Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

Last updated on 11/16/21 10:21 AM

https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

String processing

String. Sequence of characters.

Important fundamental abstraction.

- Programming systems (e.g., Java programs).
- Communication systems (e.g., email).
- Information processing.
- Genomic sequences.
- ...

> *"The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology."* — *M. V. Olson*

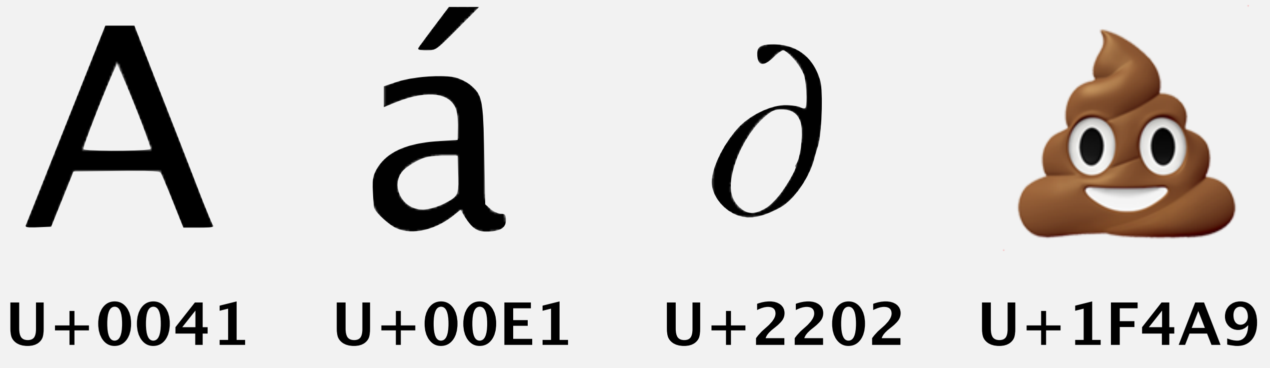
The char data type

C char data type. Typically an 8-bit integer (between 0 and 255).

- Supports 7-bit ASCII.
- Represents only $2^8 = 256$ characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

all 2^7 = 128 ASCII characters



some Unicode characters

can use as an index into an array

Java char data type. A 16-bit unsigned integer (between 0 and 65,535).

- Supports 16-bit Unicode 1.0.1.
- Supports 21-bit Unicode 10.0.0 (awkwardly via UTF-8).

I 💖 Unicode

U+1F496

💖

The String data type (in Java 11)

String data type. Immutable sequence of characters.

Java 11 representation. A fixed-length `char[]` array.

operation	description	Java	running time
length	*number of characters*	`s.length()`	1
indexing	*character at index i*	`s.charAt(i)`	1
concatenation	*concatenate one string to the end of the other*	`s + t`	$m + n$
⋮	⋮		

String performance trap

Q. How to build a long string, one character at a time?

```
public static String reverse(String s)
{
   String reverse = "";
   for (int i = s.length() - 1; i >= 0; i--)
      reverse += s.charAt(i);
   return reverse;
}
```

quadratic time

$(1 + 2 + 3 + \ldots + n)$

StringBuilder data type. Mutable sequence of characters.

Java representation. A resizing `char[]` array.

alternatively,

`new StringBuilder(s).reverse().toString()`

```
public static String reverse(String s)
{
   StringBuilder reverse = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      reverse.append(s.charAt(i));
   return reverse.toString();
}
```

linear time

$n + (1 + 2 + 4 + 8 + 16 + \ldots + n)$

THE STRING DATA TYPE: IMMUTABILITY

Q. Why are Java strings immutable?

Alphabets

Digital key. Sequence of digits over a given alphabet.

Radix. Number of digits R in alphabet.

name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

techniques also extend to
64-bit integers and other digital keys

Note. We use extended ASCII strings in this lecture (but analyze in terms of R).

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

Review: summary of the performance of sorting algorithms

Frequency of calls to `compareTo()`.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ n^2	¼ n^2	$\Theta(1)$	✔	`compareTo()`
mergesort	$n \log_2 n$	$n \log_2 n$	$\Theta(n)$	✔	`compareTo()`
quicksort	$1.39\ n \log_2 n$ *	$1.39\ n \log_2 n$ *	$\Theta(\log n)$ *		`compareTo()`
heapsort	$2\ n \log_2 n$	$2\ n \log_2 n$	$\Theta(1)$		`compareTo()`

* probabilistic

Sorting lower bound. In the worst case, any compare-based sorting algorithm makes $\Omega(n \log n)$ compares. ← `compareTo()` not constant time for string keys

Q. Can we sort strings faster (despite lower bound)?

A. Yes, by exploiting access to individual characters. ← use characters to make R-way decisions (instead of binary decisions)

Key-indexed counting: assumptions about keys

Assumption. Each key is an integer between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort class roster by section number.
- Sort phone numbers by area code.
- Sort playing cards by suit.
- Sort string by first letter.
- Use as a subroutine in string sorting algorithm.

Remark. Keys typically have associated data $\Rightarrow$ can't simply count keys of each value.

input		**sorted result**	
name	*section*	*(by section)*	
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

↑ *keys are small integers*

Key-indexed counting demo

Goal. Sort an array `a[]` of n characters between 0 and $R - 1$.

$R = 6$

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Distribute items to auxiliary array using cumulative frequencies.
- Copy back into original array.

```
int n = a.length;
int[] count = new int[R+1];

for (int i = 0; i < n; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < n; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < n; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

Key-indexed counting demo

Goal. Sort an array `a[]` of n characters between 0 and $R - 1$.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- Copy back into original array.

count frequencies →

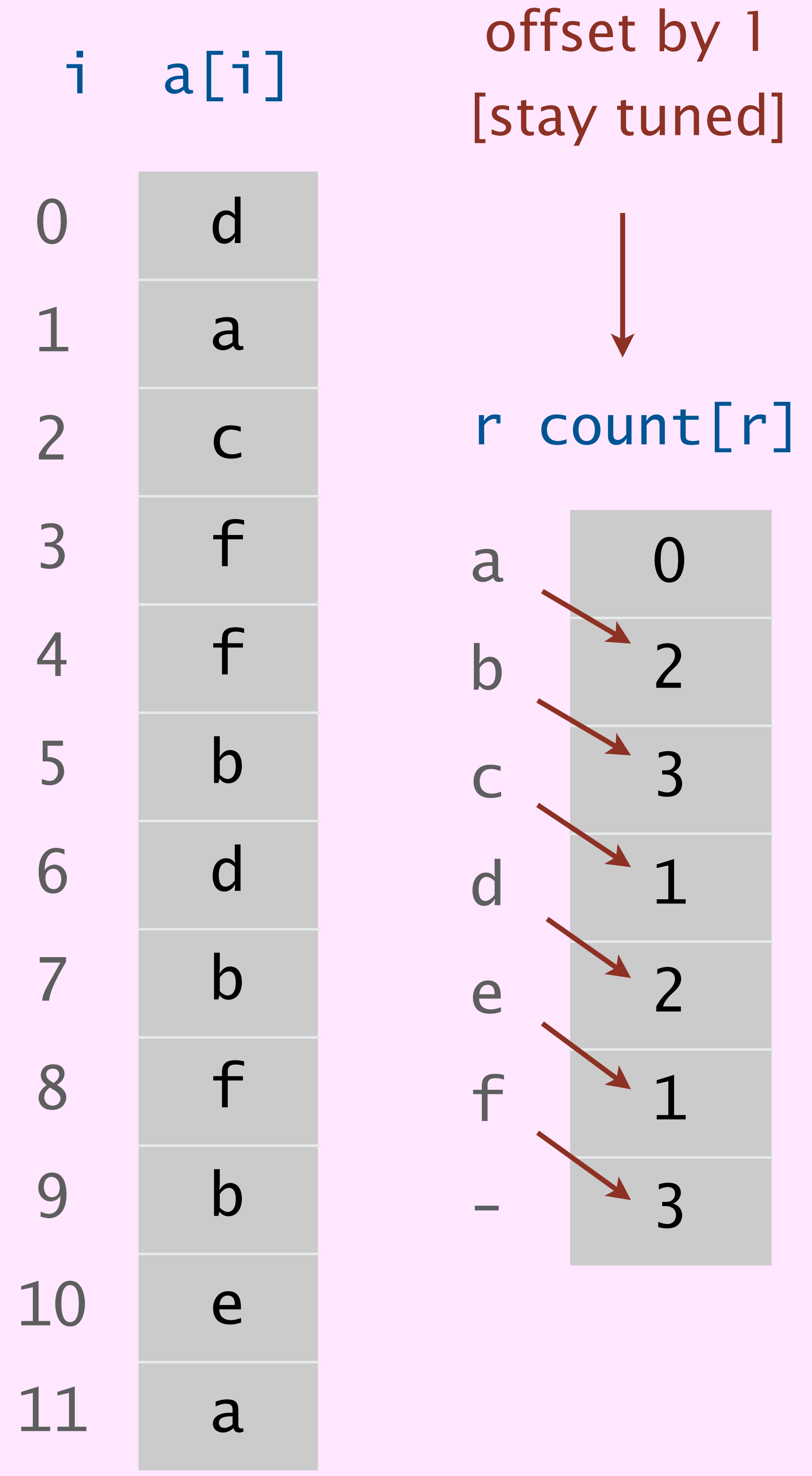
```
int n = a.length;
int[] count = new int[R+1];

for (int i = 0; i < n; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < n; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < n; i++)
   a[i] = aux[i];
```



i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array `a[]` of n characters between 0 and $R - 1$.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- Copy back into original array.

```
int n = a.length;
int[] count = new int[R+1];

for (int i = 0; i < n; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < n; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < n; i++)
   a[i] = aux[i];
```

compute cumulates →

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array `a[]` of n characters between 0 and $R - 1$.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- Copy back into original array.

move items →

```
int n = a.length;
int[] count = new int[R+1];

for (int i = 0; i < n; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < n; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < n; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. Sort an array `a[]` of n characters between 0 and $R - 1$.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- Copy back into original array.

```
int n = a.length;
int[] count = new int[R+1];

for (int i = 0; i < n; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < n; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < n; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Radix sorting: quiz 1

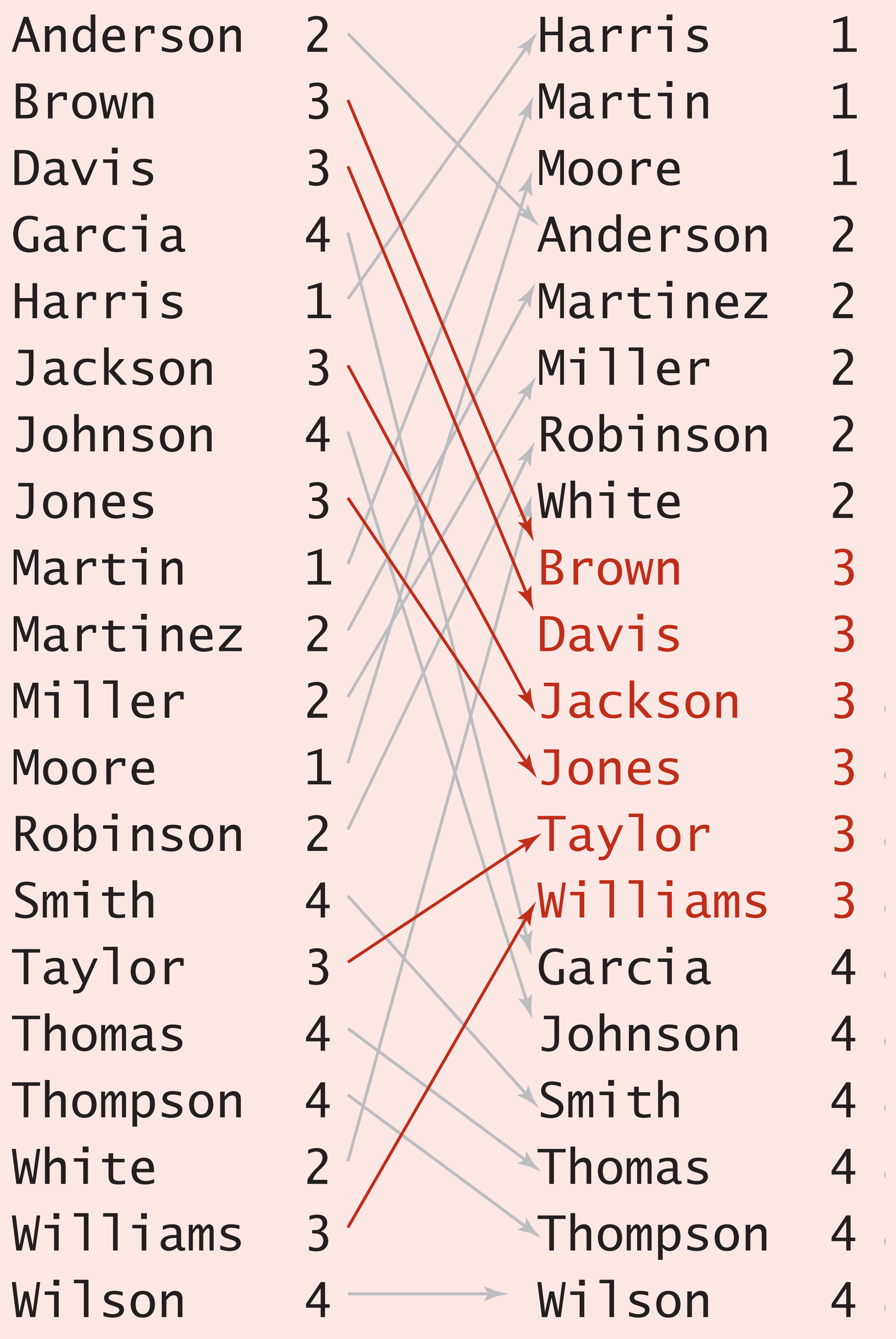
Which of the following are properties of key-indexed counting?

A. $\Theta(n + R)$ time.

B. $\Theta(n + R)$ extra space.

C. Stable.

D. All of the above.



Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

stability


Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

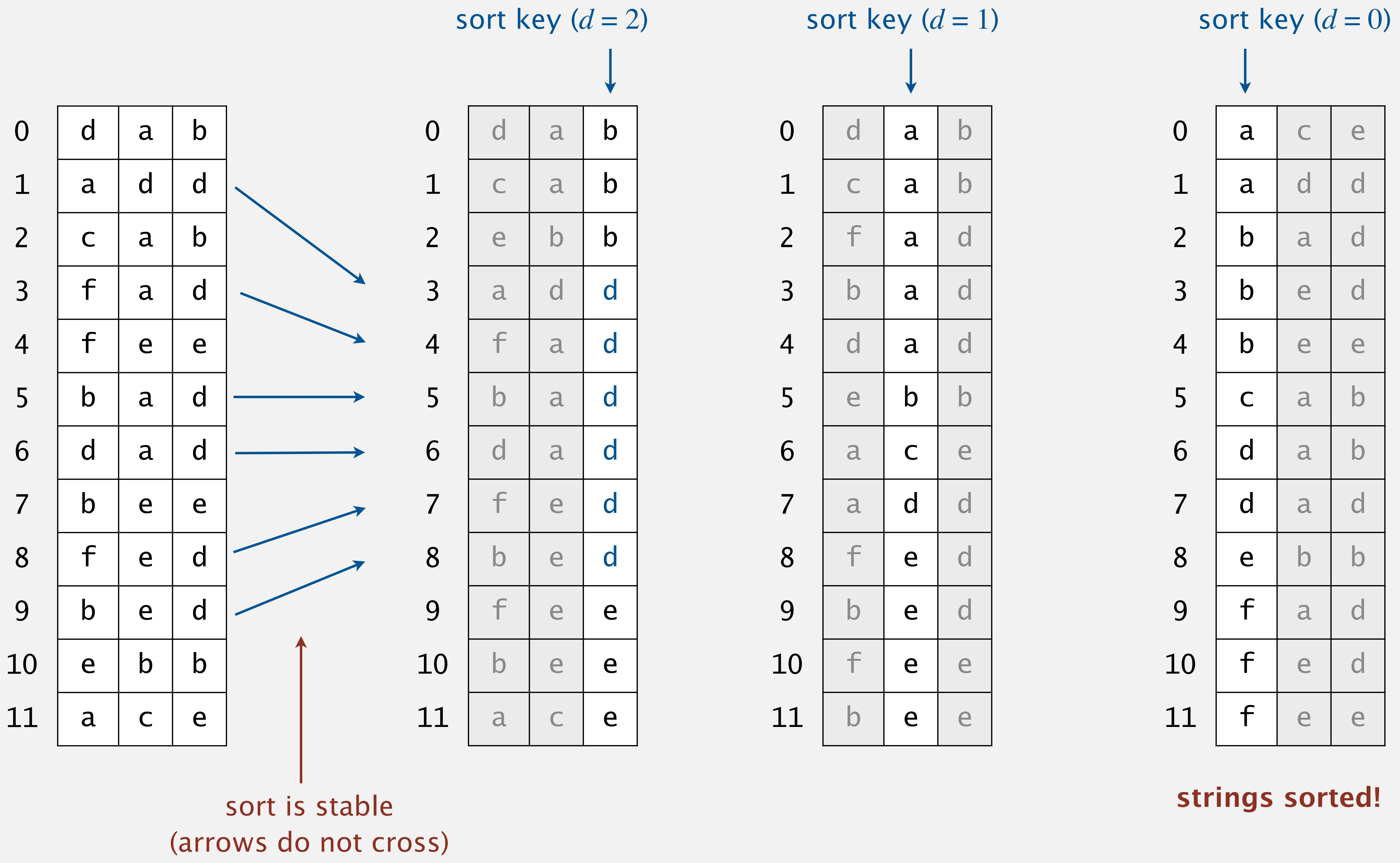
https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

Least-significant-digit-first (LSD) radix sort

- Consider characters from right to left.
- Stably sort using character d as the key (using key-indexed counting).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

sort key ($d = 2$)

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

sort key ($d = 1$)

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

sort key ($d = 0$)

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

sort is stable
(arrows do not cross)

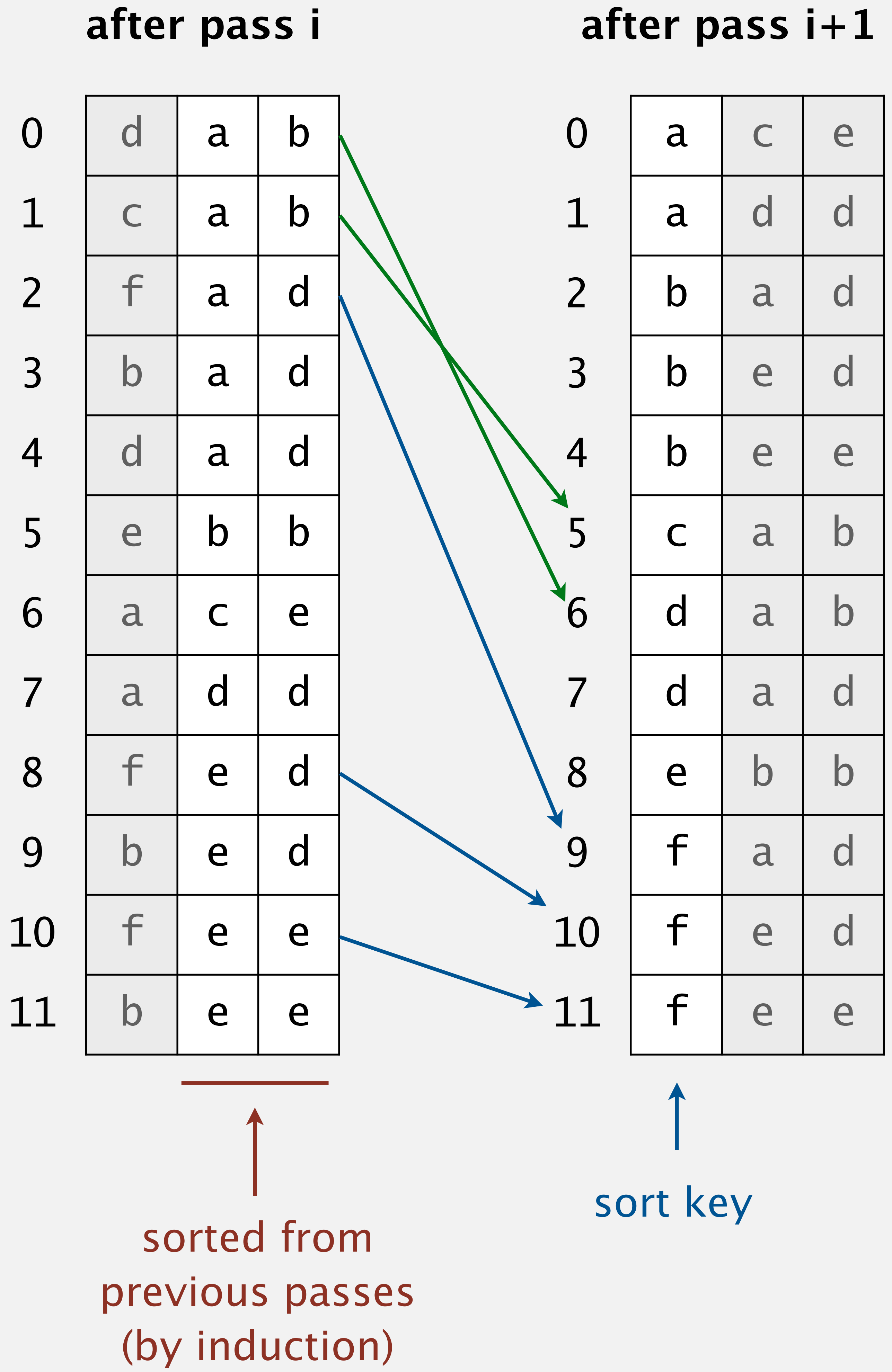
strings sorted!

LSD string sort: correctness proof

Proposition. LSD sorts any array of n strings, each of length w, in $\Theta(w(n+R))$ time.

Pf of correctness. [by induction on # passes i]

- Inductive hypothesis: after pass i, strings are sorted by last i characters.
- After pass $i+1$, string are sorted by last $i+1$ last characters because...
  - if two strings differ on sort key, key-indexed counting puts them in proper relative order
  - if two strings agree on sort key, stability of key-indexed counting keeps them in proper relative order



	after pass i		
0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

	after pass i+1		
0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

Proposition. LSD sort is stable.

Pf. Key-indexed counting is stable.

LSD string sort (for fixed-length strings): Java implementation

```java
public class LSD
{
   public static void sort(String[] a, int w)
   {
      int R = 256;
      int n = a.length;
      String[] aux = new String[n];

      for (int d = w-1; d >= 0; d--)
      {
         int[] count = new int[R+1];
         for (int i = 0; i < n; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < n; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < n; i++)
            a[i] = aux[i];

      }
   }
}
```

radix *R*

fixed-length *w* strings

do key-indexed counting
for each digit from right to left

key-indexed counting
(using character *d*)

Summary of the performance of sorting algorithms

Frequency of calls to `compareTo()` and `charAt()`.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ n^2	¼ n^2	$\Theta(1)$	✔	`compareTo()`
mergesort	$n \log_2 n$	$n \log_2 n$	$\Theta(n)$	✔	`compareTo()`
quicksort	$1.39\, n \log_2 n$ *	$1.39\, n \log_2 n$ *	$\Theta(\log n)$ *		`compareTo()`
heapsort	$2\, n \log_2 n$	$2\, n \log_2 n$	$\Theta(1)$		`compareTo()`
LSD sort †	$2\, w\, n$	$2\, w\, n$	$\Theta(n + R)$	✔	`charAt()`

* probabilistic

† fixed-length w keys

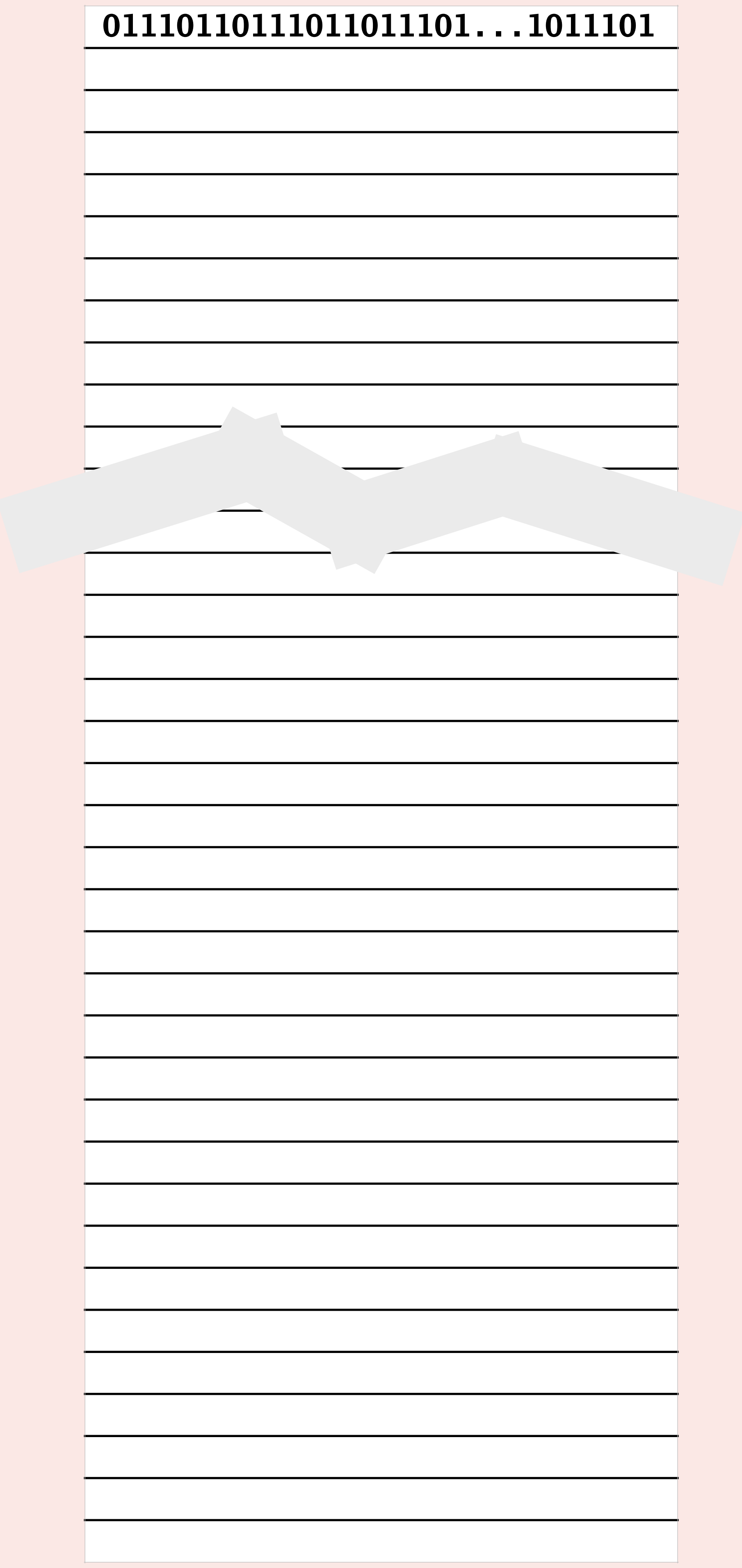
a call to `compareTo()` can involve as many as $2w$ calls to `charAt()`

Google CEO Eric Schmidt interviews Barack Obama in November 2007

Radix sorting: quiz 2

Which algorithm below is fastest for sorting 1 million 32-bit integers?

- **A.** Insertion sort.
- **B.** Mergesort.
- **C.** Quicksort.
- **D.** LSD sort.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

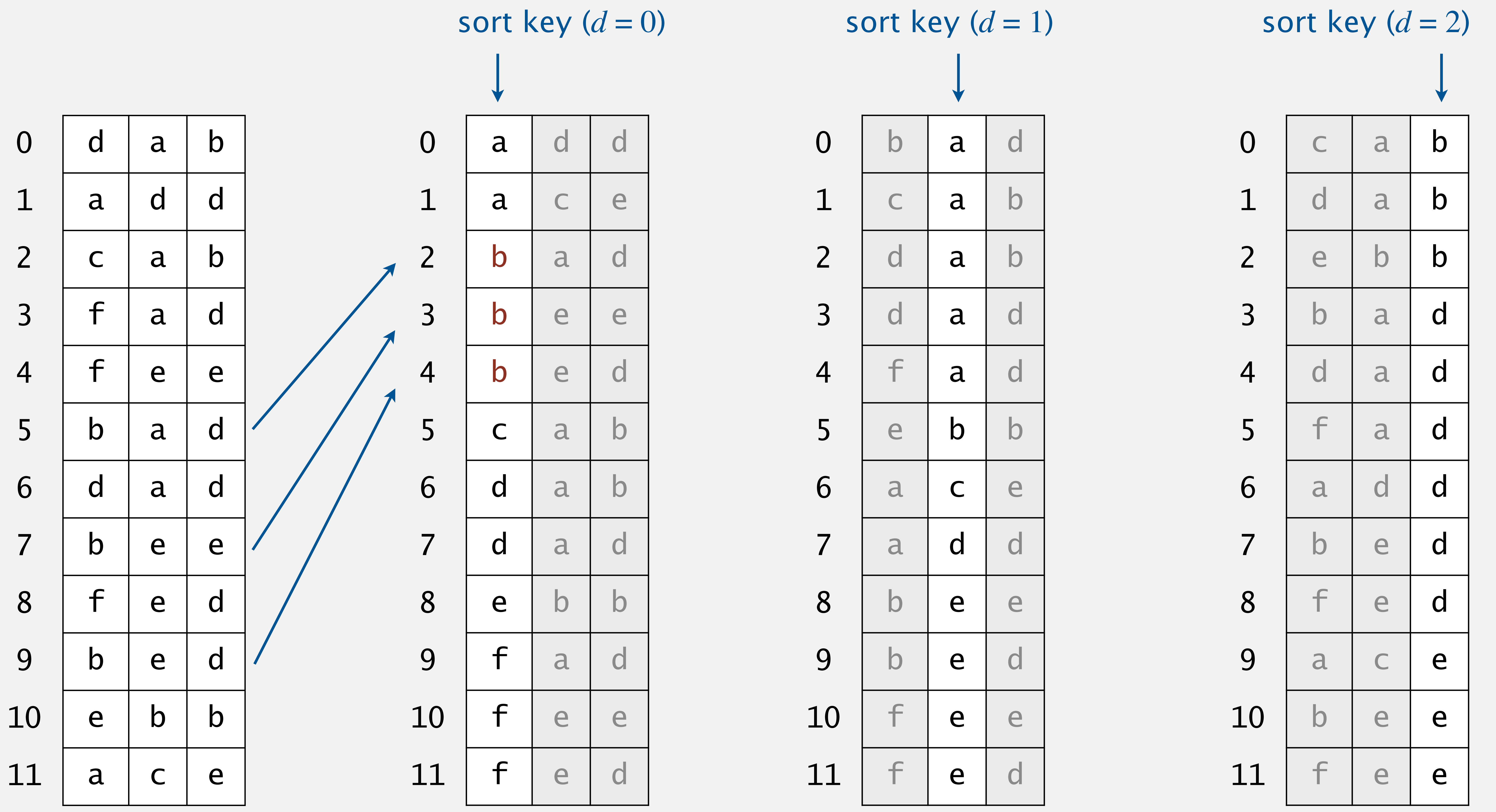
https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

Reverse LSD

- Consider characters from left to right.
- Stably sort using character d as the key (using key-indexed counting).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

sort key ($d = 0$)

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key ($d = 1$)

0	b	a	d
1	c	a	b
2	d	a	b
3	d	a	d
4	f	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	b	e	e
9	b	e	d
10	f	e	e
11	f	e	d

sort key ($d = 2$)

0	c	a	b
1	d	a	b
2	e	b	b
3	b	a	d
4	d	a	d
5	f	a	d
6	a	d	d
7	b	e	d
8	f	e	d
9	a	c	e
10	b	e	e
11	f	e	e

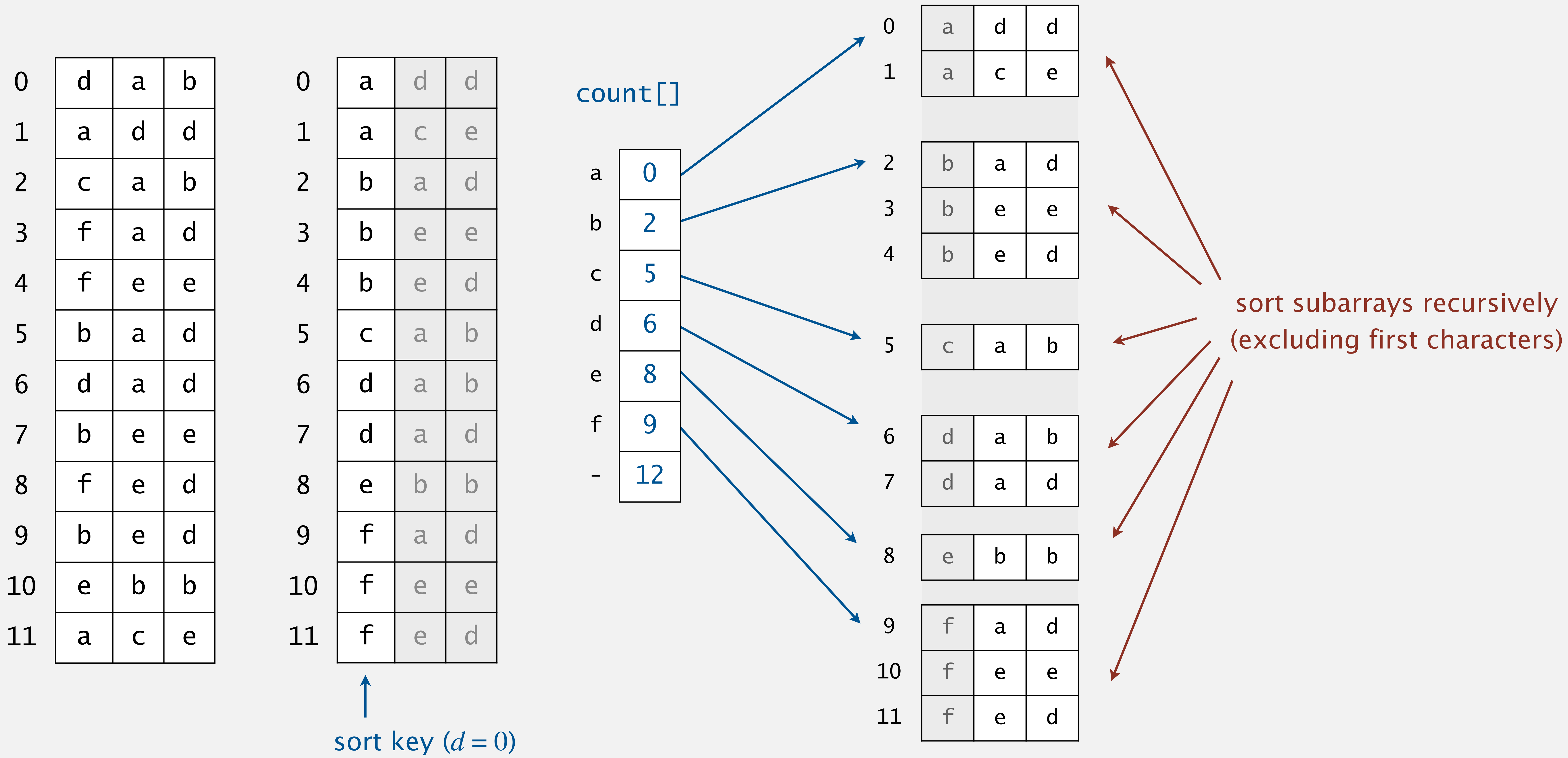
strings not sorted!

Most-significant-digit-first (MSD) radix sort

Overview.

- Partition array into R subarrays according to first character. ← use key-indexed counting
- Recursively sort all strings that start with each character. ← key-indexed counts delineate subarray boundaries

  (excluding the first characters in subsequent sorts)



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key ($d = 0$)

count[]

a	0
b	2
c	5
d	6
e	8
f	9
-	12

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort subarrays recursively
(excluding first characters)

MSD string sort (for fixed-length strings): Java implementation

```java
public static void sort(String[] a, int w)
{
   aux = new String[a.length];
   sort(a, aux, w, 0, a.length - 1, 0);
}

private static void sort(String[] a, String[] aux, int w, int lo, int hi, int d)
{
   if (hi <= lo || d == w) return;

   int[] count = new int[R+1];
   for (int i = lo; i <= hi; i++)
      count[a[i].charAt(d) + 1]++;
   for (int r = 0; r < R; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[a[i].charAt(d)]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   sort(a, aux, w, lo, lo + count[0] - 1, d+1);
   for (int r = 1; r < R; r++)
      sort(a, aux, w, lo + count[r-1], lo + count[r] - 1, d+1);

}
```

fixed-length *w* strings

recycles `aux[]` array
but not `count[]` array

sort `a[lo..hi]` assuming first `d`
characters already match

subarrays of length 0 or 1; or all *w* characters match

key-indexed counting
(using character *d*)

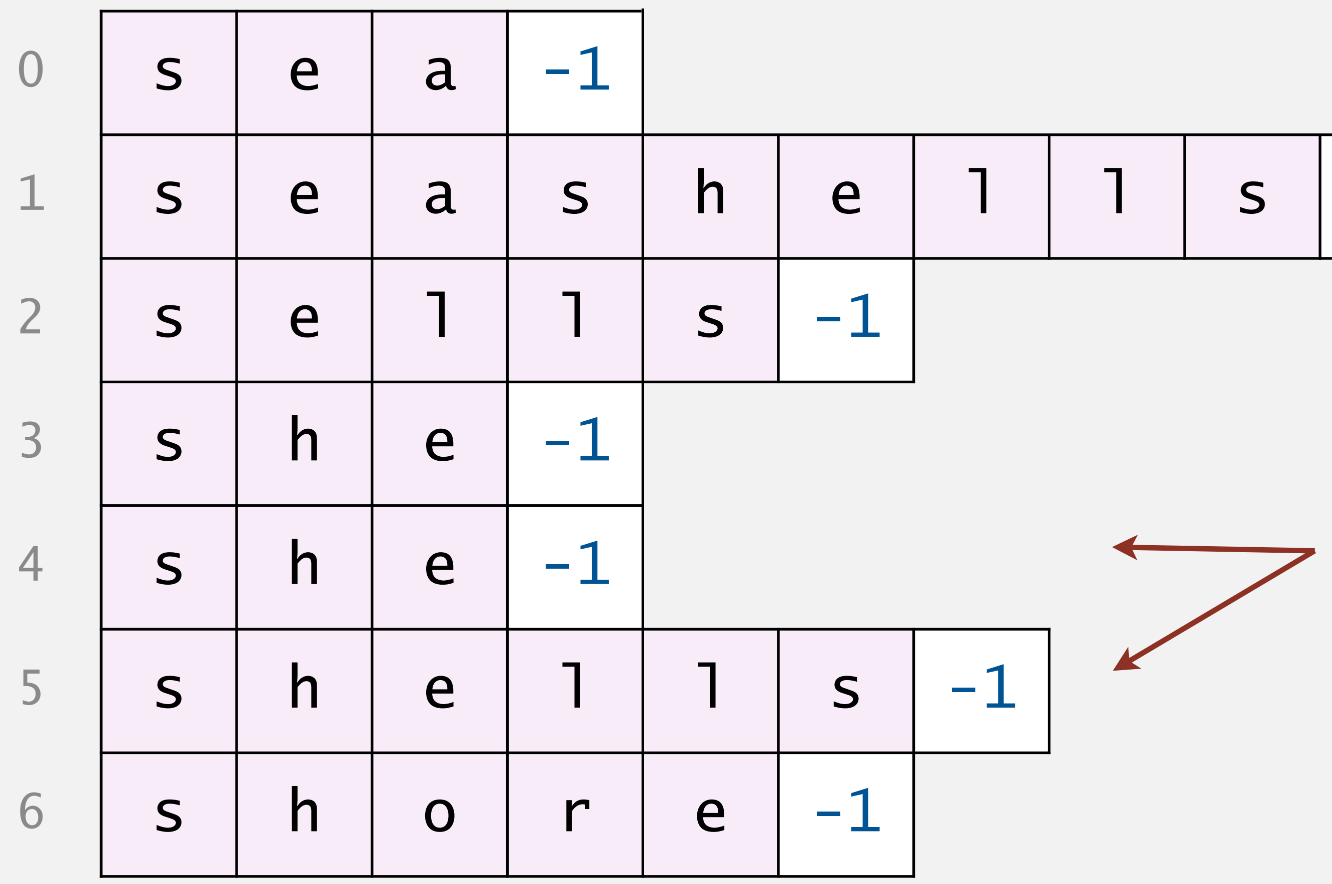
sort *R* subarrays recursively

at this place in code, `count[r]` = number of keys ≤ r

Variable-length strings

Useful trick. Treat strings as if they had an extra char at end (smaller than any char).

why smaller?



0	s	e	a	-1						
1	s	e	a	s	h	e	l	l	s	-1
2	s	e	l	l	s	-1				
3	s	h	e	-1						
4	s	h	e	-1						
5	s	h	e	l	l	s	-1			
6	s	h	o	r	e	-1				

"she" before "shells"

```
private static int charAt(String s, int d)
{
   if (d < s.length()) return s.charAt(d);
   else return -1;
}
```

C strings. Terminated with null character (`'\0'`) ⇒ no extra work needed.

Radix sorting: quiz 3

For which family of inputs is MSD sort likely to be faster than LSD sort?

A. Random strings.

B. All equal strings.

C. Both A and B.

D. Neither A nor B.

random	**all equal**
1EI0402	1DNB377
1HYL490	1DNB377
1ROZ572	1DNB377
2HXE734	1DNB377
2IYE230	1DNB377
2XOR846	1DNB377
3CDB573	1DNB377
3CVP720	1DNB377
3IGJ319	1DNB377
3KNA382	1DNB377
3TAV879	1DNB377
4CQP781	1DNB377
4QGI234	1DNB377
4YHV229	1DNB377

MSD string sort: performance

Observation. MSD examines just enough character to sort the keys.

Proposition. For random strings, MSD examines $\Theta(n \log_R n)$ characters.

Remark. This can be sublinear in the input size $\Theta(n\, w)$. ← `compareTo()` based sorts can also be sublinear

Proposition. In the worst case, MSD requires $\Theta(n + wR)$ extra space.

random	all equal
1 E I O 4 0 2	1 D N B 3 7 7
1 H Y L 4 9 0	1 D N B 3 7 7
1 R O Z 5 7 2	1 D N B 3 7 7
2 H X E 7 3 4	1 D N B 3 7 7
2 I Y E 2 3 0	1 D N B 3 7 7
2 X O R 8 4 6	1 D N B 3 7 7
3 C D B 5 7 3	1 D N B 3 7 7
3 C V P 7 2 0	1 D N B 3 7 7
3 I G J 3 1 9	1 D N B 3 7 7
3 K N A 3 8 2	1 D N B 3 7 7
3 T A V 8 7 9	1 D N B 3 7 7
4 C Q P 7 8 1	1 D N B 3 7 7
4 Q G I 2 3 4	1 D N B 3 7 7
4 Y H V 2 2 9	1 D N B 3 7 7

Summary of the performance of sorting algorithms

Frequency of calls to `compareTo()` and `charAt()`.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ n^2	¼ n^2	$\Theta(1)$	✔	`compareTo()`
mergesort	$n \log_2 n$	$n \log_2 n$	$\Theta(n)$	✔	`compareTo()`
quicksort	$1.39\ n \log_2 n$ *	$1.39\ n \log_2 n$ *	$\Theta(\log n)$ *		`compareTo()`
heapsort	$2\ n \log_2 n$	$2\ n \log_2 n$	$\Theta(1)$		`compareTo()`
LSD sort †	$2\ w\ n$	$2\ w\ n$	$\Theta(n + R)$	✔	`charAt()`
MSD sort ‡	$2\ w\ n$	$n \log_R n$	$\Theta(n + w\ R)$	✔	`charAt()`

* probabilistic

† fixed-length w keys

‡ average-length w keys

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

- MSD is much too slow for small subarrays.
- Essential for performance.

Optimization 1. Replace recursion with explicit stack.

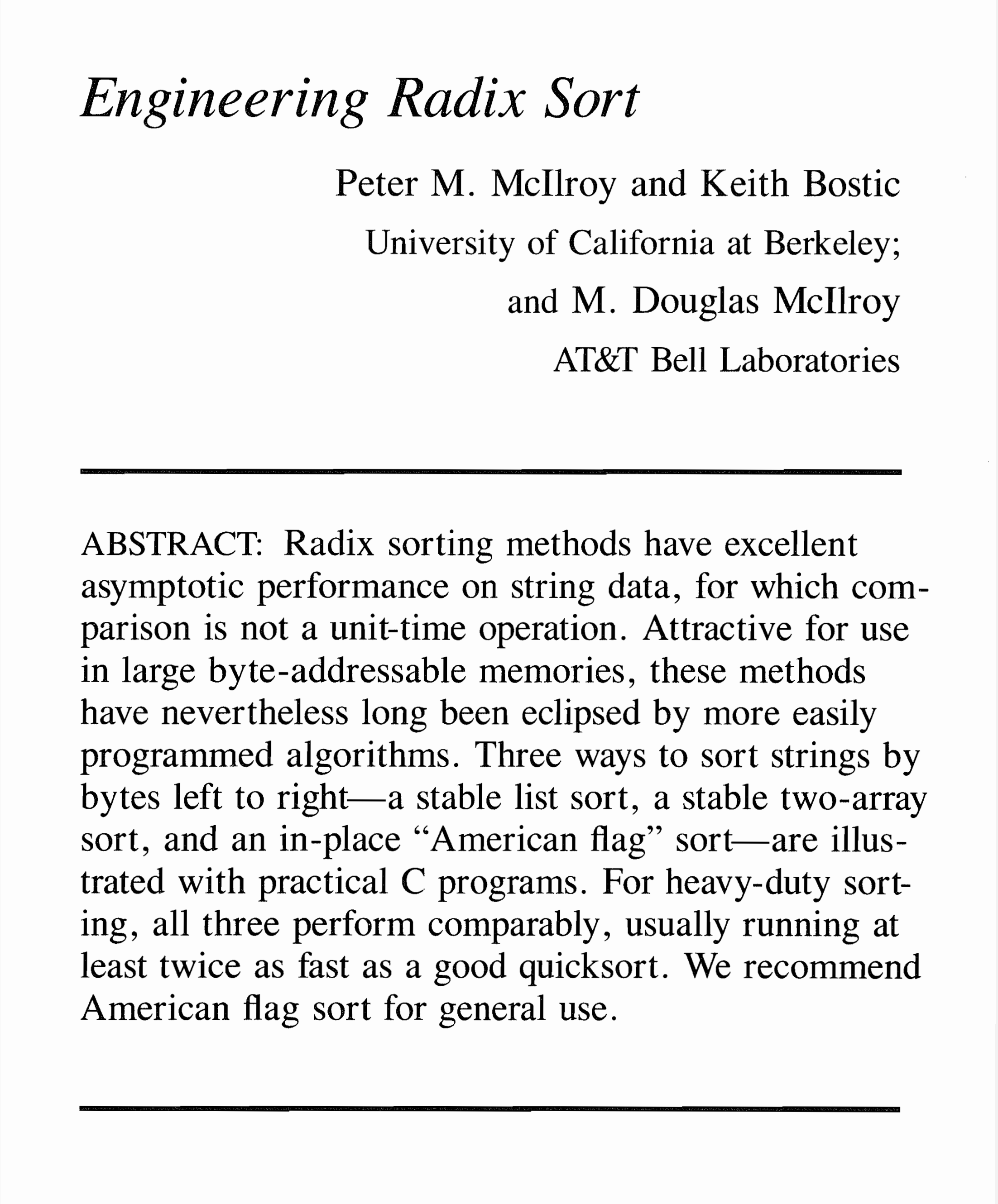
- Push subarrays to be sorted onto stack.
- One `count[]` array now suffices.

Optimization 2. Do R-way partitioning in place.

- Eliminates `aux[]` array.
- Sacrifices stability.

American national flag problem

Dutch national flag problem



Engineering Radix Sort

Peter M. McIlroy and Keith Bostic
University of California at Berkeley;
and M. Douglas McIlroy
AT&T Bell Laboratories

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place "American flag" sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

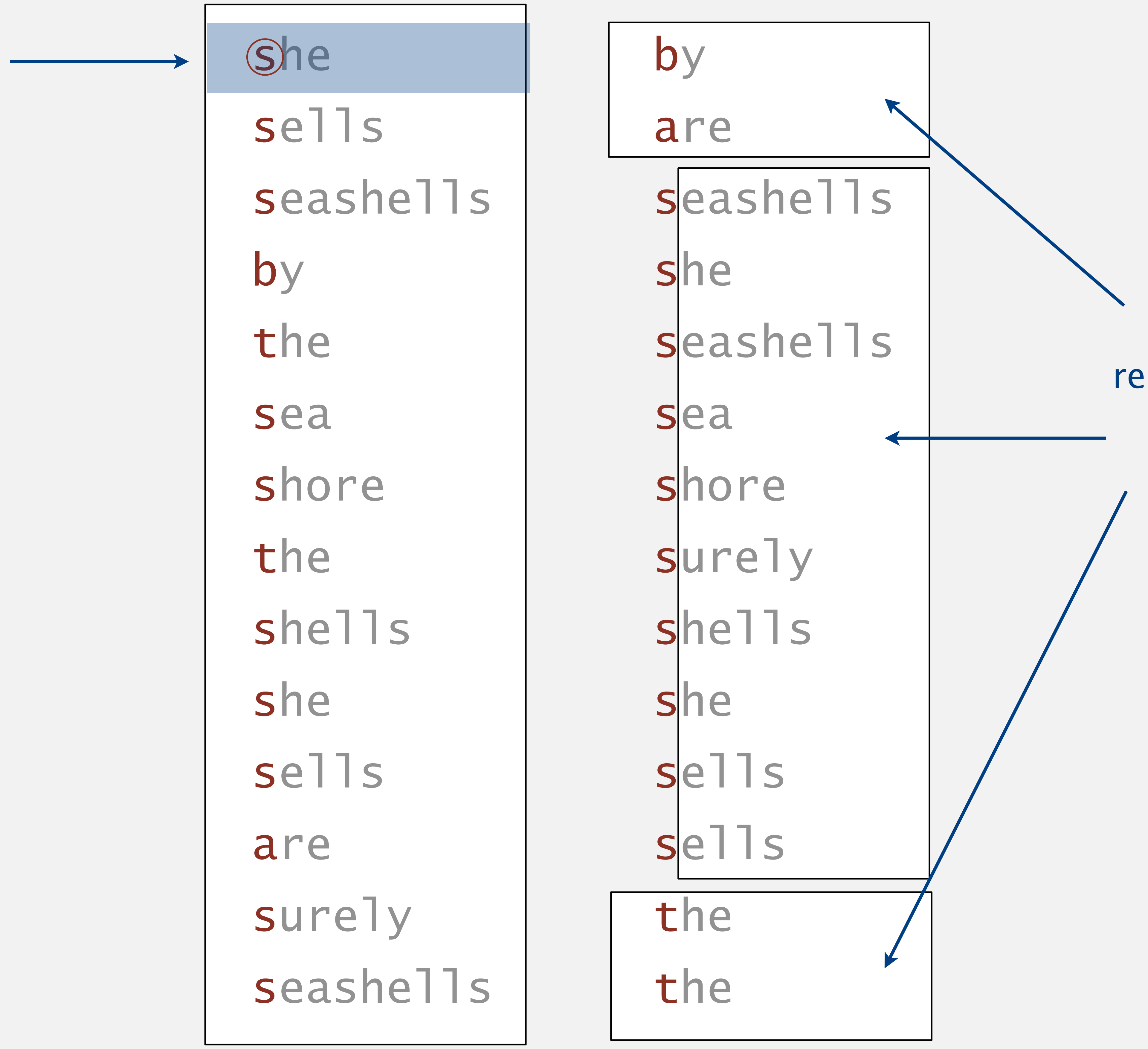
5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

3-way string quicksort

Overview.

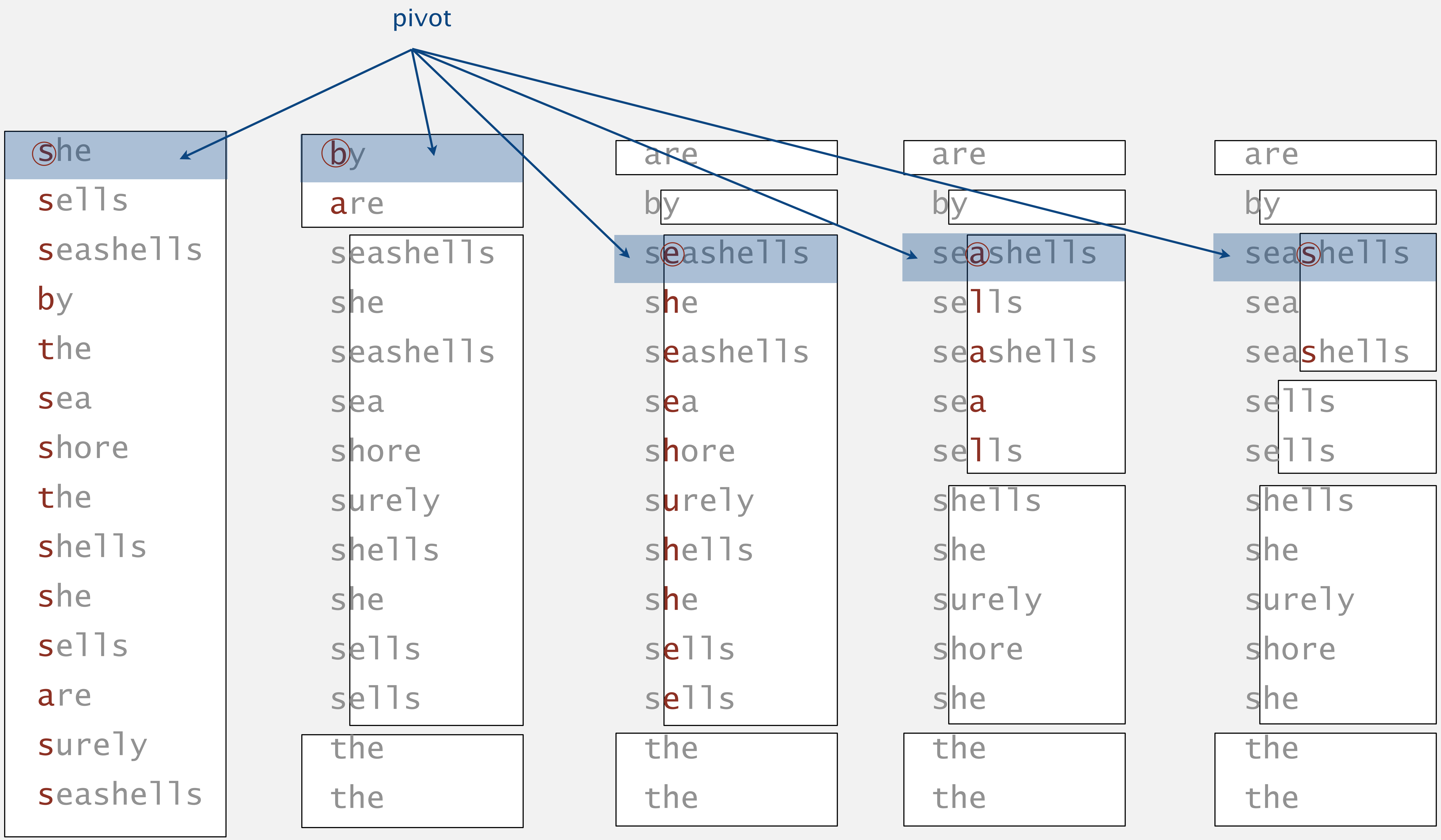
- Partition array into 3 subarrays according to first character of pivot. ← use Dijkstra 3-way partitioning algorithm
- Recursively sort 3 subarrays. ← exclude first character when sorting middle subarray (since known to be equal)



use first character of pivot to partition into “less”, “equal”, and “greater” subarrays

recursively sort 3 subarrays, excluding first character for middle subarray

3-way string quicksort: trace of recursive calls



she	by	are	are	are
sells	are	by	by	by
seashells	seashells	seashells	seashells	seashells
by	she	she	sells	sea
the	seashells	seashells	seashells	seashells
sea	sea	sea	sea	sells
shore	shore	shore	sells	sells
the	surely	surely	shells	shells
shells	shells	shells	she	she
she	she	she	surely	surely
sells	sells	sells	shore	shore
are	sells	sells	she	she
surely	the	the	the	the
seashells	the	the	the	the

Trace of first few recursive calls for 3-way string quicksort (subarrays of length 1 not shown)

3-way string quicksort: Java implementation

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int pivot = charAt(a[lo], d);

    int lt = lo, gt = hi;
    int i = lo + 1;
    while (i <= gt)
    {
       int c = charAt(a[i], d);
       if      (c < pivot) exch(a, lt++, i++);
       else if (c > pivot) exch(a, i, gt--);
       else                i++;
    }

    sort(a, lo, lt-1, d);
    if (pivot != -1) sort(a, lt, gt, d+1);
    sort(a, gt+1, hi, d);

}
```

sort `a[lo..hi]` assuming first *d* characters are equal

subarrays of length 0 or 1

Dijkstra 3-way partitioning (using character at index *d*)

sort 3 subarrays recursively

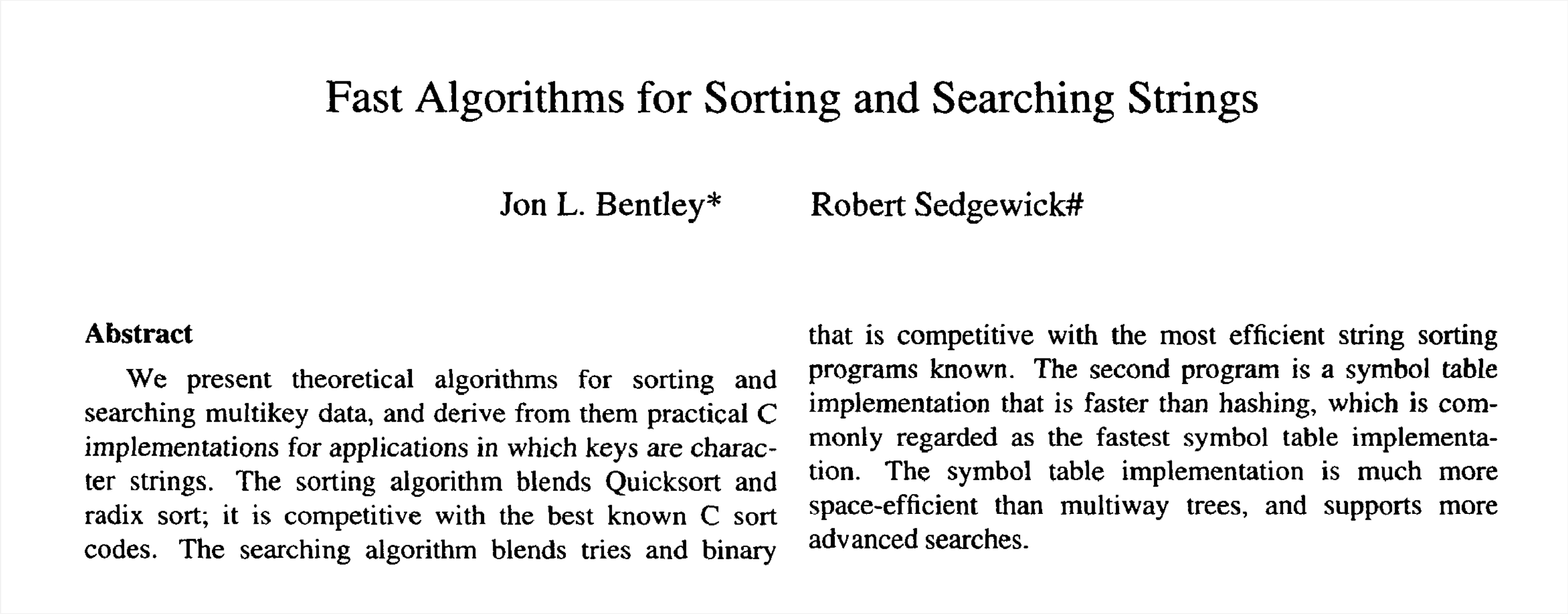
3-way string quicksort vs. competitors

3-way string quicksort vs. MSD sort.

- In-place; short inner loop; cache-friendly.
- Not stable.

3-way string quicksort vs. standard quicksort.

- Typically uses $\sim 2\, n \ln n$ character compares (instead of $\sim 2\, n \ln n$ string compares).
- Faster for keys with long common prefixes (and this is a common case!)



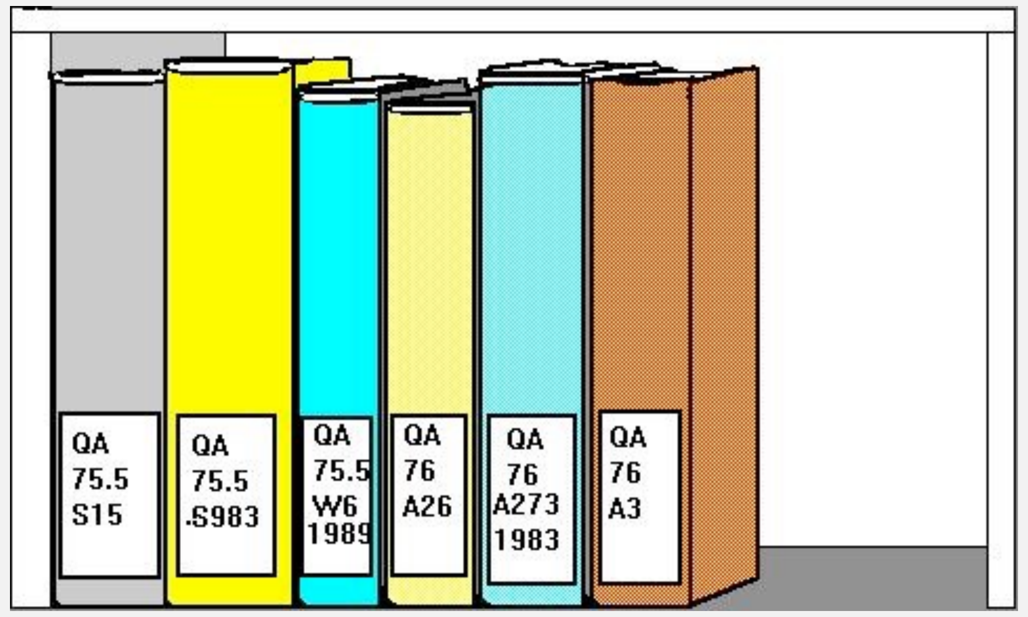
Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary

that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.



library of Congress call numbers

Bottom line. 3-way string quicksort is often the method of choice for sorting strings.

Summary of the performance of sorting algorithms

Frequency of calls to `compareTo()` and `charAt()`.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} n^2$	$\frac{1}{4} n^2$	$\Theta(1)$	✔	`compareTo()`
mergesort	$n \log_2 n$	$n \log_2 n$	$\Theta(n)$	✔	`compareTo()`
quicksort	$1.39\ n \log_2 n$ *	$1.39\ n \log_2 n$ *	$\Theta(\log n)$ *		`compareTo()`
heapsort	$2\ n \log_2 n$	$2\ n \log_2 n$	$\Theta(1)$		`compareTo()`
LSD sort †	$2\ w\ n$	$2\ w\ n$	$\Theta(n + R)$	✔	`charAt()`
MSD sort ‡	$2\ w\ n$	$n \log_R n$	$\Theta(n + w\ R)$	✔	`charAt()`
3-way string quicksort	$1.39\ w\ n \log_2 R$ *	$1.39\ n \log_2 n$ *	$\Theta(\log n + w)$ *		`charAt()`

* probabilistic
† fixed-length w keys
‡ average-length w keys

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

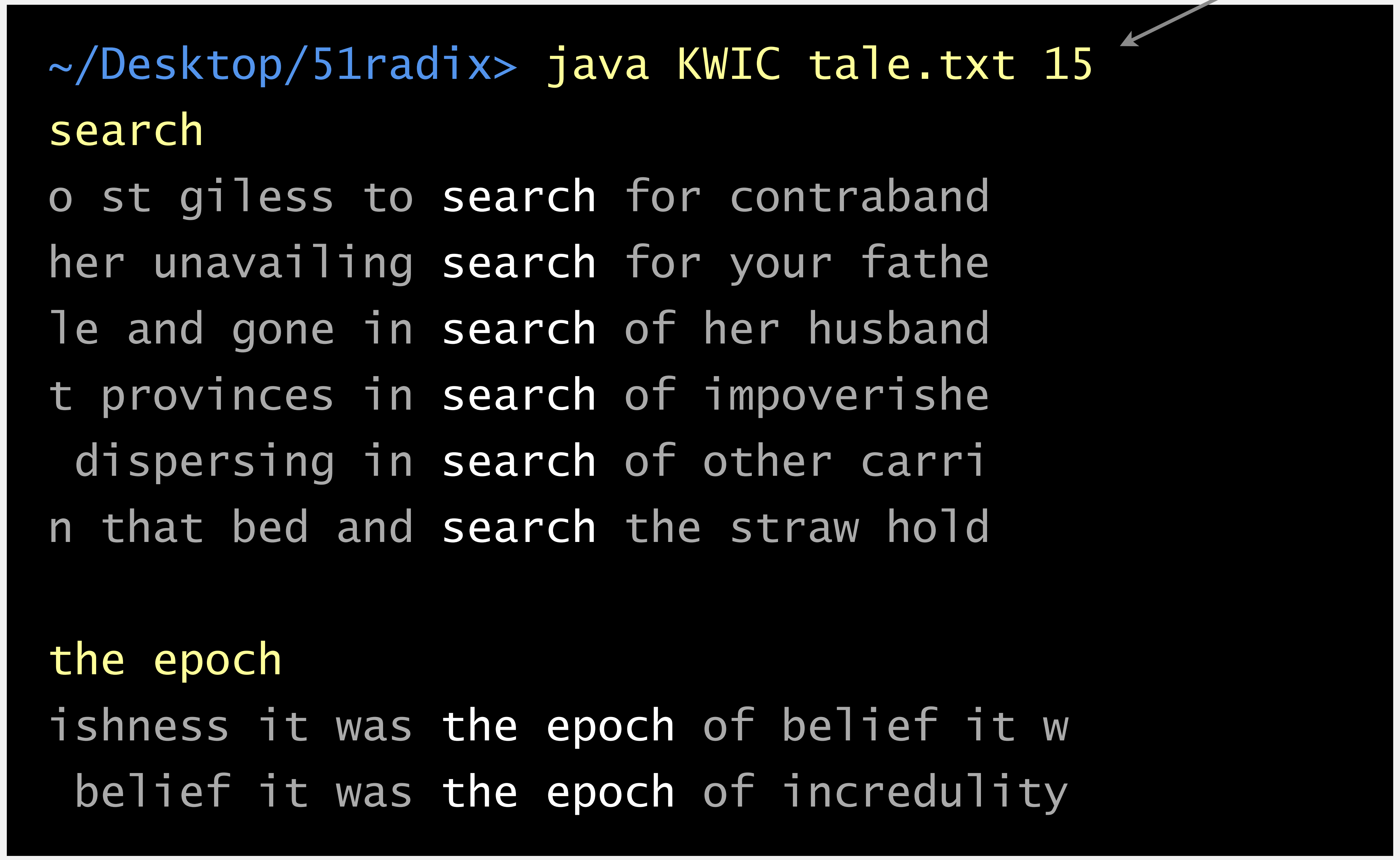
https://algs4.cs.princeton.edu

5.1 STRING SORTS

- *strings in Java*
- *key-indexed counting*
- *LSD radix sort*
- *MSD radix sort*
- *3-way radix quicksort*
- *suffix arrays*

Keyword-in-context search

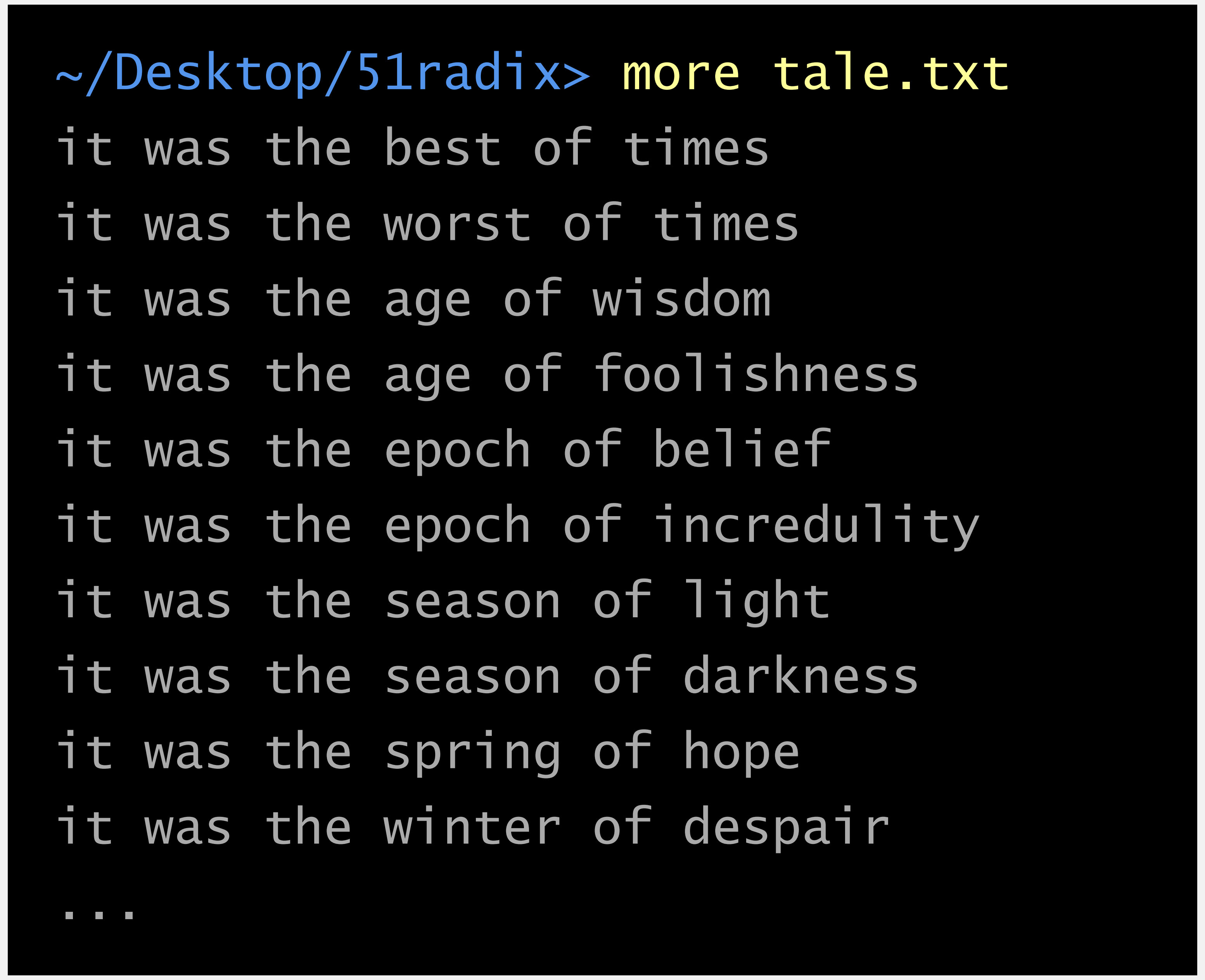
Given a text of n characters, preprocess it to enable fast substring search (find all occurrences of query string and surrounding context).



number of characters of surrounding context

```
~/Desktop/51radix> java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
 dispersing in search of other carri
n that bed and search the straw hold

the epoch
ishness it was the epoch of belief it w
 belief it was the epoch of incredulity
```



```
~/Desktop/51radix> more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
...
```

Applications. Linguistics, databases, web search, word processing, ….

Suffix sort

input string

i	t	w	a	s	b	e	s	t	i	t	w	a	s	w
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

form suffixes

0	i t w a s b e s t i t w a s w
1	t w a s b e s t i t w a s w
2	w a s b e s t i t w a s w
3	a s b e s t i t w a s w
4	s b e s t i t w a s w
5	b e s t i t w a s w
6	e s t i t w a s w
7	s t i t w a s w
8	t i t w a s w
9	i t w a s w
10	t w a s w
11	w a s w
12	a s w
13	s w
14	w

sort suffixes to bring query strings together

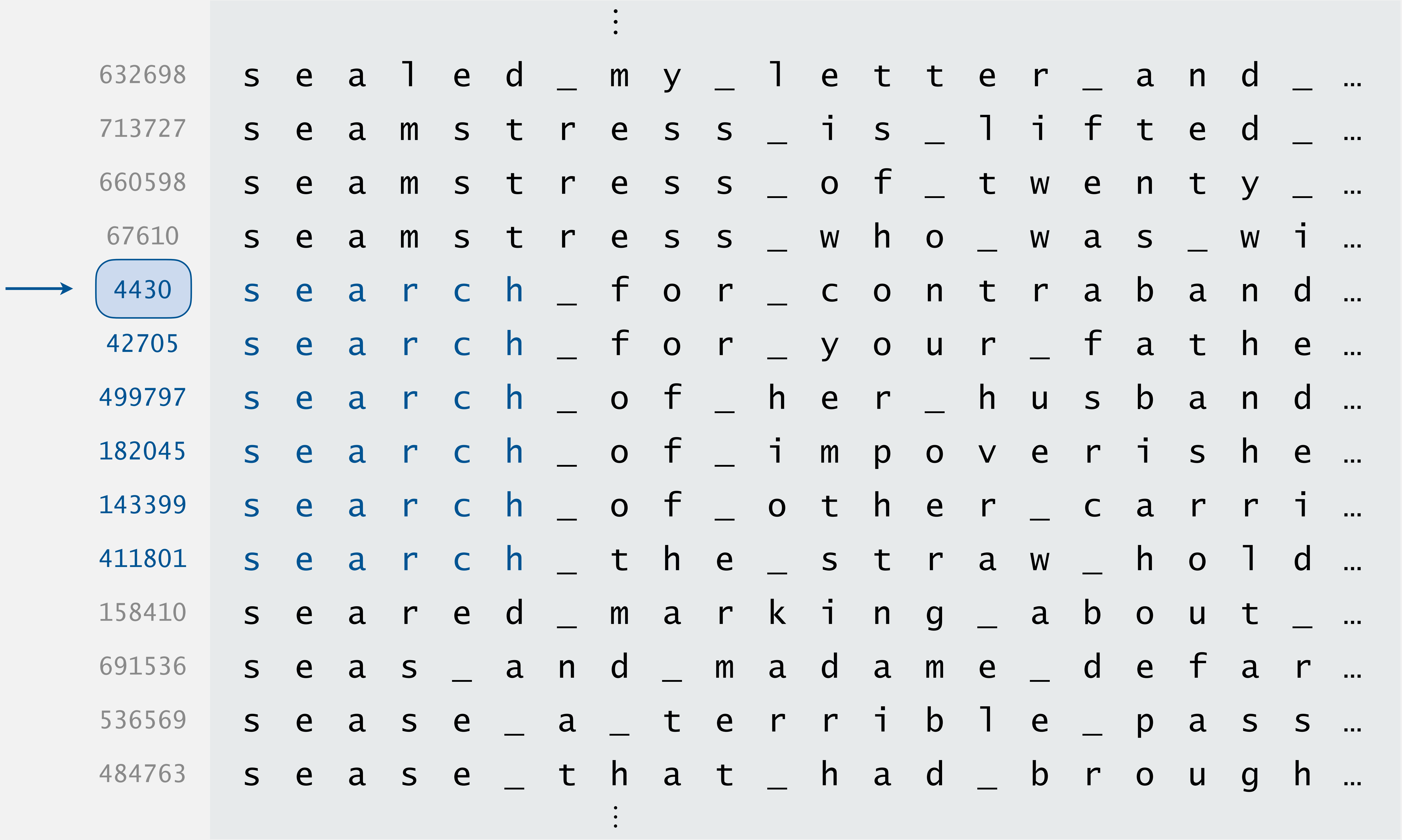
3	a s b e s t i t w a s w
12	a s w
5	b e s t i t w a s w
6	e s t i t w a s w
0	i t w a s b e s t i t w a s w
9	i t w a s w
4	s b e s t i t w a s w
7	s t i t w a s w
13	s w
8	t i t w a s w
1	t w a s b e s t i t w a s w
10	t w a s w
14	w
2	w a s b e s t i t w a s w
11	w a s w

array of suffix indices
(in sorted order)

Keyword-in-context search: suffix-sorting solution

- Preprocess: suffix sort the text.
- Query: binary search for query; scan until mismatch.

KWIC search for “search” in Tale of Two Cities



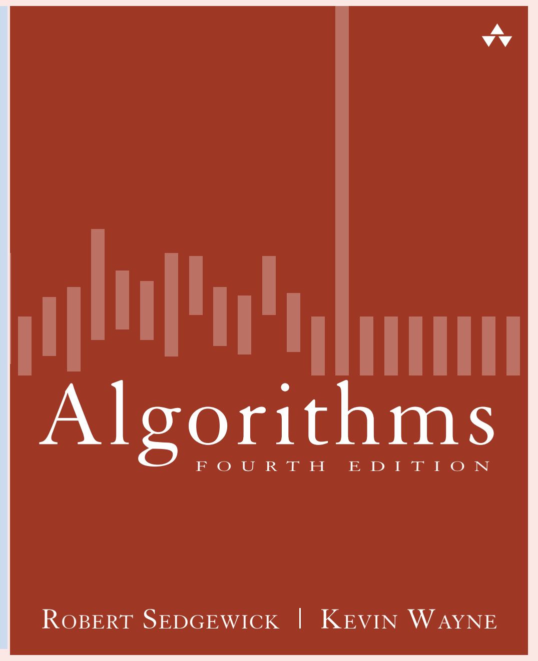
	⋮
632698	sealed_my_letter_and_…
713727	seamstress_is_lifted_…
660598	seamstress_of_twenty_…
67610	seamstress_who_was_wi…
4430	search_for_contraband…
42705	search_for_your_fathe…
499797	search_of_her_husband…
182045	search_of_impoverishe…
143399	search_of_other_carri…
411801	search_the_straw_hold…
158410	seared_marking_about_…
691536	seas_and_madame_defar…
536569	sease_a_terrible_pass…
484763	sease_that_had_brough…
	⋮

Radix sorting: quiz 4

How much memory as a function of n?

```
String[] suffixes = new String[n];
for (int i = 0; i < n; i++)
    suffixes[i] = s.substring(i, n);

Arrays.sort(suffixes);
```



3rd printing (2012)

A. $\Theta(1)$

B. $\Theta(n)$

C. $\Theta(n \log n)$

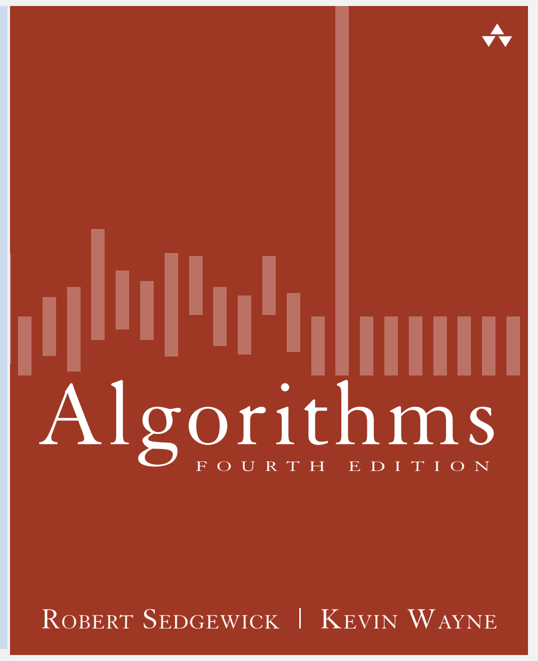
D. $\Theta(n^2)$

War story

Q. How to efficiently form (and sort) the n suffixes?

```
String[] suffixes = new String[n];
for (int i = 0; i < n; i++)
    suffixes[i] = s.substring(i, n);

Arrays.sort(suffixes);
```



3rd printing (2012)

input file	characters	Java 7u5	Java 7u6
`amendments.txt`	18 K	0.25 sec	2.0 sec
`aesop.txt`	192 K	1.0 sec	*out of memory*
`mobydick.txt`	1.2 M	7.6 sec	*out of memory*
`chromosome11.txt`	7.1 M	61 sec	*out of memory*

$\Theta(n^2)$ time and space to form suffixes!

The String data type: Java 7u6 implementation

```
public final class String implements Comparable<String>
{
   private char[] value;  // sequence of characters in string
   private int hash;      // cache of hashCode()
   …
```

String s = "Hello, World";

`value[]`	H	E	L	L	O	,		W	O	R	L	D
	0	1	2	3	4	5	6	7	8	9	10	11

String t = s.substring(7, 12);

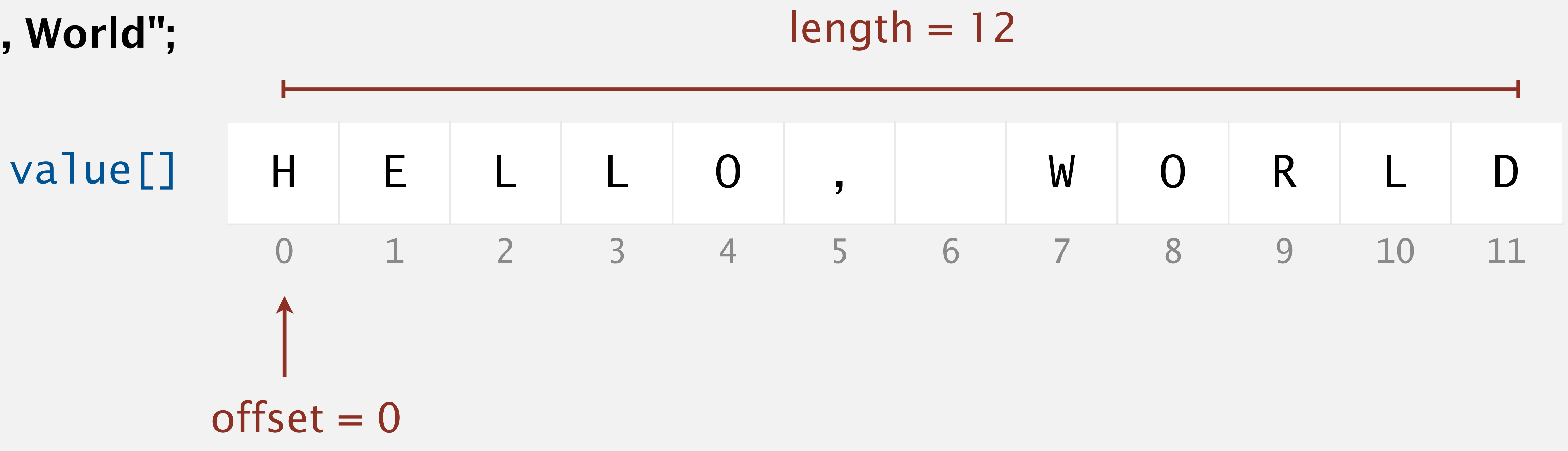
(allocates new char[] array ⇒ linear extra memory)

`value[]`	W	O	R	L	D
	0	1	2	3	4

The String data type: Java 7u5 implementation

```
public final class String implements Comparable<String>
{
   private char[] value;  // shared character array
   private int offset;    // index of first char in string
   private int length;    // length of string
   private int hash;      // cache of hashCode()
   …
```

String s = "Hello, World";



length = 12

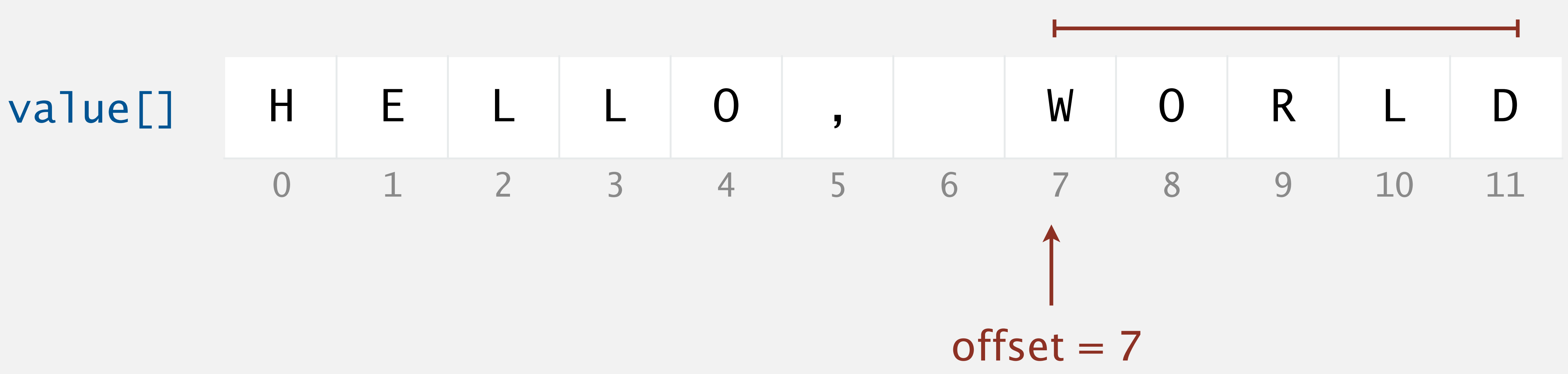
value[]	H	E	L	L	O	,		W	O	R	L	D
	0	1	2	3	4	5	6	7	8	9	10	11

offset = 0

String t = s.substring(7, 12);

(reuses original char[] array ⇒ constant extra memory)

length = 5



value[]	H	E	L	L	O	,		W	O	R	L	D
	0	1	2	3	4	5	6	7	8	9	10	11

offset = 7

The String data type: performance

String data type (in Java). Sequence of characters (immutable).

Java 7u5. Immutable `char[]` array, offset, length, hash cache.

Java 7u6. Immutable `char[]` array, hash cache.

operation	Java 7u5	Java 7u6
length	1	1
indexing	1	1
concatenation	$m + n$	$m + n$
substring extraction	1	n
immutable?	✔	✔
memory	$64 + 2n$	$56 + 2n$

A Reddit exchange

> I'm the author of the substring() change. As has been suggested in the analysis here there were two motivations for the change
>
> - Reduce the size of String instances. Strings are typically 20-40% of common apps footprint.
> - Avoid memory leakage caused by retained substrings holding the entire character array.

bondolo

> Changing this function, in a bugfix release no less, was totally irresponsible. It broke backwards compatibility for numerous applications with errors that didn't even produce a message, just freezing and timeouts... All pain, no gain. Your work was not just vain, it was thoroughly destructive, even beyond its immediate effect.

cypherpunks

http://www.reddit.com/r/programming/comments/1qw73v/til_oracle_changed_the_internal_string

Suffix sort

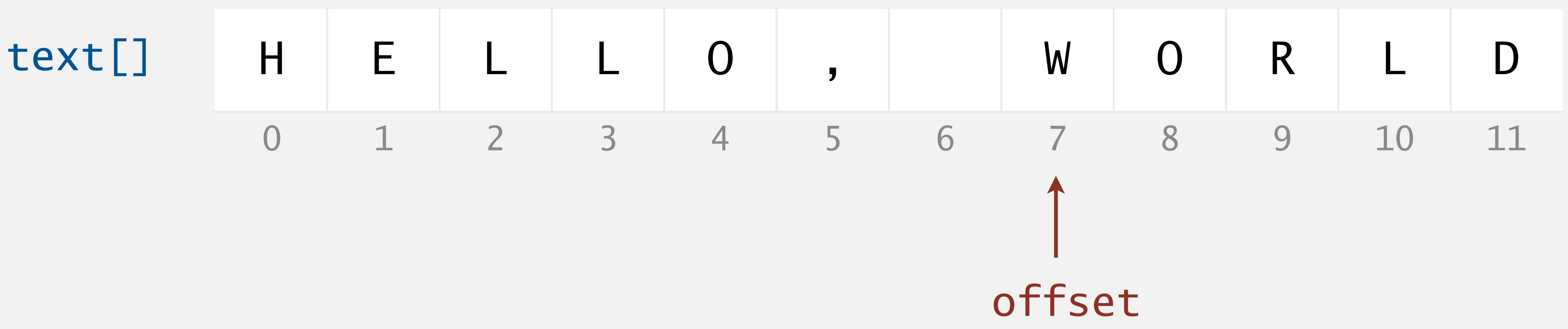
Q. How to efficiently form (and sort) suffixes in Java 7u6?

A. Define `Suffix` class à la Java 7u5 `String` representation.

```
public class Suffix implements Comparable<Suffix>
{
   private final String text;
   private final int offset;

   public Suffix(String text, int offset) {
      this.text = text;
      this.offset = offset;
   }

   public int length()                { return text.length() - offset;   }
   public char charAt(int i)          { return text.charAt(offset + i);  }
   public int compareTo(Suffix that)  { /* see textbook */               }
}
```



text[]	H	E	L	L	O	,		W	O	R	L	D
	0	1	2	3	4	5	6	7	8	9	10	11

offset

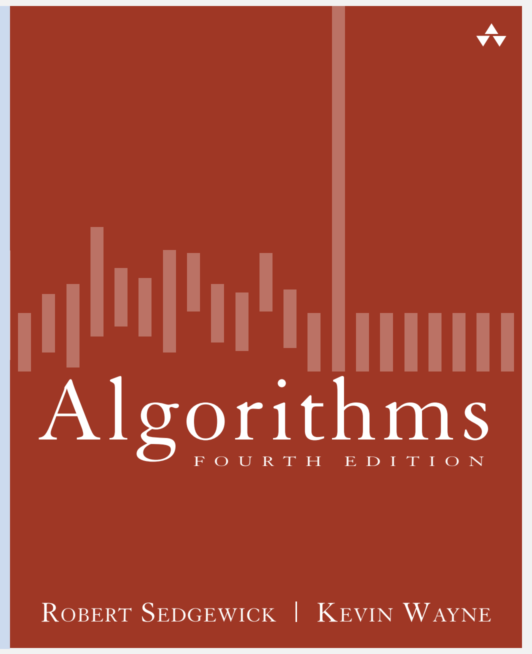
Suffix sort

Q. How to efficiently form (and sort) suffixes in Java 7u6?

A. Define `Suffix` class à la Java 7u5 `String` representation.

```
Suffix[] suffixes = new Suffix[n];
for (int i = 0; i < n; i++)
    suffixes[i] = new Suffix(s, i);

Arrays.sort(suffixes);
```



4th printing (2013)

Optimizations. [5× faster and 32× less memory than Java 7u5 version]

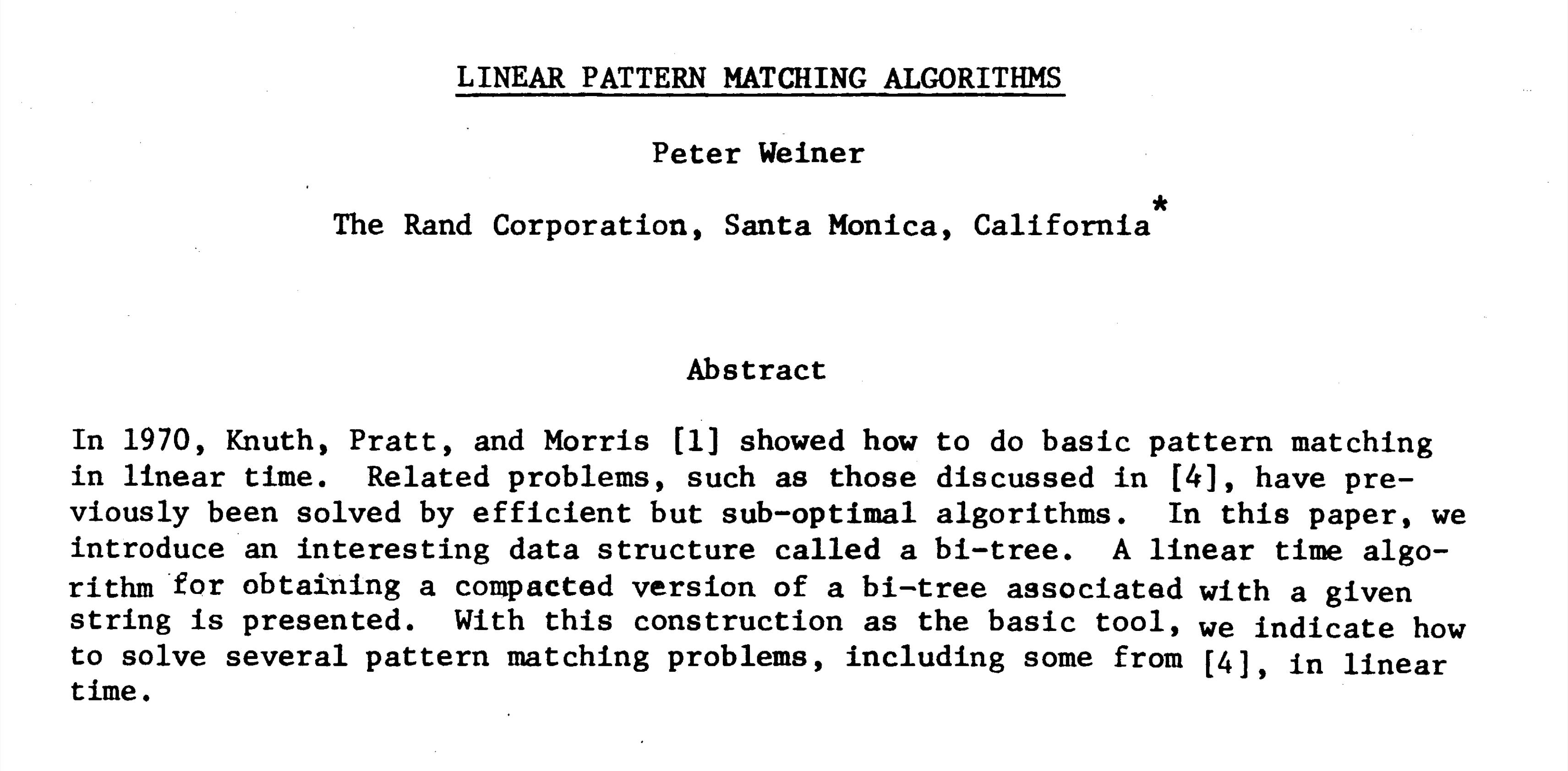
- Use 3-way string quicksort instead of `Arrays.sort()`.
- Manipulate suffix offsets directly instead of via explicit `Suffix` objects.

Suffix arrays: theory

Conjecture. [Knuth 1970] Impossible to compute suffix array in $\Theta(n)$ time.

Proposition. [Weiner 1973] Can solve in $\Theta(n)$ time (suffix trees).

“ has no practical virtue… but a historic monument in the area of string processing. ”



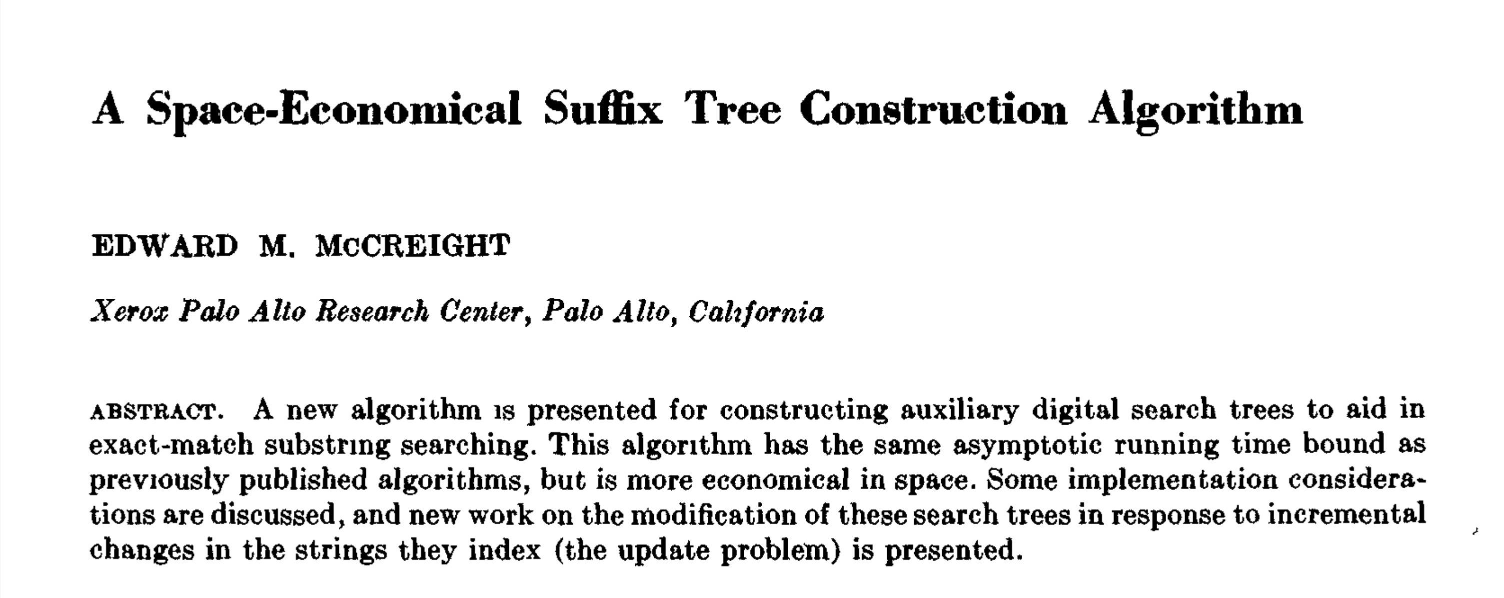
LINEAR PATTERN MATCHING ALGORITHMS

Peter Weiner

The Rand Corporation, Santa Monica, California*

Abstract

In 1970, Knuth, Pratt, and Morris [1] showed how to do basic pattern matching in linear time. Related problems, such as those discussed in [4], have previously been solved by efficient but sub-optimal algorithms. In this paper, we introduce an interesting data structure called a bi-tree. A linear time algorithm for obtaining a compacted version of a bi-tree associated with a given string is presented. With this construction as the basic tool, we indicate how to solve several pattern matching problems, including some from [4], in linear time.



A Space-Economical Suffix Tree Construction Algorithm

EDWARD M. McCREIGHT

Xerox Palo Alto Research Center, Palo Alto, California

ABSTRACT. A new algorithm is presented for constructing auxiliary digital search trees to aid in exact-match substring searching. This algorithm has the same asymptotic running time bound as previously published algorithms, but is more economical in space. Some implementation considerations are discussed, and new work on the modification of these search trees in response to incremental changes in the strings they index (the update problem) is presented.

On–line construction of suffix trees [1]

Esko Ukkonen

Department of Computer Science, University of Helsinki,
P. O. Box 26 (Teollisuuskatu 23), FIN–00014 University of Helsinki, Finland
Tel.: +358-0-7084172, fax: +358-0-7084441
Email: `ukkonen@cs.Helsinki.FI`

Suffix arrays: practice

Applications. Bioinformatics, information retrieval, data compression, …

Many ingenious algorithms.

- Constants and memory footprint very important.
- State-of-the art still changing.

year	algorithm	worst case	memory
1991	**Manber–Myers**	$n \log n$	$8\ n$
1999	**Larsson–Sadakane**	$n \log n$	$8\ n$
2003	**Kärkkäinen–Sanders**	n	$13\ n$
2003	**Ko–Aluru**	n	$10\ n$
2008	**divsufsort2**	$n \log n$	$5\ n$
2010	**sais**	n	$6\ n$

see lecture videos

about 10× faster than Manber–Myers

good choices (libdivsufsort)

String sorting summary

We can develop linear-time sorts.

- Key compares not necessary for string keys.
- Use characters as index in an array.

We can develop sublinear-time sorts.

- Input size = total number of characters (not number of strings).
- Not all of the characters have to be examined.

Long strings are rarely random in practice.

- Goal is often to learn the structure!
- May need specialized algorithms.