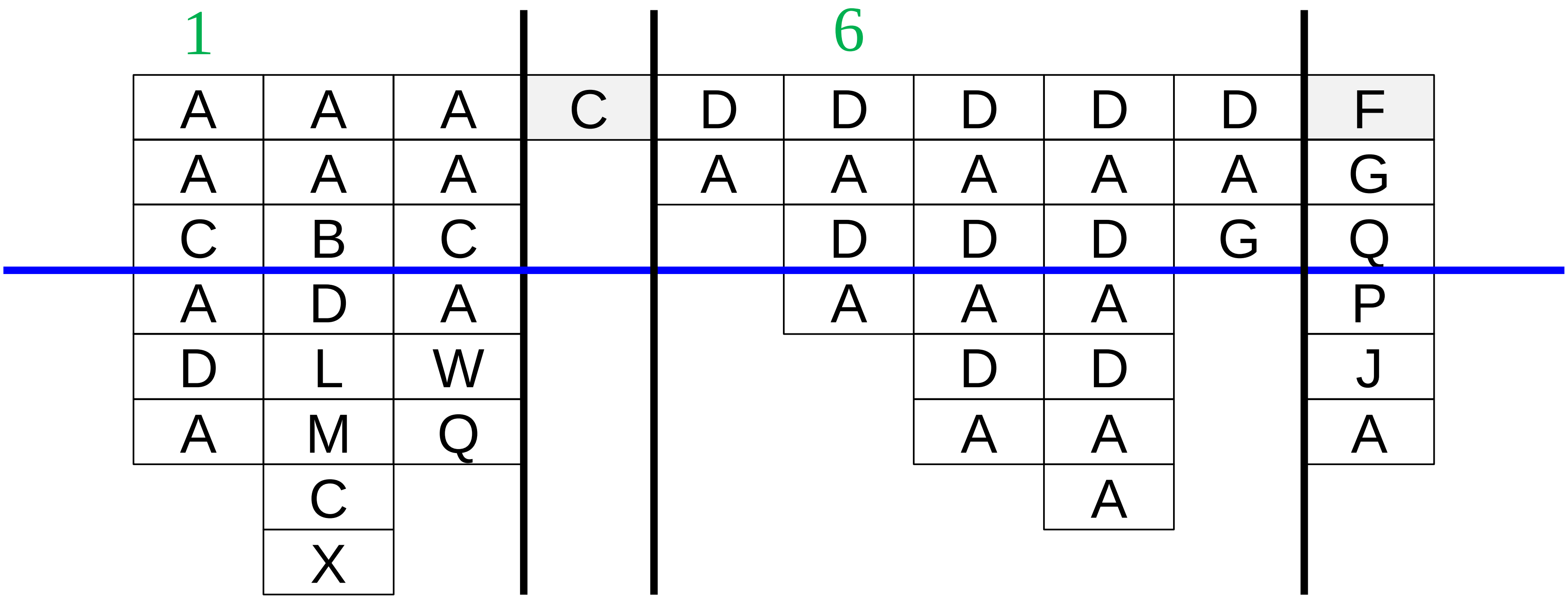
Groups are active or inactive.

Forward Radix Sort
[Andersson-Nilsson (1994)]



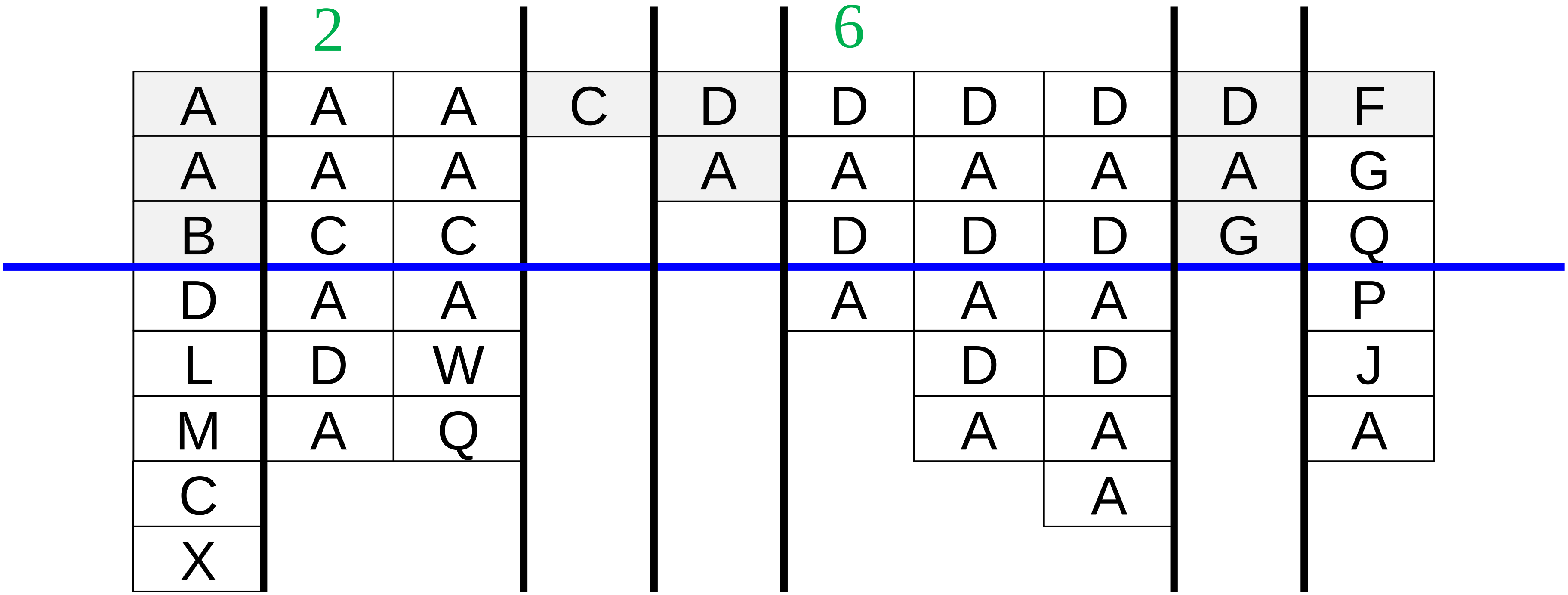
Forward Radix Sort

[Andersson-Nilsson (1994)]



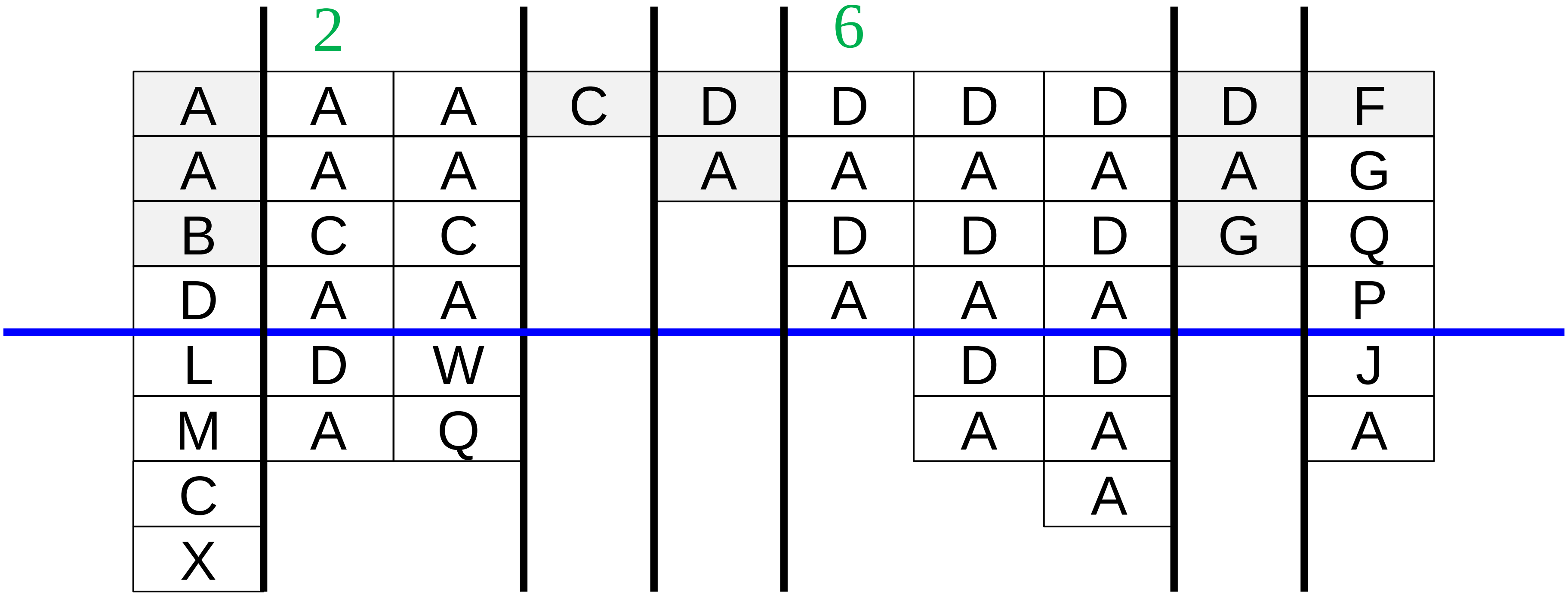
Forward Radix Sort

[Andersson-Nilsson (1994)]

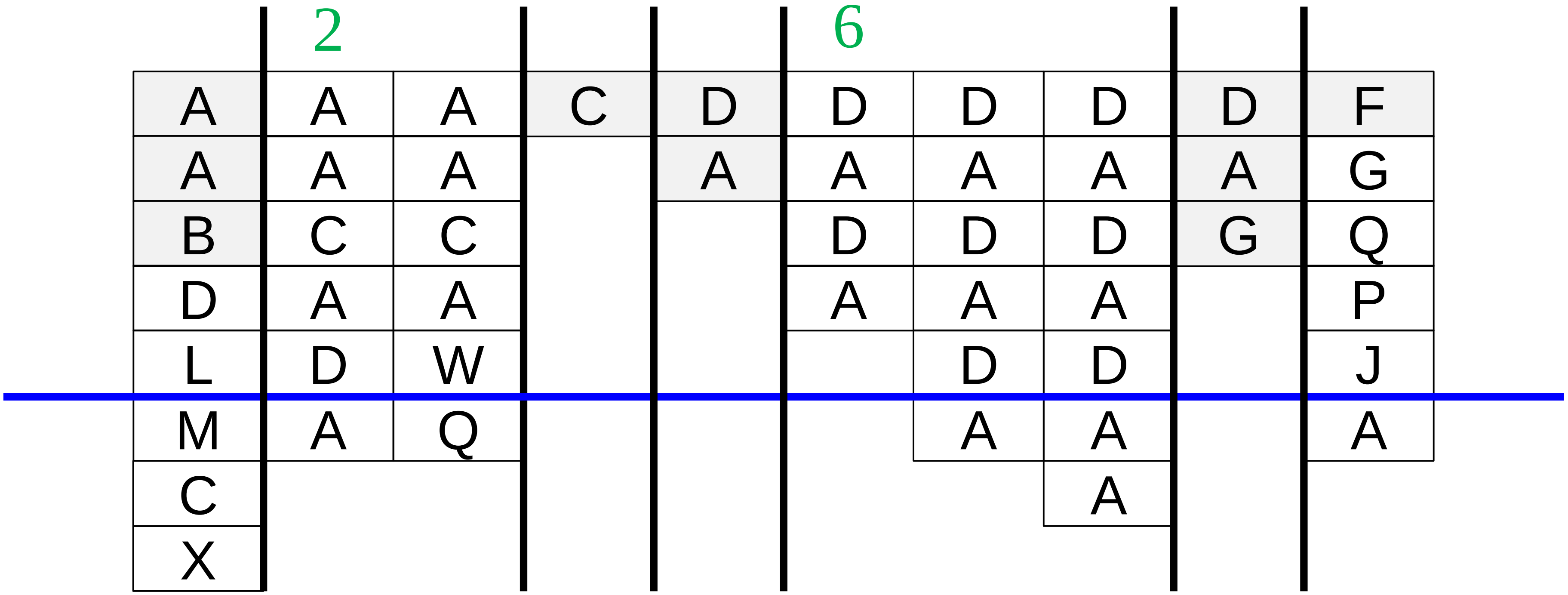


Forward Radix Sort

[Andersson-Nilsson (1994)]

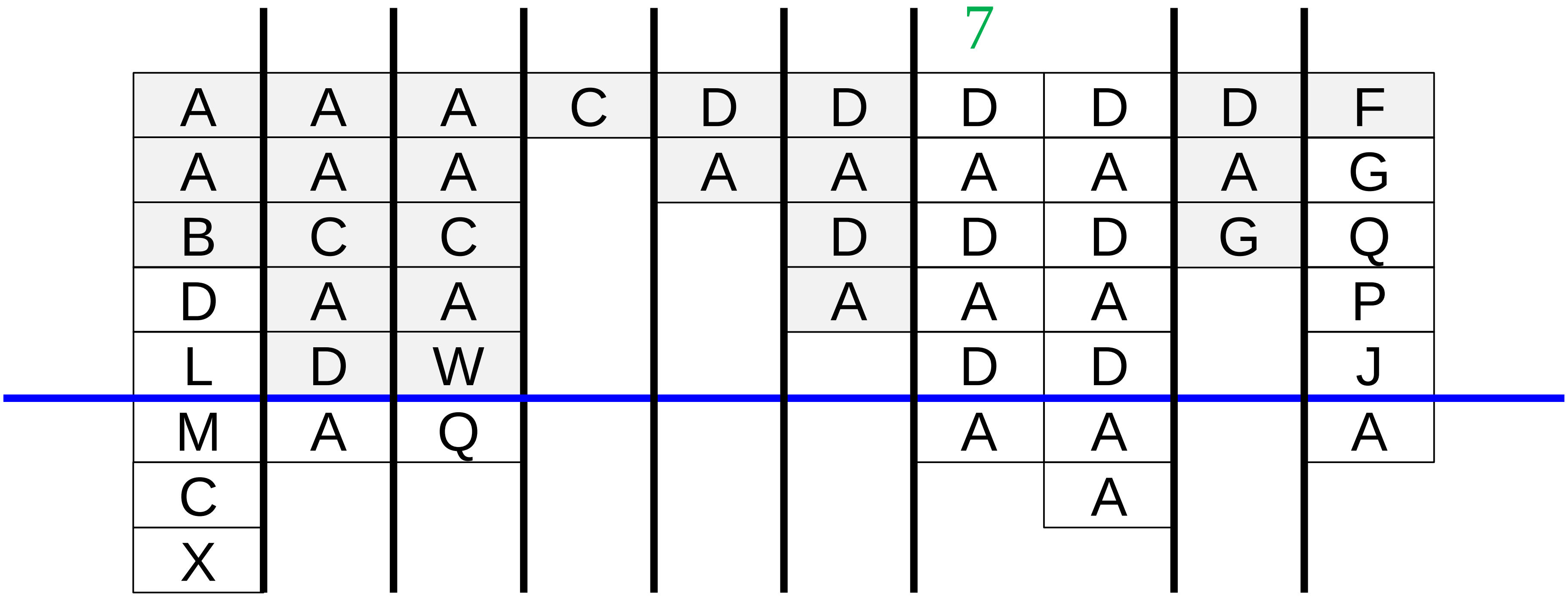


Forward Radix Sort
[Andersson-Nilsson (1994)]



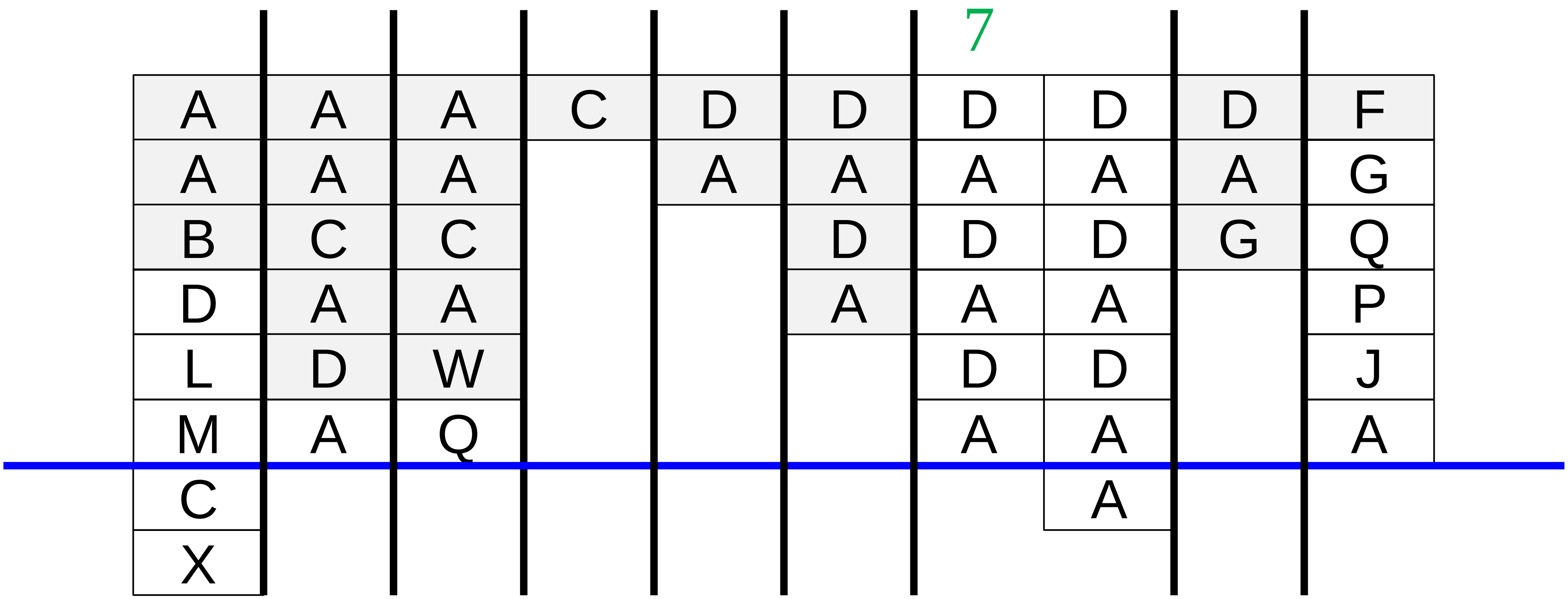
Forward Radix Sort

[Andersson-Nilsson (1994)]



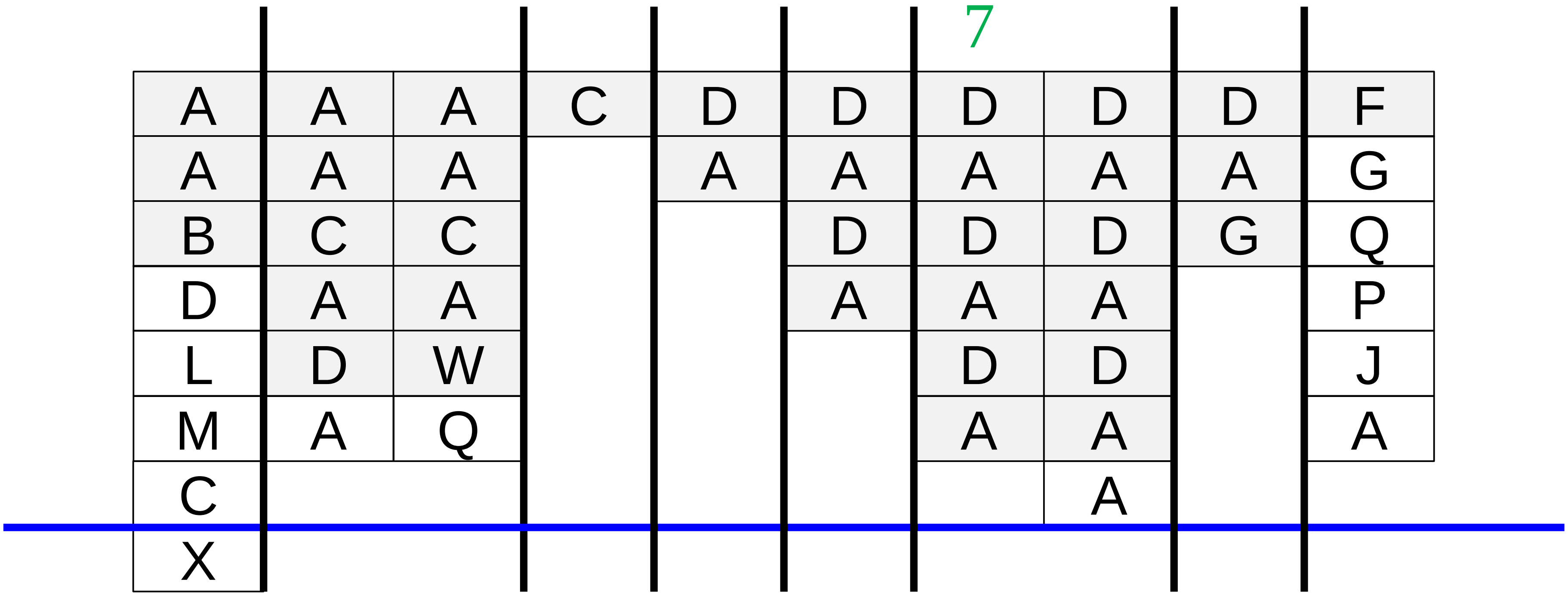
Forward Radix Sort

[Andersson-Nilsson (1994)]



Forward Radix Sort

[Andersson-Nilsson (1994)]



Forward Radix Sort

[Andersson-Nilsson (1994)]

A	A	A	C	D	D	D	D	D	F
A	A	A		A	A	A	A	A	G
B	C	C			D	D	D	G	Q
D	A	A			A	A	A		P
L	D	W				D	D		J
M	A	Q				A	A		A
C							A		
X									

Forward Radix Sort
[Andersson-Nilsson (1994)]

The i-th pass:

Sequentially scan the items in the active groups.

Append item x into bucket no. x_i.

(The buckets are shared by all groups.)
(Each item remembers the group it belongs to.)

"Empty" each active group.

Scan the non-empty buckets, in increasing order.

Append each item to its group.

How do we find the *non-empty* buckets?

Forward Radix Sort

(Slight deviation from [Andersson-Nilsson (1994)])

The i-th pass:

Consider each active group separately.

Use hashing to determine the different characters appearing in the i-th position.

If there are $k+1$ different characters, then the number of groups increases by k.

Sort the k non-minimal characters.

Total size of all sorting problems is at most n !

Forward Radix Sort

(Slight deviation from [Andersson-Nilsson (1994)])

Total size of all sorting problems,
in all passes, is at most n.

We promised one sorting of size at most n.

Having a collection of smaller problems
is in many cases better.

$$\sum k_i \le n \implies \sum k_i \log\log k_i \le n \log\log n$$

But, in some cases, e.g., if we want to use naïve
bucket sort, having one large problem is better.

Forward Radix Sort [Andersson-Nilsson (1994)]

Obtaining one sorting problem of size n.

Perform two phases.

In the first phase, split into sub-groups, but keep the sub-groups in arbitrary order.

Weaker invariant: After the i-th pass, all items in an active group have the same first i characters.

If c is a non-minimal character appearing in the i-th position in some group, add (i, c) to a list.

Forward Radix Sort
[Andersson-Nilsson (1994)]

<u>Obtaining one sorting problem of size n.</u>

If c is a non-minimal character appearing in the i-th position in some group, in the i-th pass, then add (i, c) to a list.

The total length of the list is at most n.

After sorting the list, in the second phase, we can run the original algorithm.

Slight problem: i cannot be bounded in terms of n.

Forward Radix Sort
[Andersson-Nilsson (1994)]

Obtaining one sorting problem of size n.

If c is a non-minimal character appearing in the i-th position in some group, in the i-th pass, then add (i, c) to a list.

Slight problem: i cannot be bounded in terms of n.

Simple solution: Replace (i, c) by (i', c), where i' is the number of passes, at or before the i-th pass, in which at least one group splits.

Now $i' \leq n$ and can be encoded using $\log n$ bits.

Range reduction revisited

We can view each w-bit word as a 2-character string, composed of $w/2$-bit characters

Using **forward radix sort** of Andersson and Nilsson, we get an alternative to the range reduction step of Kirkpatrick and Reisch.

Signature Sort

[Andersson-Hagerup-Nilsson-Raman (1998)]

Sorting in $O(n)$ expected time if $w \geq (\log n)^{2+\epsilon}$.

Split each b-bit key into q parts/characters, such that $O(q \log n)$-bit keys can be sorted in linear time.

We can choose $q = \Theta(w/((\log^2 n) \log \log n))$.

Use a hash function to assign each (b/q)-bit character a *unique* $O(\log n)$-bit *signature*.

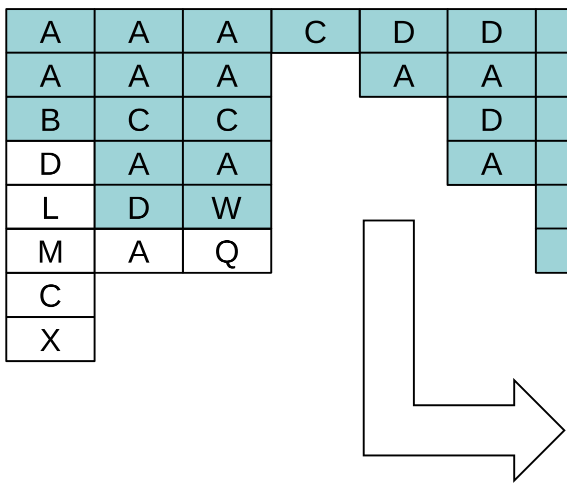
Form *shortened* keys by concatenating the *signatures* of the parts, and sort them in linear time.

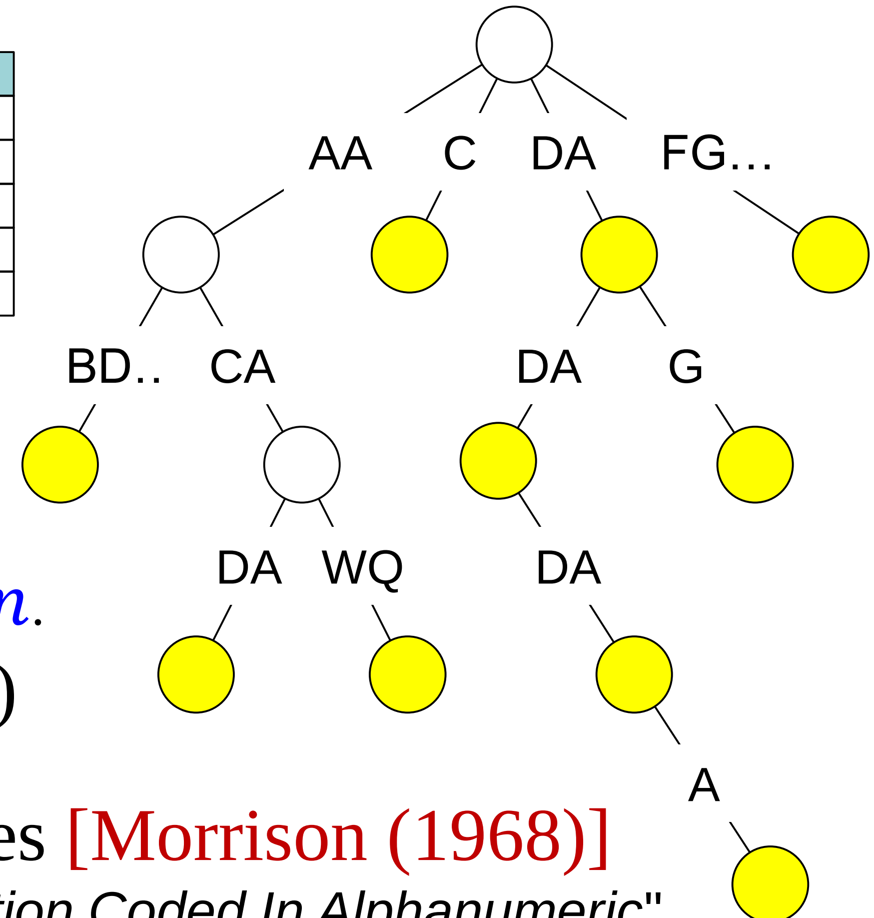
Construct a *compressed trie* of the *shortened* keys.

Sort the edges of the *trie*, possibly using recursion.

The keys now appear in the *trie* in sorted order.

Compressed tries





Number of nodes is at most $2n$.
(Only the root may be unary.)

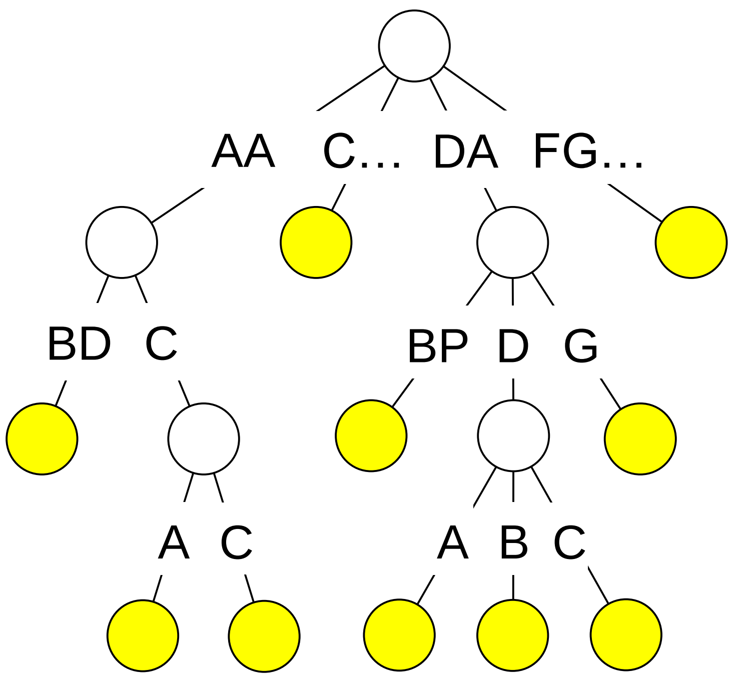
Also known as PATRICIA tries [Morrison (1968)]
"*Practical Algorithm To Retrieve Information Coded In Alphanumeric*".

Exercise: Show that a compessed trie of a sorted collection of strings can be constructed in $O(N)$ time.

Signature sort example

A	A	A	C	D	D	D	D	D	F
A	A	A	Z	A	A	A	A	A	G
B	C	C	O	B	D	D	D	G	Q
D	A	C	P	P	A	B	C	P	P

a	a	a	c	d	d	d	d	d	f
a	a	a	z	a	a	a	a	a	g
b	c	c	o	b	d	d	d	g	q
d	a	c	p	p	a	b	c	p	p

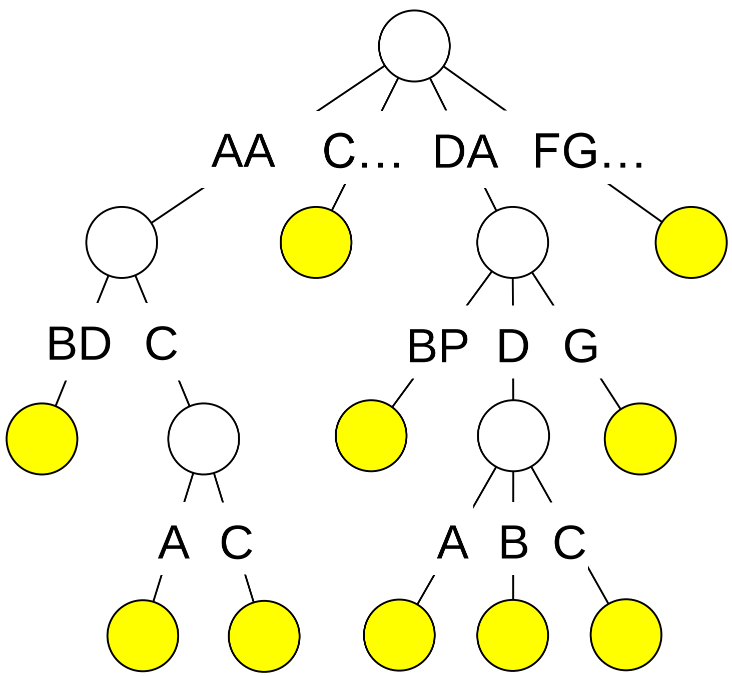


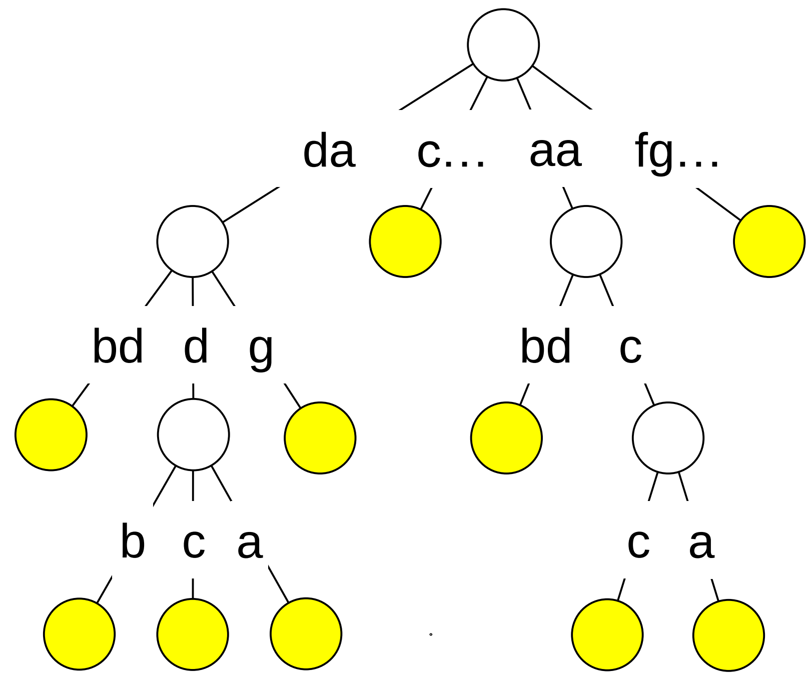
Signature sort example

A	A	A	C	D	D	D	D	D	F
A	A	A	Z	A	A	A	A	A	G
B	C	C	O	B	D	D	D	G	Q
D	A	C	P	P	A	B	C	P	P

d	d	d	d	d	c	a	a	a	f
a	a	a	a	a	z	a	a	a	g
b	d	d	d	g	o	b	c	c	q
p	b	c	a	p	p	d	c	a	p

$b < d < c < a < f < g$





Signature Sort
[Andersson-Hagerup-Nilsson-Raman (1998)]

Sorting in $O(n)$ expected time if $w \geq (\log n)^{2+\epsilon}$.

Q: How do we find *unique signatures*?

Q: How do we sort the *shortened* keys?

A: Using packed sorting in $O(n)$ time.

Q: How do we construct the *trie* of the *shortened* keys? Note that $O(N) = O(nq)$ is not fast enough!

Q: How do we reorder the *trie* of the *shortened* keys to obtain the *trie* of the original keys?

A: Sort the original first character on each edge. If characters are not short enough use recursion.

Signature Sort
[Andersson-Hagerup-Nilsson-Raman (1998)]

$$T(n, w, b) = O(n) + T(n, w, q \log n) + T(n, w, b/q)$$

The trie has up to $2n$ edges.
Why is this n and not $2n$?

As we are only going to repeat it a *constant* number of times, it does not really matter.

But we can get down to n.

Use the trick of finding the minimum edge separately and not including it in the sort.

Exercise: Number of chars to sort is exactly $n - 1$.

Signature Sort
[Andersson-Hagerup-Nilsson-Raman (1998)]

$$T(n, w, b) = O(n) + T(n, w, q \log n) + T(n, w, b/q)$$

As $q = \dfrac{w}{\log^2 n \log \log n}$ we have:

$$T(n, w, q \log n) = O(n)$$

$$T(n, w, w/q) = T(n, w, \log^2 n \log \log n)$$

(This is $O(n)$ if $w \geq \log^3 n \ (\log \log n)^2$.)

If we iterate i times we get:

$$T(n, w, w) = O(in) + T\left(n, w, w/q^i\right)$$

Signature Sort

[Andersson-Hagerup-Nilsson-Raman (1998)]

If we iterate i times we get:

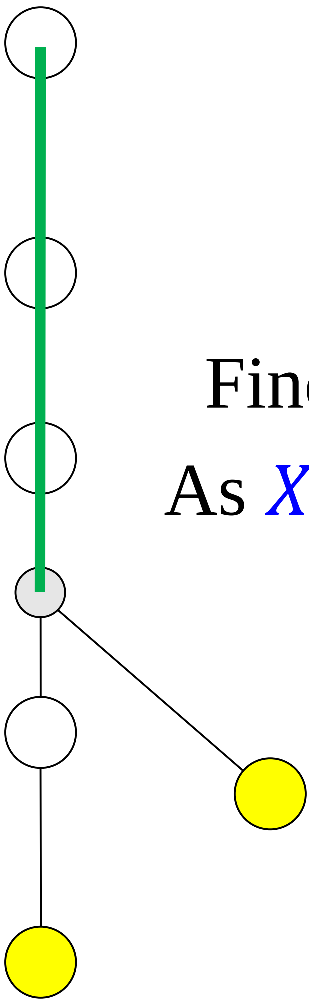
$$T(n,w,w) = O(in) + T\left(n,w,w/q^i\right)$$

If $w \geq (\log n)^{2+\epsilon}$, then $q = \dfrac{w}{\log^2 n \log \log n} \geq (\log n)^{\epsilon'}$.

If i is a large enough constant, e.g., $i \geq 1/\epsilon'$,
then $q^i \geq \log n \log \log n$.

Thus, for $w \geq (\log n)^{2+\epsilon}$, we can sort in $O(n)$ time.

Constructing a compressed trie in $O(n)$ time



Add the strings to the trie one by one.

Suppose that we are about to insert X_k.

The left-most path corresponds to X_{k-1}.

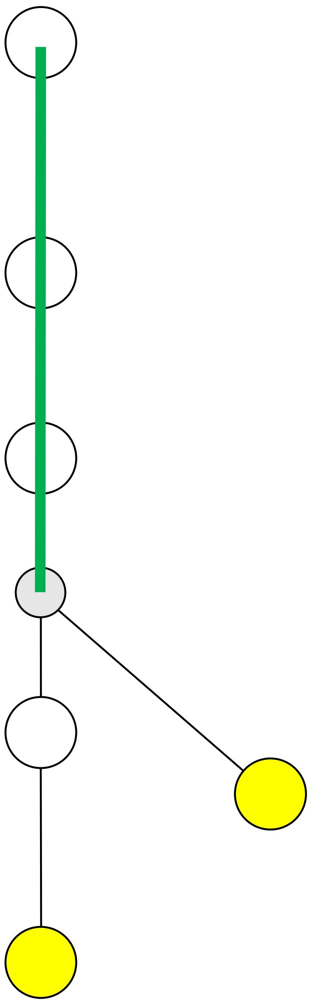
Find the *longest common prefix* of X_{k-1} and X_k.

As X_{k-1}, X_k are packed, we can do it in $O(1)$ time.

We may need to add an internal node,
unless the common prefix ends at node.

How do we find the parent
of the new internal node?

Constructing a compressed trie in $O(n)$ time



How do we find the parent
of the new internal node?

We can (probably) use bit tricks
to do it in $O(1)$ time.

We can also slowly climb up from last leaf.

Each node we pass *exits* the left-most path.

Total number of operations is $O(n)$.

Note: Similar to the linear time
construction of Cartesian trees.

Computing unique signatures
[Andersson-Hagerup-Nilsson-Raman (1998)]

We have at most $nq \le n^2$ different characters.

Let H be an (almost) universal family of hash functions from $[0, 2^{b/q} - 1]$ to $[0, 2^{\ell} - 1]$.

The expected number of *collisions* is at most $n^4/2^{\ell}$.

For $\ell = 5 \log n$, there are no collisions, w.h.p.

Which family of hash functions should we use?

How do we compute the signatures of all q characters of a given word in $O(1)$ time?

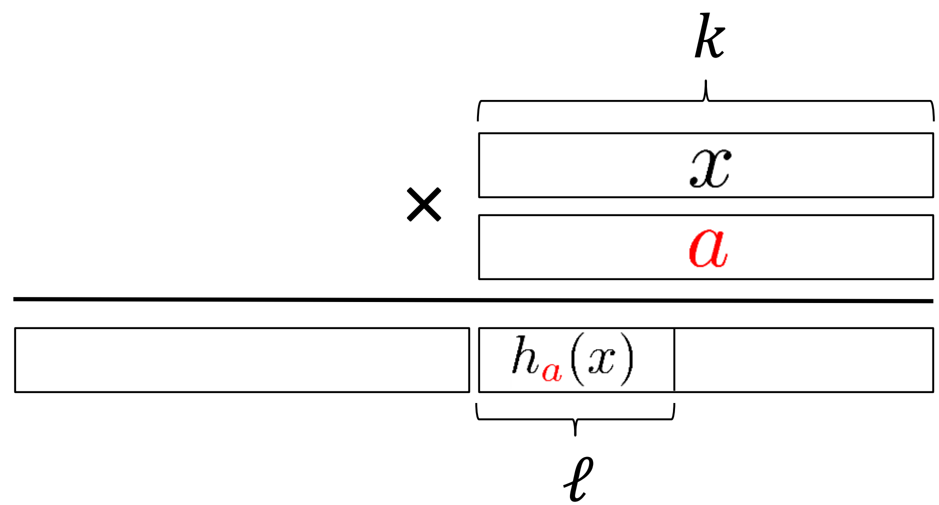
Multiplicative hash functions

[Dietzfelbinger-Hagerup-Katajainen-Penttonen (1997)]

$$h_a : [2^k] \to [2^\ell]$$

$$h_a(x) = \left\lfloor \frac{ax \bmod 2^k}{2^{k-\ell}} \right\rfloor$$

$1 \le a < 2^k$ odd



$$h_a(x) = ((a * x) \overbrace{\wedge \big((1 \ll k) - 1\big)}^{\text{Not necessary if } k = w}) \gg (k - \ell)$$

Form an "almost-universal" family

Extremely fast in practice!