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*

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.

- Supports 7-bit ASCII.
- Represents only 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

A	á	∂	💩
U+0041	U+00E1	U+2202	U+1F4A9

some Unicode characters

Java char data type. A 16-bit unsigned integer. ← between 0 and $2^{16} - 1 = 65{,}535$

- Supports 16-bit Unicode 1.0.1. ← 7,161 characters
- Supports 21-bit Unicode 10.0 (via UTF-8). ← 136,755 characters and emoji

I 💖 Unicode

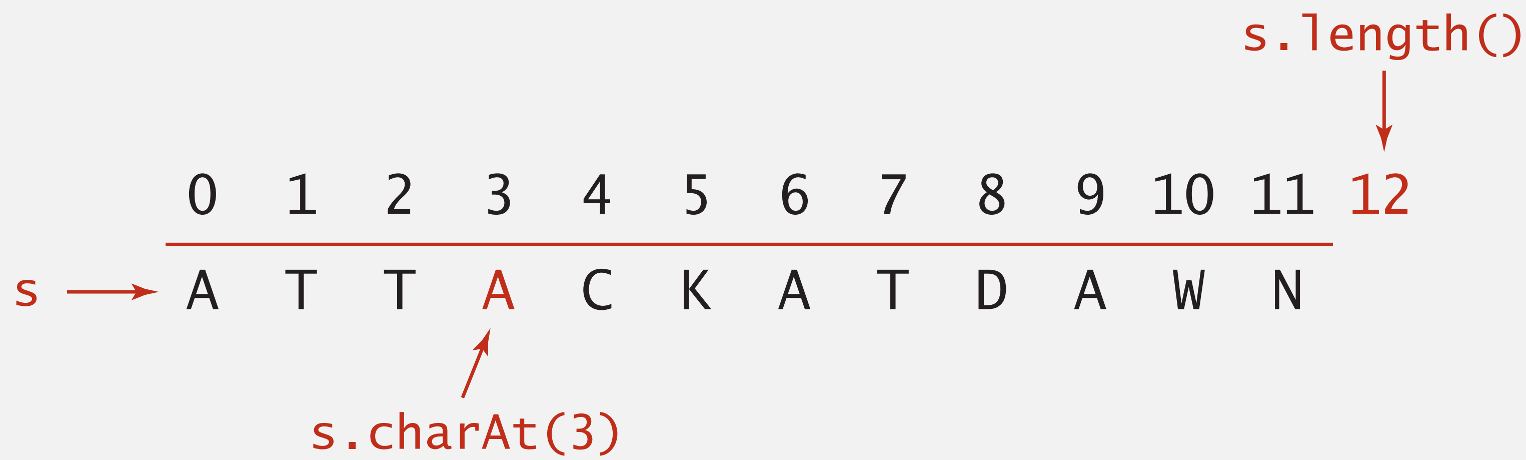
U+1F496

💖

The String data type (in Java)

String data type. Immutable sequence of characters.

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



operation	description	Java	running time
length	number of characters	`s.length()`	1
indexing	i^{th} character	`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.

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

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*

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 operations.

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

* probabilistic

`compareTo()` not constant time for strings

Lower bound. $\sim n \lg n$ compares required by any compare-based algorithm.

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

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

input *name*	*section*	sorted result *(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 integers between 0 and $R - 1$.

$R = 6$

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- 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 integers between 0 and $R - 1$.

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

count frequencies →

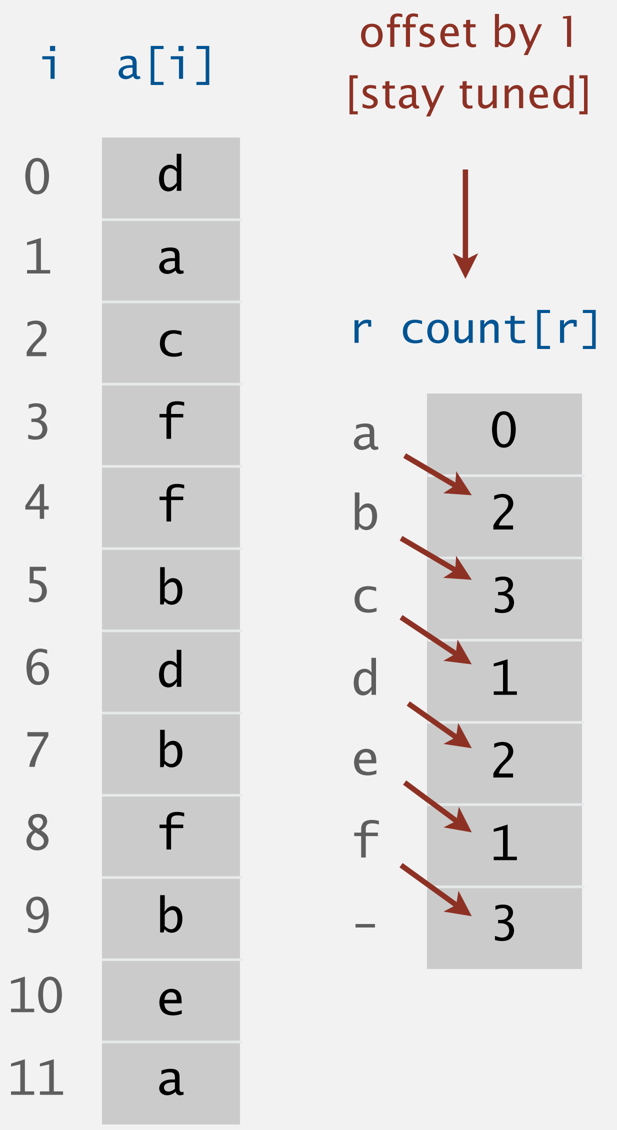
```
int n = a.length;
int[] count = new int[R+1];

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];
```



Key-indexed counting demo

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

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- 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 integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- 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];
```

move items →

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 integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- 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

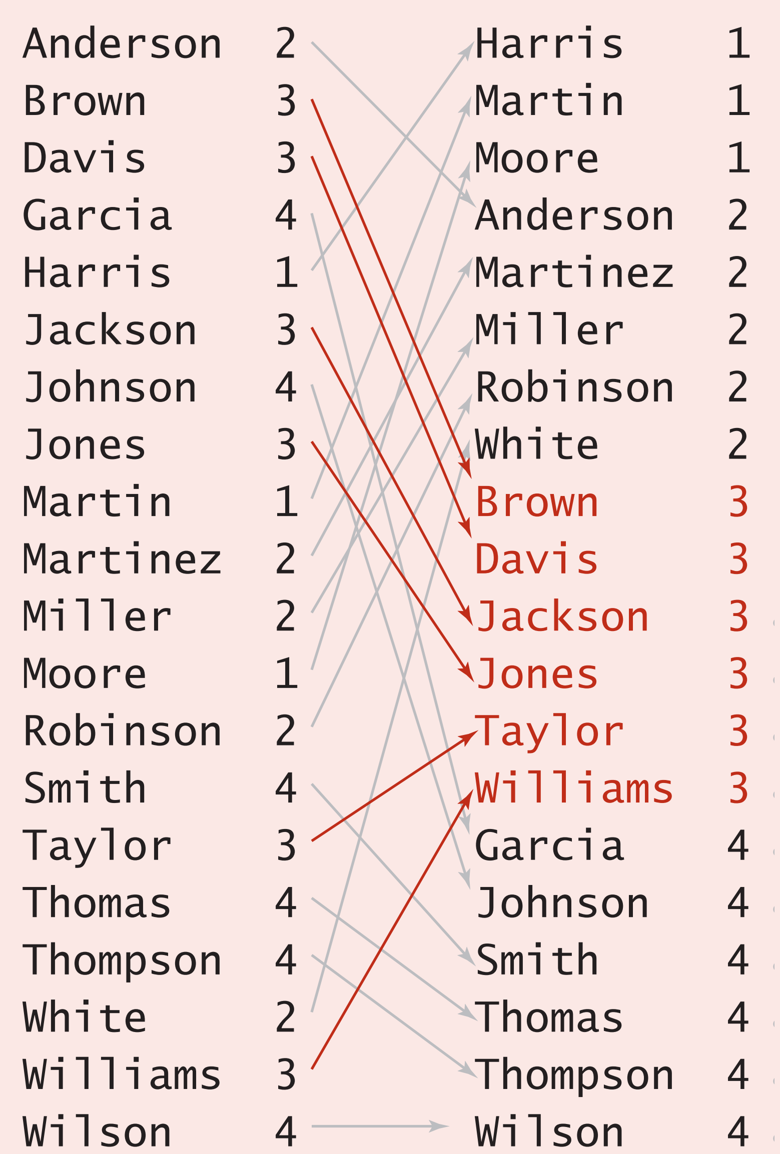
Which of the following are properties of key-indexed counting?

A. Running time proportional to $n + R$.

B. Extra space proportional to $n + R$.

C. Stable.

D. All of the above.



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 d^{th} character as the key (using key-indexed counting).

	input	sort key ($d = 2$)	sort key ($d = 1$)	sort key ($d = 0$)
0	d a b	d a b	d a b	a c e
1	a d d	c a b	c a b	a d d
2	c a b	e b b	f a d	b a d
3	f a d	a d d	b a d	b e d
4	f e e	f a d	d a d	b e e
5	b a d	b a d	e b b	c a b
6	d a d	d a d	a c e	d a b
7	b e e	f e d	a d d	d a d
8	f e d	b e d	f e d	e b b
9	b e d	f e e	b e d	f a d
10	e b b	b e e	f e e	f e d
11	a c e	a c e	b e e	f e e

sort is stable (arrows do not cross)

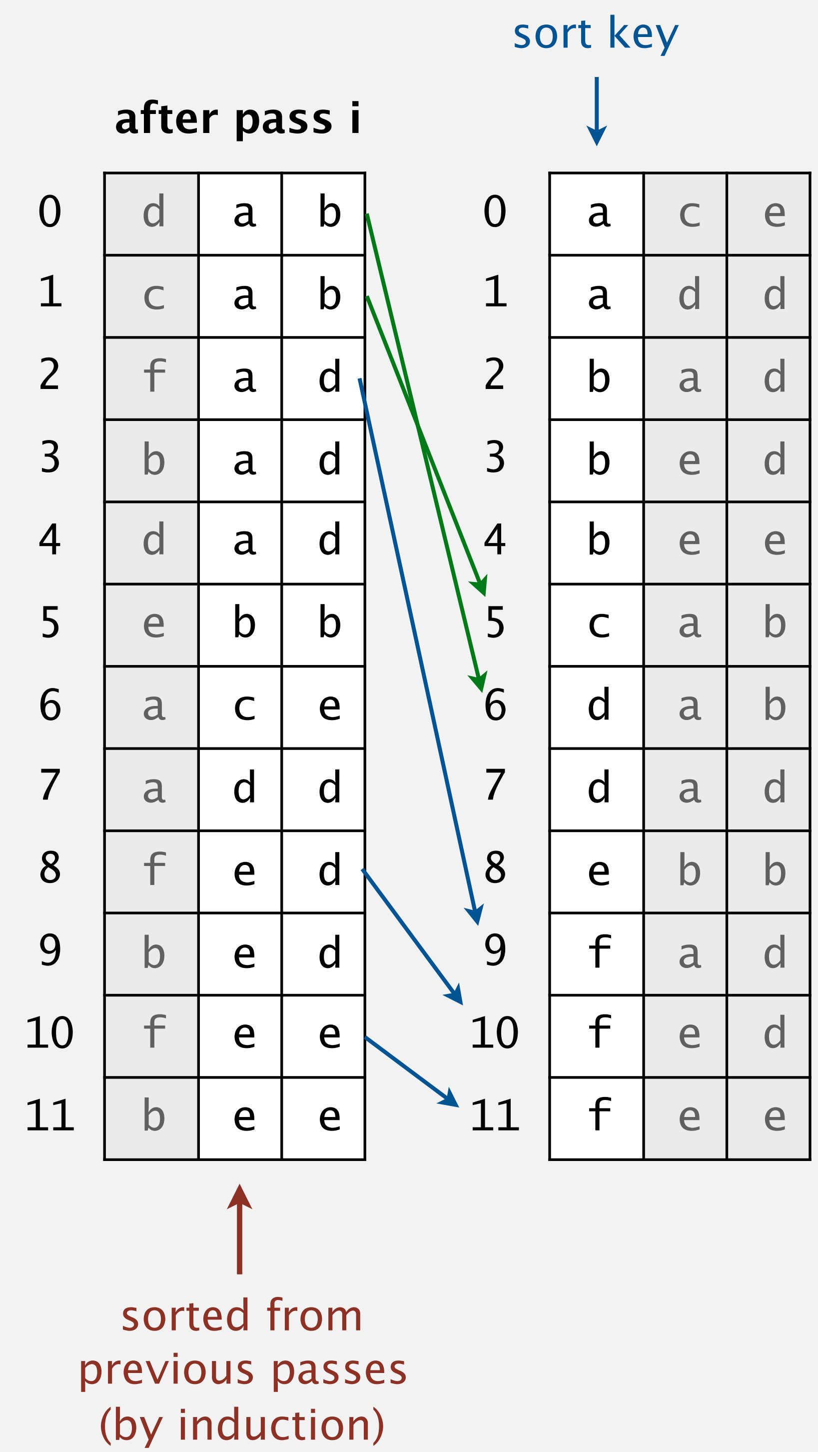
sorted!

LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on 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.
  - 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



Proposition. LSD sort is stable.

Pf. Key-indexed counting is stable.

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

```
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];
      }
   }
}
```

fixed-length w strings

radix R

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

key-indexed counting

Summary of the performance of sorting algorithms

Frequency of operations.

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

* probabilistic

† fixed-length w keys

1 call to `compareTo()` can involve as many as w calls to `charAt()`

Q. What if strings are not all of same length?

Radix sorting: quiz 2

Which sorting method to use to sort 1 million 32-bit integers?

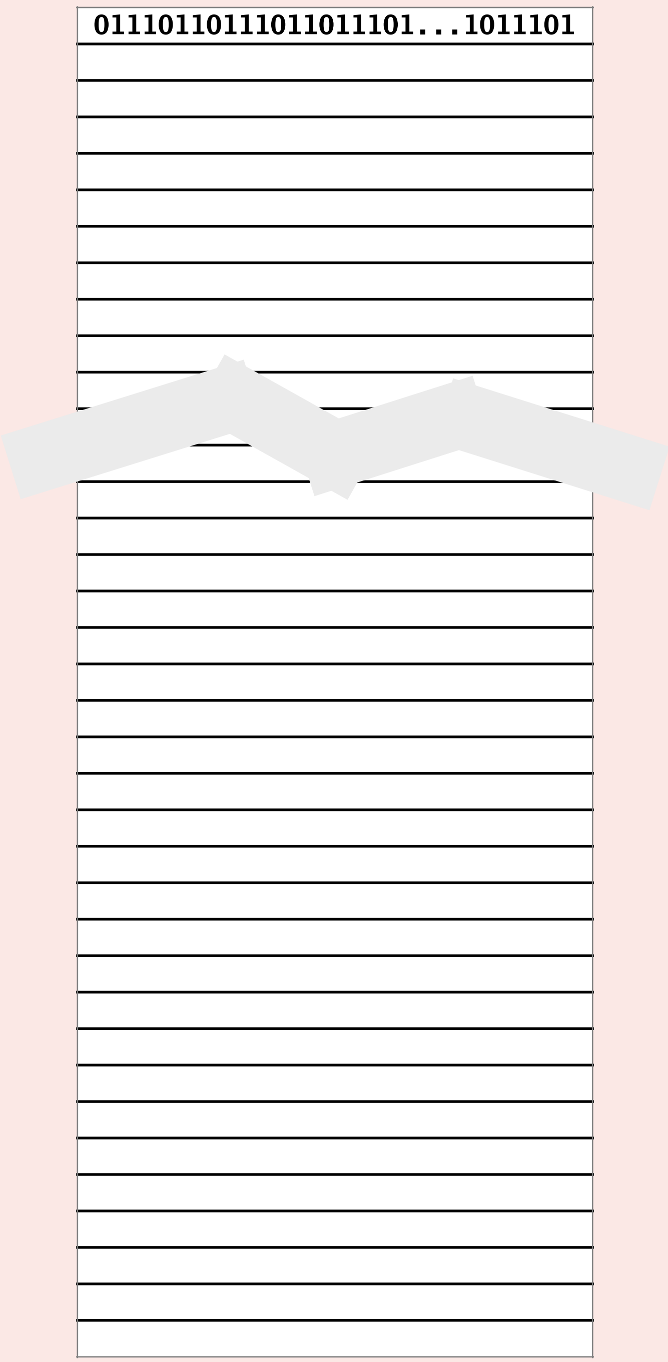
A. Insertion sort.

B. Mergesort.

C. Quicksort.

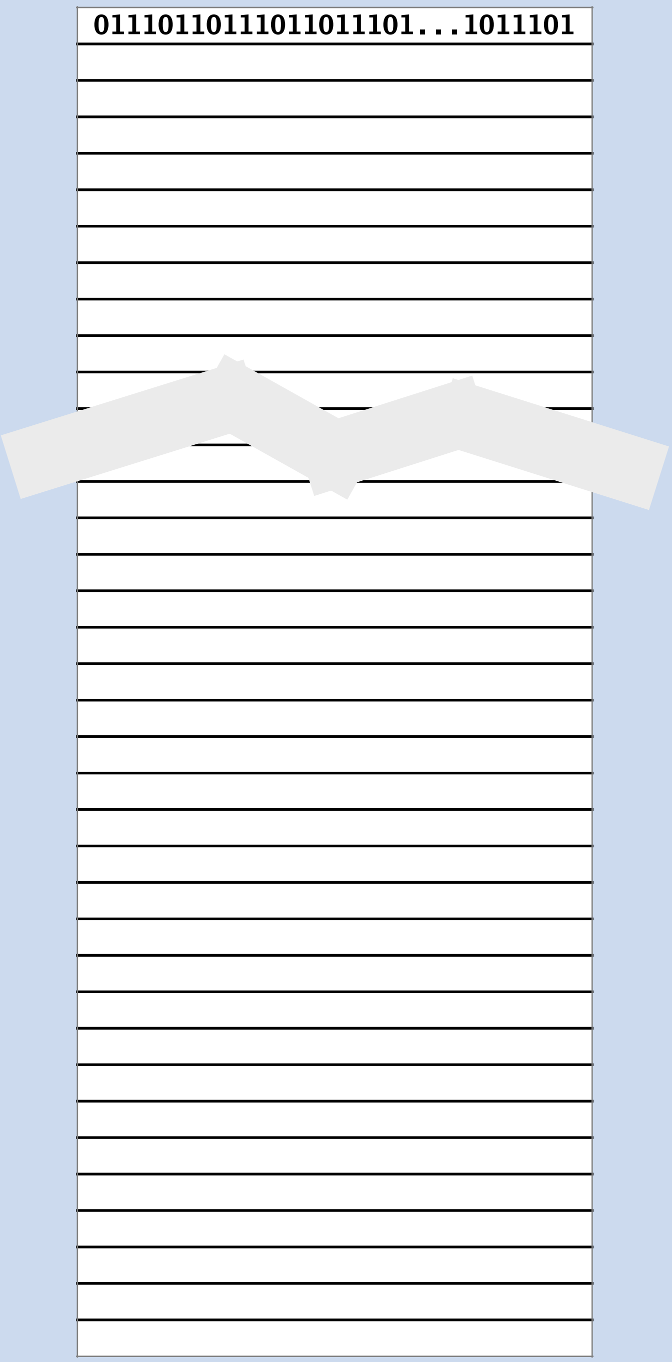
D. Heapsort.

E. LSD radix sort.



Sort Array of 128-Bit Numbers

Problem. Sort huge array of random 128-bit numbers.
Ex. Supercomputer sort, internet router.



Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD radix sort.

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 d^{th} character as the key (using key-indexed counting).

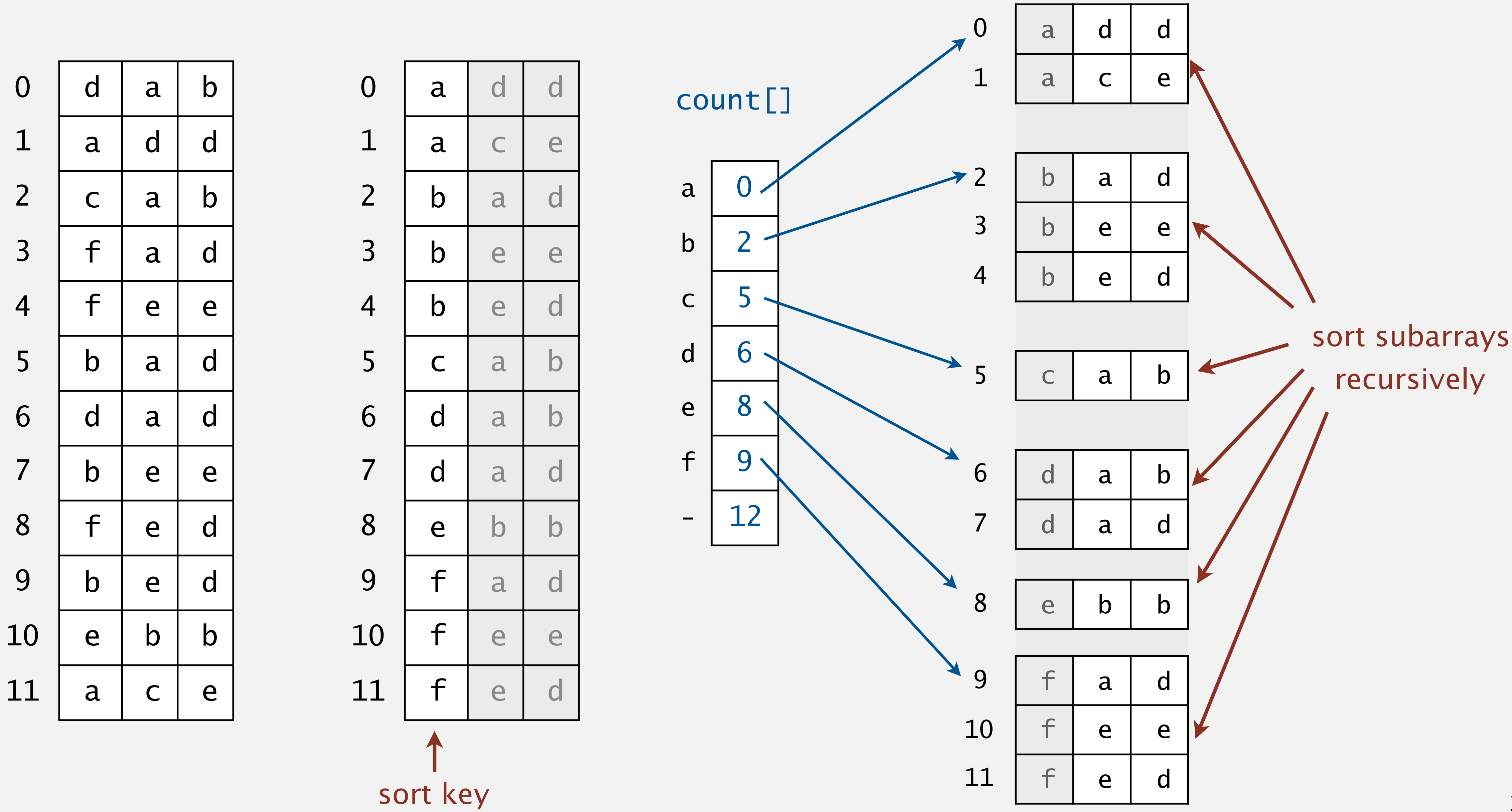
	input	sort key ($d = 0$)	sort key ($d = 1$)	sort key ($d = 2$)
0	d a b	a d d	b a d	c a b
1	a d d	a c e	c a b	d a b
2	c a b	b a d	d a b	e b b
3	f a d	b e e	d a d	b a d
4	f e e	b e d	f a d	d a d
5	b a d	c a b	e b b	f a d
6	d a d	d a b	a c e	a d d
7	b e e	d a d	a d d	b e d
8	f e d	e b b	b e e	f e d
9	b e d	f a d	b e d	a c e
10	e b b	f e e	f e e	b e e
11	a c e	f e d	f e d	f e e

not sorted!

Most-significant-digit-first string sort

MSD string (radix) sort.

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



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

```
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];

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

}
```

fixed-length w strings

recycles aux[] array but not count[] array

key-indexed counting

sort R subarrays recursively

Variable-length strings

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				
7	s	u	r	e	l	y	-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. Have extra char '\0' at end $\Rightarrow$ no extra work needed.

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!

compareTo() based sorts can also be sublinear!

Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} n^2$	$\frac{1}{4} n^2$	1	✔	`compareTo()`
mergesort	$n \lg n$	$n \lg n$	n	✔	`compareTo()`
quicksort	$1.39\, n \lg n$ *	$1.39\, n \lg n$	$c \lg n$ *		`compareTo()`
heapsort	$2\, n \lg n$	$2\, n \lg n$	1		`compareTo()`
LSD sort †	$2\, w\, n$	$2\, w\, n$	$n + R$	✔	`charAt()`
MSD sort ‡	$2\, w\, n$	$n \log_R n$	$n + D\, R$	✔	`charAt()`

D = function-call stack depth
(length of longest prefix match)

* 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.

https://algs4.cs.princeton.edu

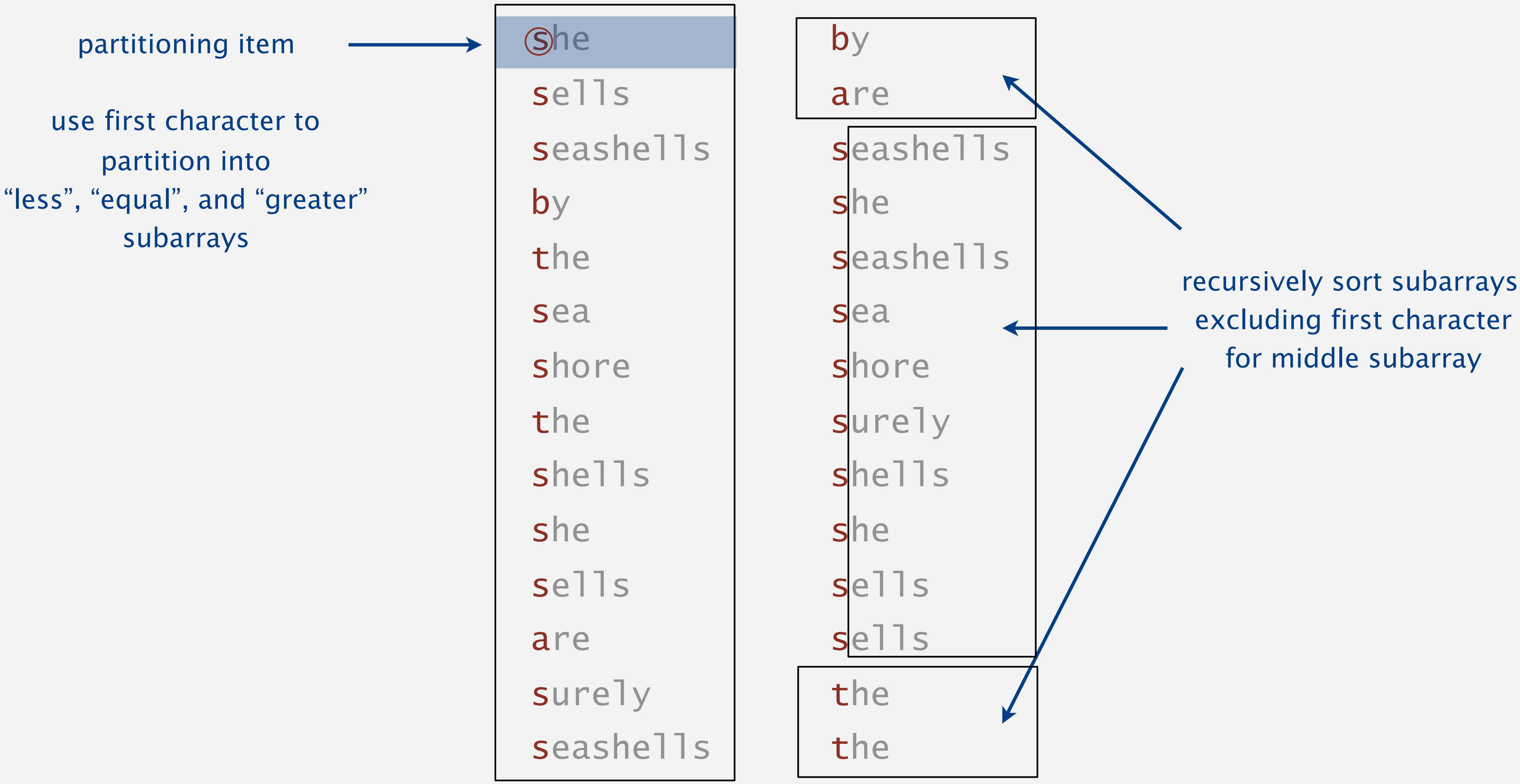
5.1 String Sorts

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

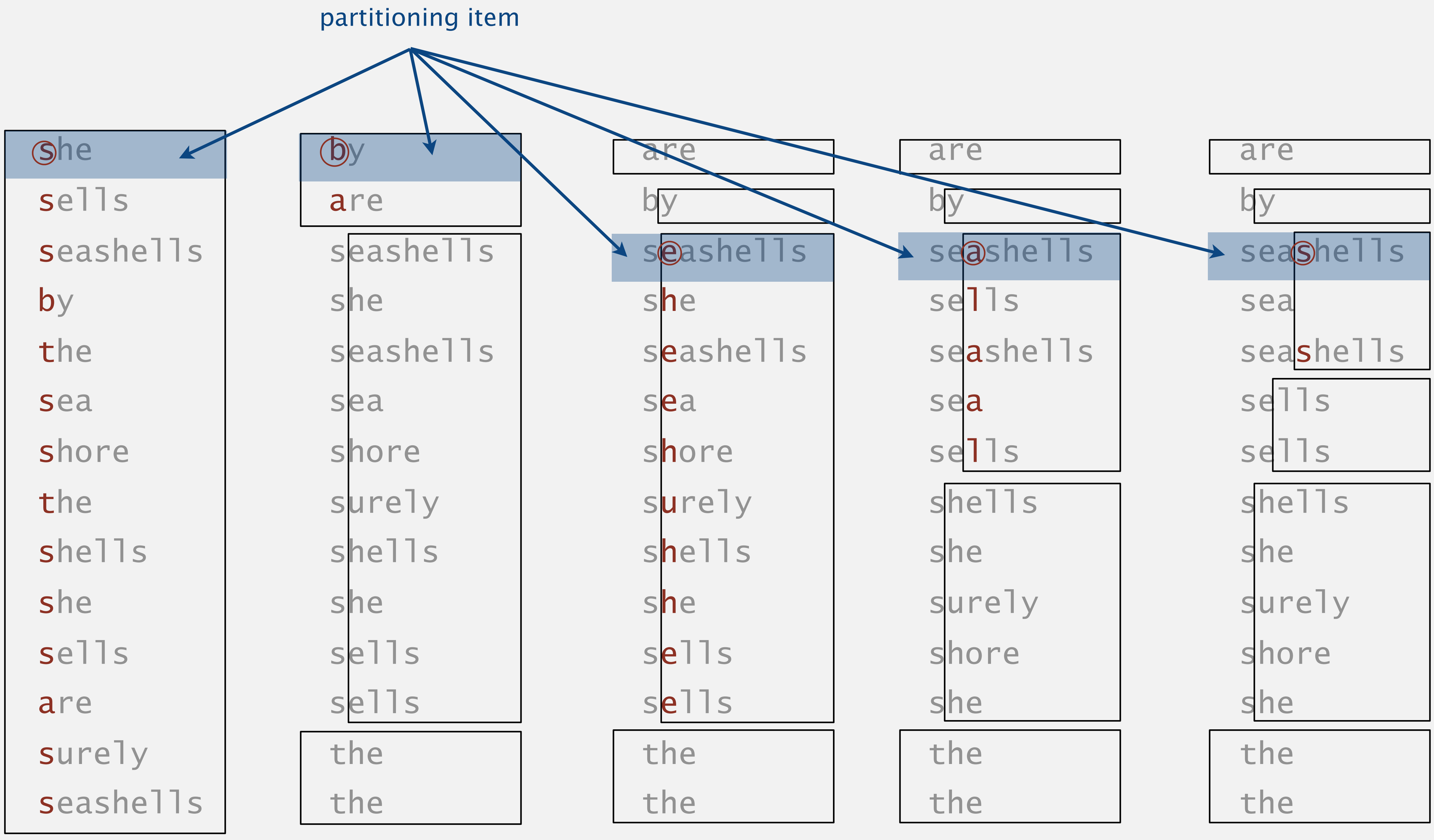
3-way string quicksort

Overview. Do 3-way partitioning on the d^{th} character.

- Less overhead than R-way partitioning in MSD radix sort.
- Does not re-examine characters equal to the partitioning char.
  (but does re-examine characters not equal to the partitioning char)



3-way string quicksort: trace of recursive calls



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 v = charAt(a[lo], d);   ←  to support variable-length strings

   int lt = lo, gt = hi;                          3-way partitioning
   int i = lo + 1;                                (using dth character)
   while (i <= gt)
   {
      int c = charAt(a[i], d);
      if      (c < v) exch(a, lt++, i++);
      else if (c > v) exch(a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);                          sort 3 subarrays recursively
   if (v != -1) sort(a, lt, gt, d+1);
   sort(a, gt+1, hi, d);

}
```

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses $\sim 2\,n \ln n$ string compares on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.

- Uses $\sim 2\,n \ln n$ character compares on average for random strings.
- Avoids re-comparing long common prefixes.

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 search trees; the first program is a sorting algorithm 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.

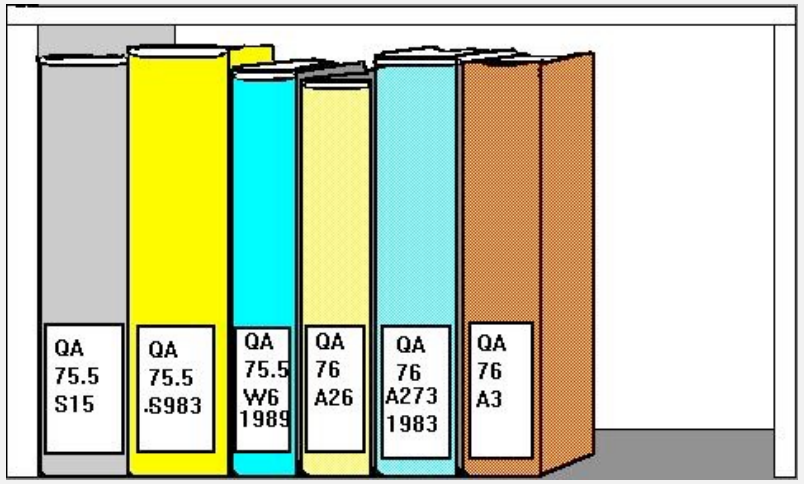
3-way string quicksort vs. MSD string sort

MSD string sort.

- Is cache-inefficient.
- Too much memory storing `count[]`.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

- Is in-place.
- Is cache-friendly.
- Has a short inner loop.
- But not stable.



library of Congress call numbers

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

Summary of the performance of sorting algorithms

Frequency of operations.

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

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

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 context).

```
% 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
    ⋮
```

Keyword-in-context search

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

```
% 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

better thing
t is a far far better thing that i do than
 some sense of better things else forgotte
was capable of better things mr carton ent
```

← number of characters of surrounding context

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	s e a l e d _ m y _ l e t t e r _ a n d _ …
713727	s e a m s t r e s s _ i s _ l i f t e d _ …
660598	s e a m s t r e s s _ o f _ t w e n t y _ …
67610	s e a m s t r e s s _ w h o _ w a s _ w i …
→ 4430	s e a r c h _ f o r _ c o n t r a b a n d …
42705	s e a r c h _ f o r _ y o u r _ f a t h e …
499797	s e a r c h _ o f _ h e r _ h u s b a n d …
182045	s e a r c h _ o f _ i m p o v e r i s h e …
143399	s e a r c h _ o f _ o t h e r _ c a r r i …
411801	s e a r c h _ t h e _ s t r a w _ h o l d …
158410	s e a r e d _ m a r k i n g _ a b o u t _ …
691536	s e a s _ a n d _ m a d a m e _ d e f a r …
536569	s e a s e _ a _ t e r r i b l e _ p a s s …
484763	s e a s e _ t h a t _ h a d _ b r o u g h …
	⋮

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*

Radix sorting: quiz 3

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. 1

B. n

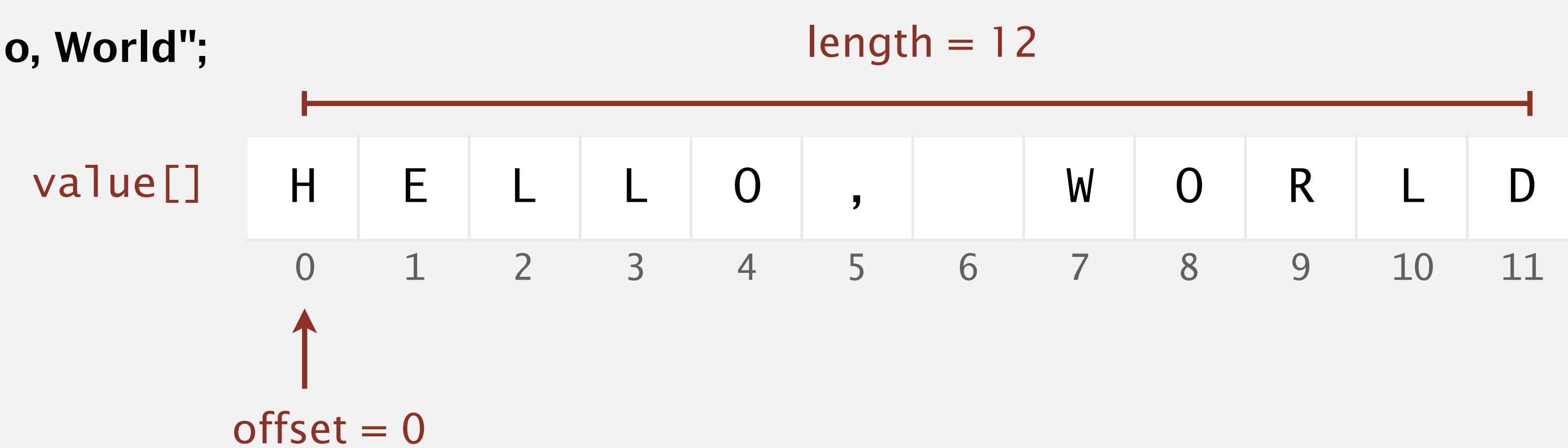
C. $n \log n$

D. n^2

The String data type: Java 7u5 implementation

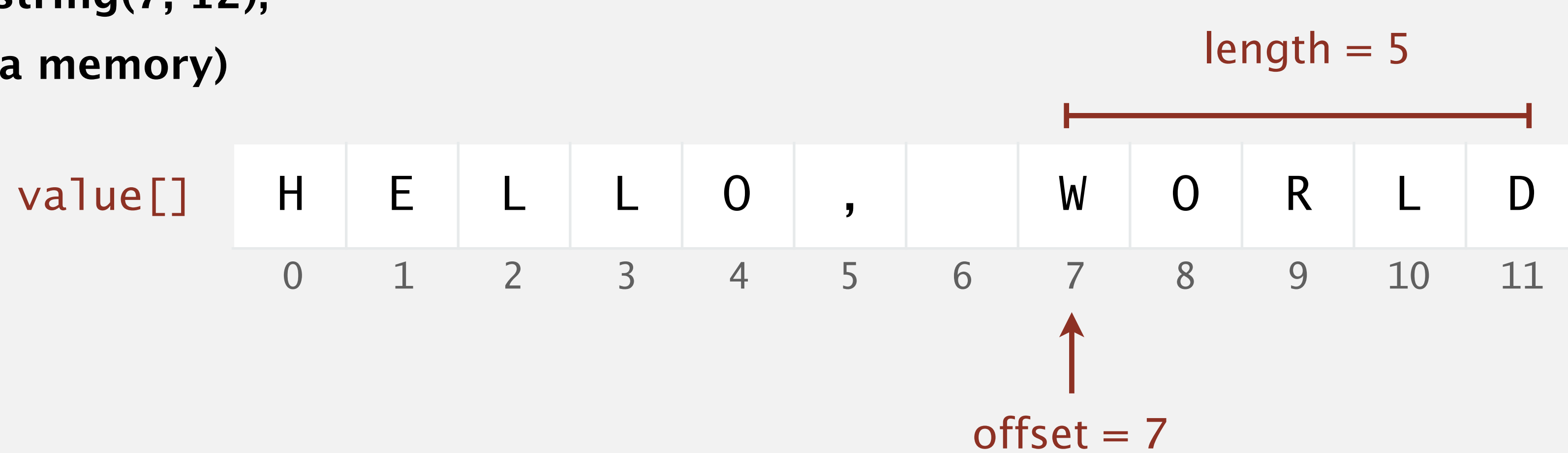
```
public final class String implements Comparable<String>
{
   private char[] value;   // characters
   private int offset;     // index of first char in array
   private int length;     // length of string
   private int hash;       // cache of hashCode()
   …
```

String s = "Hello, World";



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

(constant extra memory)



The String data type: Java 7u6 implementation

```
public final class String implements Comparable<String>
{
   private char[] value;  // characters
   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);

(linear extra memory)

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

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 ala 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 ala 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] No linear-time algorithm.

Proposition. [Weiner 1973] Linear-time algorithms (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	$6\ n$

Manber-Myers (1991): see lecture videos

Larsson-Sadakane (1999): about 10× faster than Manber-Myers

divsufsort2 (2008) and sais (2010): 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 is 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.