http://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*

http://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.

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

" 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.
- Can represent at most 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

Hexadecimal to ASCII conversion table

A	á	∂	𝔊
U+0041	U+00E1	U+2202	U+1D50A

some Unicode characters

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

I ♥ Unicode

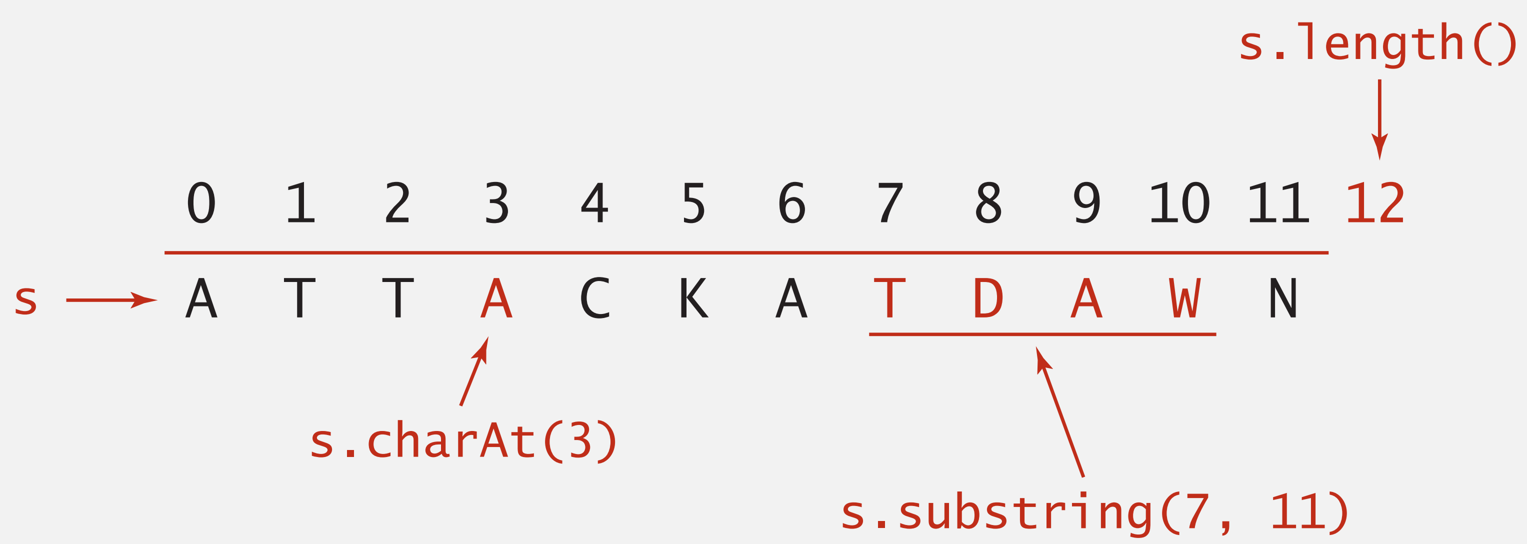
The String data type

String data type in Java. Immutable sequence of characters.

Length. Number of characters.

Indexing. Get the i^{th} character.

Concatenation. Concatenate one string to the end of another.



The String data type: immutability

Q. Why immutable?

A. All the usual reasons.

- Can use as keys in symbol table.
- Don't need to defensively copy.
- Ensures consistent state.
- Supports concurrency.
- Improves security.

```
public class FileInputStream
{
   private String filename;
   public FileInputStream(String filename)
   {
      if (!allowedToReadFile(filename))
         throw new SecurityException();
      this.filename = filename;
   }
   ...
}
```

attacker could bypass security if string type were mutable

The String data type: representation

Representation (Java 7). Immutable `char[]` array + cache of hash.

operation	Java	running time
length	`s.length()`	1
indexing	`s.charAt(i)`	1
concatenation	`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 rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
      rev += s.charAt(i);
   return rev;
}
```

quadratic time

A. Use `StringBuilder` data type (mutable `char[]` array).

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

linear time

Comparing two strings

Q. How many character compares to compare two strings of length W ?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x	e	s

Running time. Proportional to length of longest common prefix.

- Proportional to W in the worst case.
- But, often sublinear in W.

Alphabets

Digital key. Sequence of digits over fixed 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*

http://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	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	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()`

* probabilistic

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

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on key compares.

use array accesses
to make R-way decisions
(instead of binary decisions)

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data $\Rightarrow$ can't just count up number of 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];
```

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 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

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✔

a[0]	Anderson	2	Harris	1	aux[0]
a[1]	Brown	3	Martin	1	aux[1]
a[2]	Davis	3	Moore	1	aux[2]
a[3]	Garcia	4	Anderson	2	aux[3]
a[4]	Harris	1	Martinez	2	aux[4]
a[5]	Jackson	3	Miller	2	aux[5]
a[6]	Johnson	4	Robinson	2	aux[6]
a[7]	Jones	3	White	2	aux[7]
a[8]	Martin	1	Brown	3	aux[8]
a[9]	Martinez	2	Davis	3	aux[9]
a[10]	Miller	2	Jackson	3	aux[10]
a[11]	Moore	1	Jones	3	aux[11]
a[12]	Robinson	2	Taylor	3	aux[12]
a[13]	Smith	4	Williams	3	aux[13]
a[14]	Taylor	3	Garcia	4	aux[14]
a[15]	Thomas	4	Johnson	4	aux[15]
a[16]	Thompson	4	Smith	4	aux[16]
a[17]	White	2	Thomas	4	aux[17]
a[18]	Williams	3	Thompson	4	aux[18]
a[19]	Wilson	4	Wilson	4	aux[19]

http://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 string sort

LSD string (radix) sort.

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

sort key (d = 2)

sort key (d = 1)

sort key (d = 0)

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

sort is stable
(arrows do not cross)

LSD string sort: correctness proof

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

Pf. [by induction on i]

After pass i, strings are sorted by last i characters.

- If two strings differ on sort key, key-indexed sort puts them in proper relative order.
- If two strings agree on sort key, stability keeps them in proper relative order.

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

sorted from previous passes (by induction)

sort key

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: 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 †	$2\, W (N + R)$	$2\, W (N + R)$	$N + R$	✔	`charAt()`

* probabilistic
† fixed-length W keys

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

String sorting interview question

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview.

Which sorting method to use?

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


Google

How to take a census in 1900s?

1880 Census. Took 1500 people 7 years to manually process data.

Herman Hollerith. Developed counting and sorting machine to automate.

- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time (into one of 12 bins).
- Typical question: how many women of age 20 to 30?

Hollerith tabulating machine and sorter

punch card (12 holes per column)

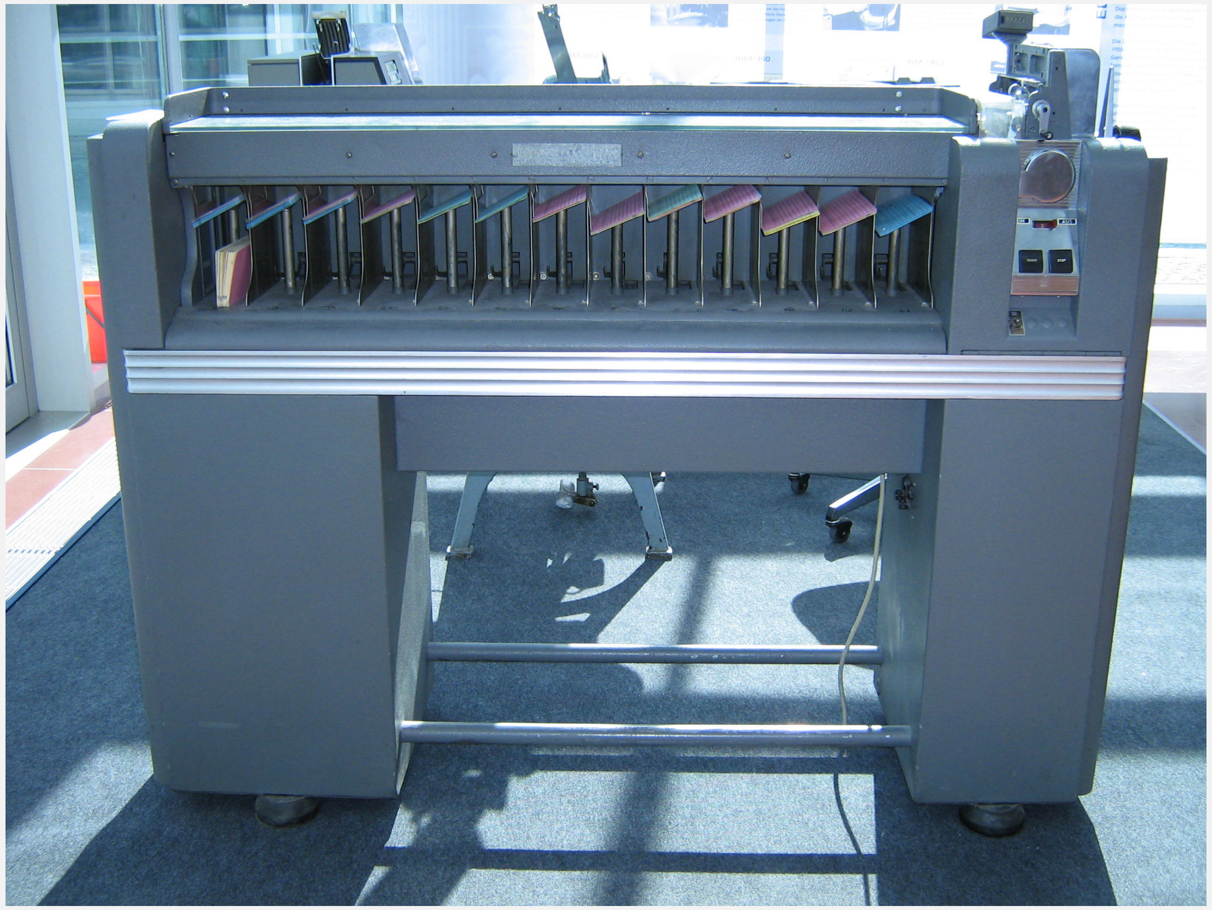
1890 Census. Finished in 1 year (and under budget)!

How to get rich sorting in 1900s?

Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

Hollerith's company later merged with 3 others to form Computing Tabulating Recording Corporation (CTRC); company renamed in 1924.



IBM 80 Series Card Sorter (650 cards per minute)

IBM®

LSD string sort: a moment in history (1960s)

card punch

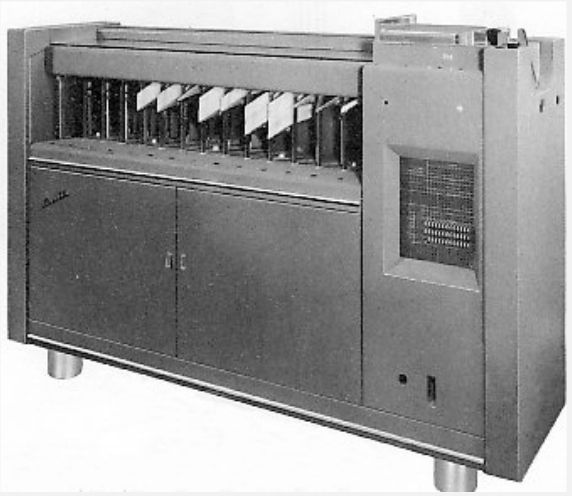
punched cards

card reader

mainframe

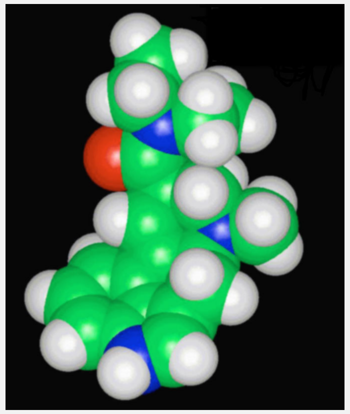
line printer

To sort a card deck
- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

not directly related
to sorting



Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

http://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).

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)

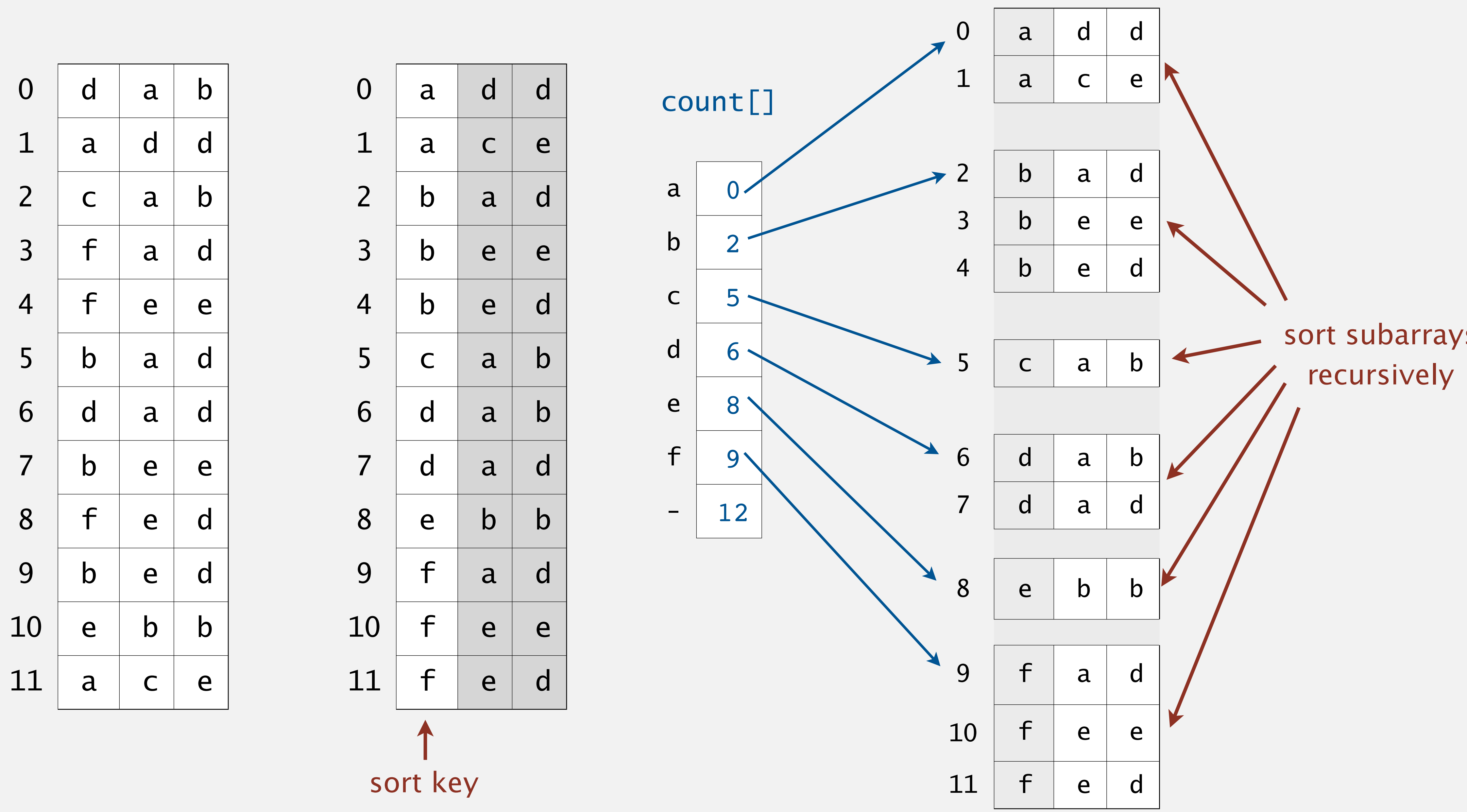
0	c	a	b
1	d	a	b
2	e	b	b
3	b	a	d
4	d	a	d
5	f	a	d
6	a	d	d
7	b	e	d
8	f	e	d
9	a	c	e
10	b	e	e
11	f	e	e

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: example

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

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: Java implementation

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

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int[] count = new int[R+2];
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

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

recycles aux[] array
but not count[] array

key-indexed counting

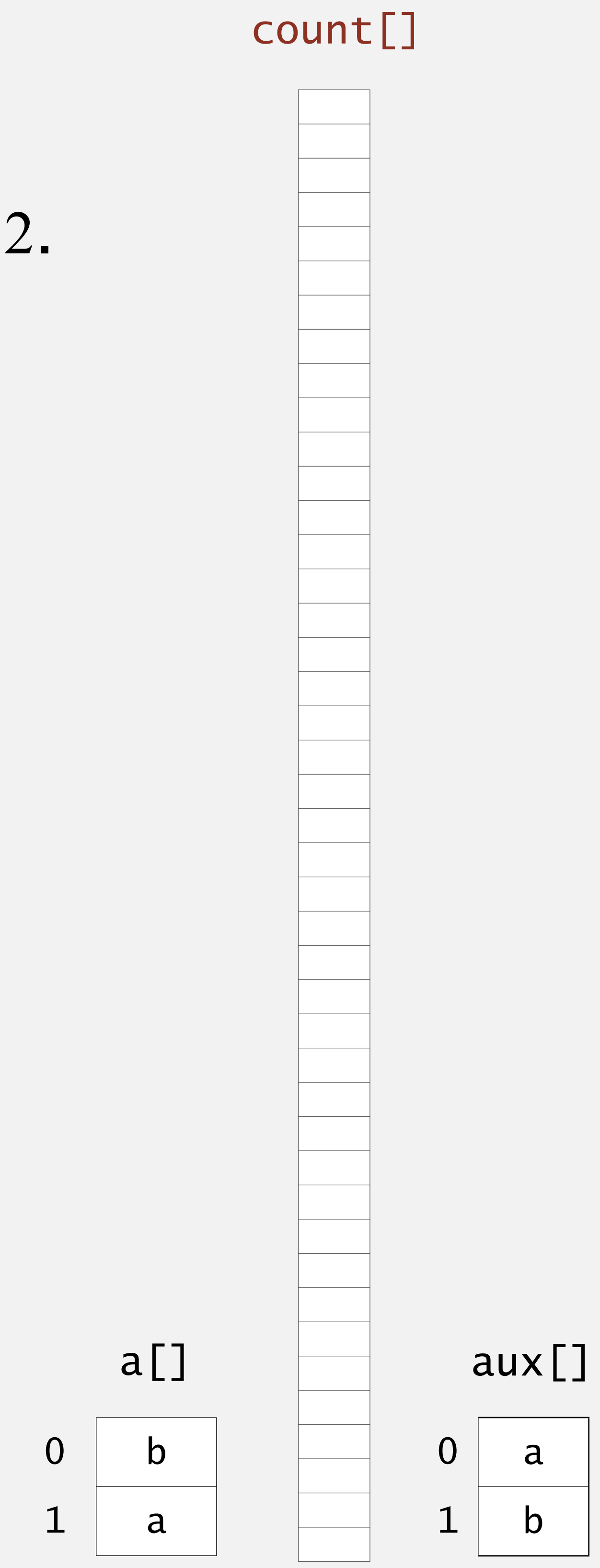
sort R subarrays recursively

MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

Observation 2. Huge number of small subarrays because of recursion.



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.

```
private static void sort(String[] a, int lo, int hi, int d)
{
   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}
```

- Implement `less()` so that it compares starting at d^{th} character.

```
private static boolean less(String v, String w, int d)
{
   for (int i = d; i < Math.min(v.length(), w.length()); i++)
   {
      if (v.charAt(i) < w.charAt(i)) return true;
      if (v.charAt(i) > w.charAt(i)) return false;
   }
   return v.length() < w.length();
}
```

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

Characters examined by MSD string sort

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 †	$2\, W (N + R)$	$2\, W (N + R)$	$N + R$	✔	`charAt()`
MSD sort ‡	$2\, W (N + R)$	$N \log_R N$	$N + D\, R$	✔	`charAt()`

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

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

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Extra space for `aux[]`.
- Extra space for `count[]`.
- Inner loop has a lot of instructions.
- Accesses memory "randomly" (cache inefficient).

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

doesn't rescan characters

tight inner loop, cache friendly

Goal. Combine advantages of MSD and quicksort.

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

Optimization 1. Replace recursion with explicit stack.

- Push subarrays to be sorted onto stack.
- Now, one `count[]` array 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.

http://algs4.cs.princeton.edu

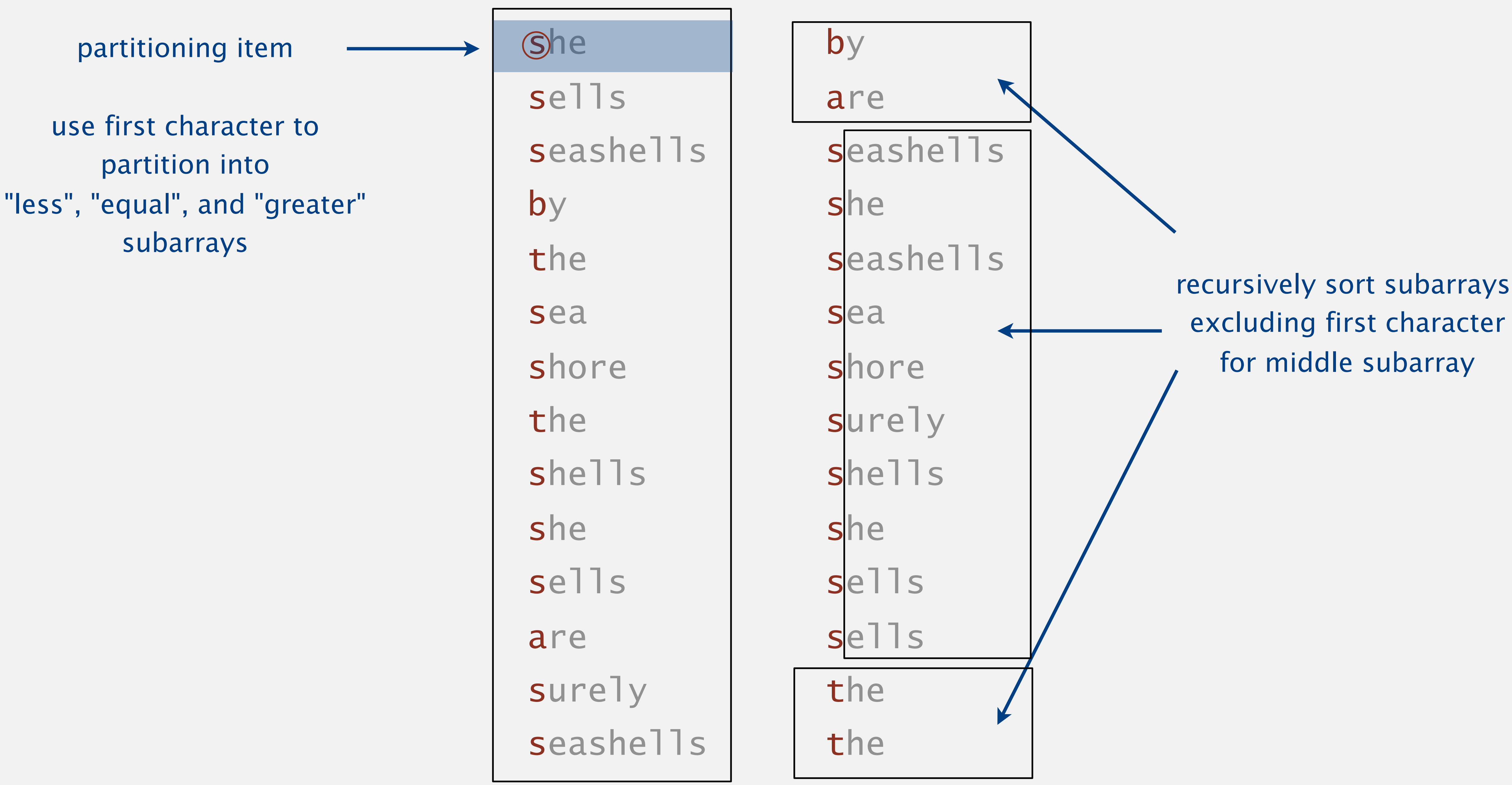
5.1 String Sorts

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

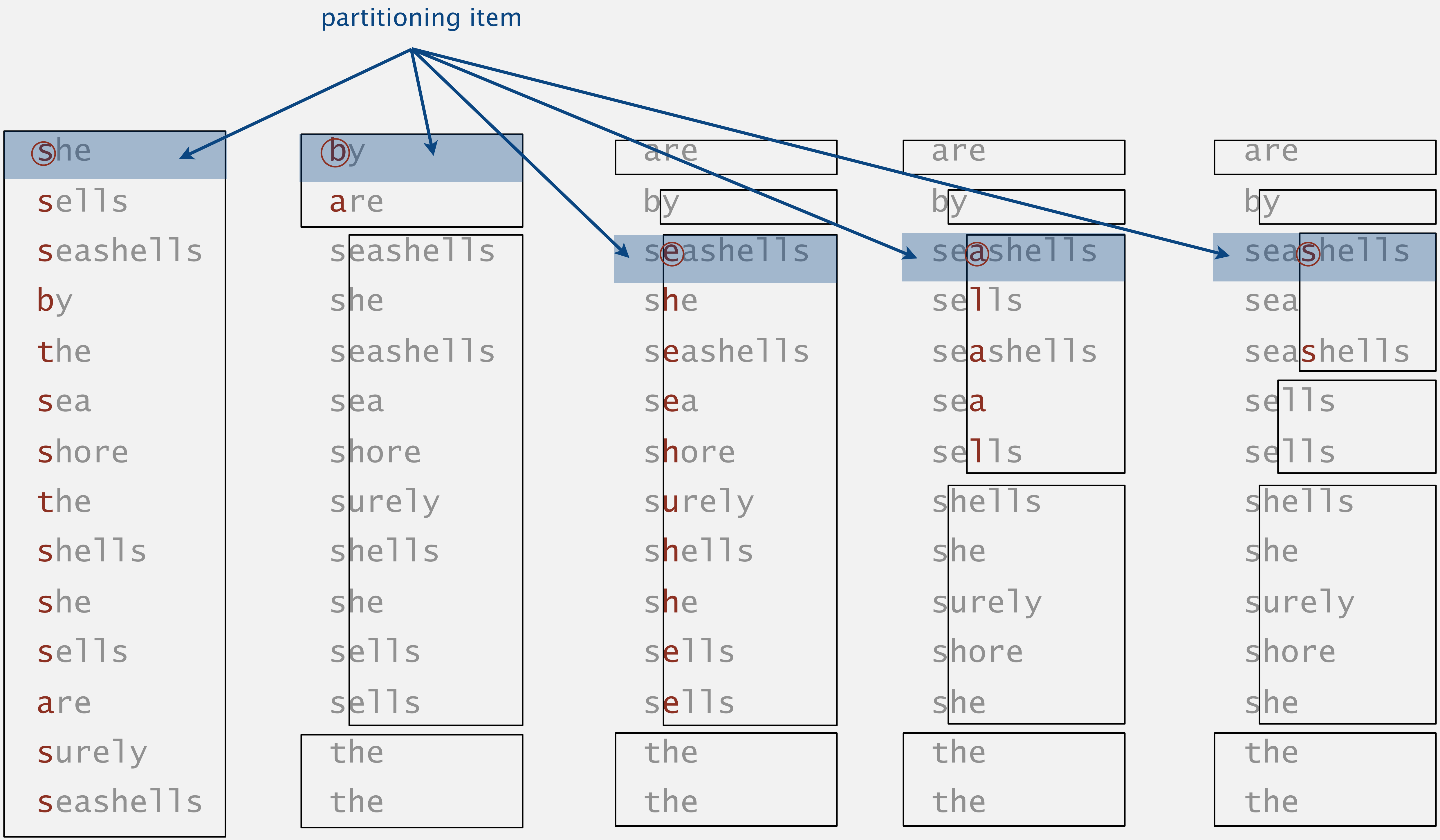
3-way string quicksort (Bentley and Sedgewick, 1997)

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

- Less overhead than R-way partitioning in MSD string 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 size 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 lt = lo, gt = hi;
   int v = charAt(a[lo], d);
   int i = lo + 1;
   while (i <= gt)
   {
      int t = charAt(a[i], d);
      if      (t < v) exch(a, lt++, i++);
      else if (t > v) exch(a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);
   if (v >= 0) sort(a, lt, gt, d+1);
   sort(a, gt+1, hi, d);
}
```

3-way partitioning (using d^{th} character)

to handle variable-length strings

sort 3 subarrays recursively

3-way string quicksort vs. standard quicksort

Standard quicksort.

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

3-way string (radix) quicksort.

- Uses $\sim 2N \ln N$ 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 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.

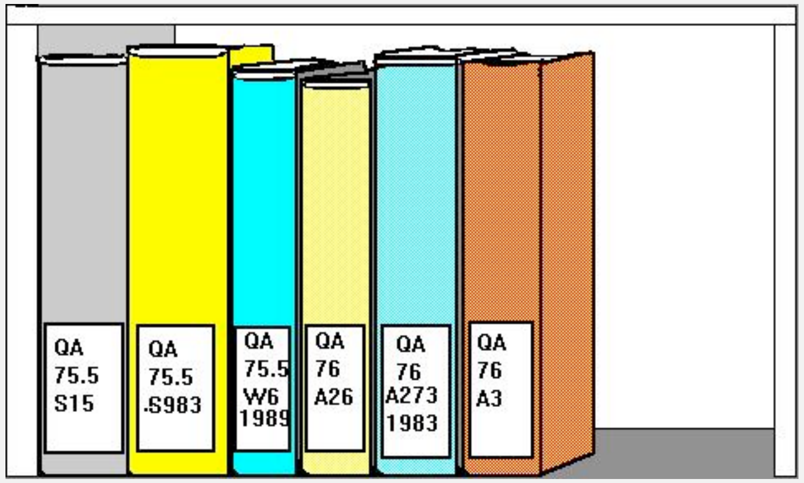
3-way string quicksort vs. MSD string sort

MSD string sort.

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

3-way string quicksort.

- Is cache-friendly.
- Is in-place.
- Has a short inner loop.



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 †	$2\, W (N + R)$	$2\, W (N + R)$	$N + R$	✔	`charAt()`
MSD sort ‡	$2\, W (N + R)$	$N \log_R N$	$N + D\, R$	✔	`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

http://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
      ⋮
```

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

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
```

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
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…
	⋮

War story

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

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

Arrays.sort(suffixes);
```

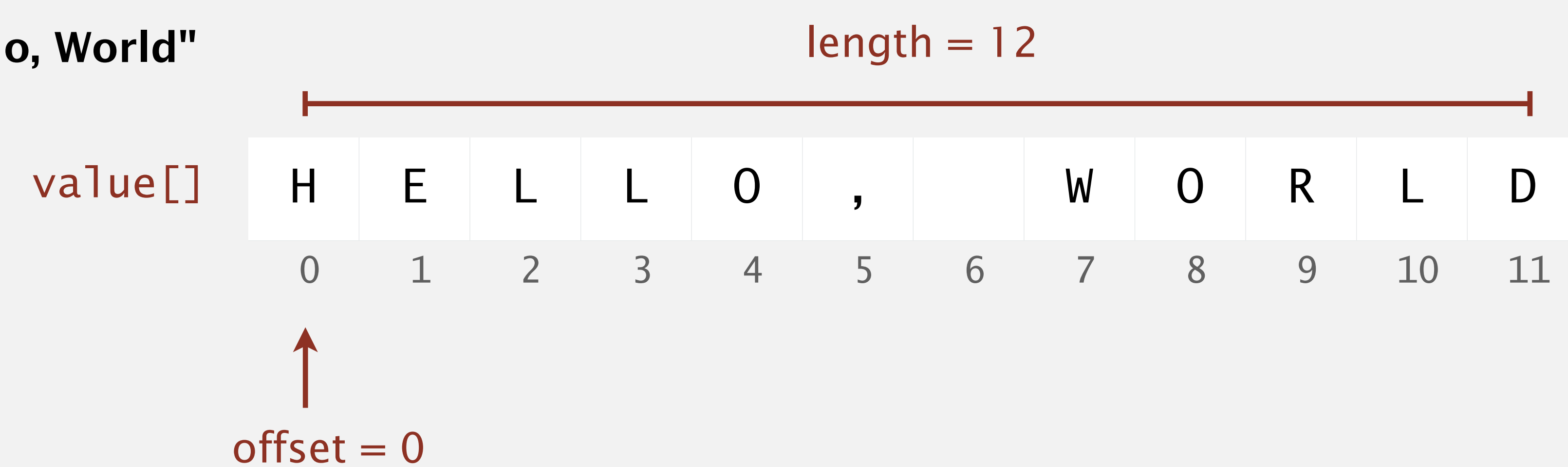
3rd printing

input file	characters	Java 7u4	Java 7u5
amendments.txt	18 thousand	0.25 sec	2.0 sec
aesop.txt	192 thousand	1.0 sec	*out of memory*
mobydick.txt	1.2 million	7.6 sec	*out of memory*
chromosome11.txt	7.1 million	61 sec	*out of memory*

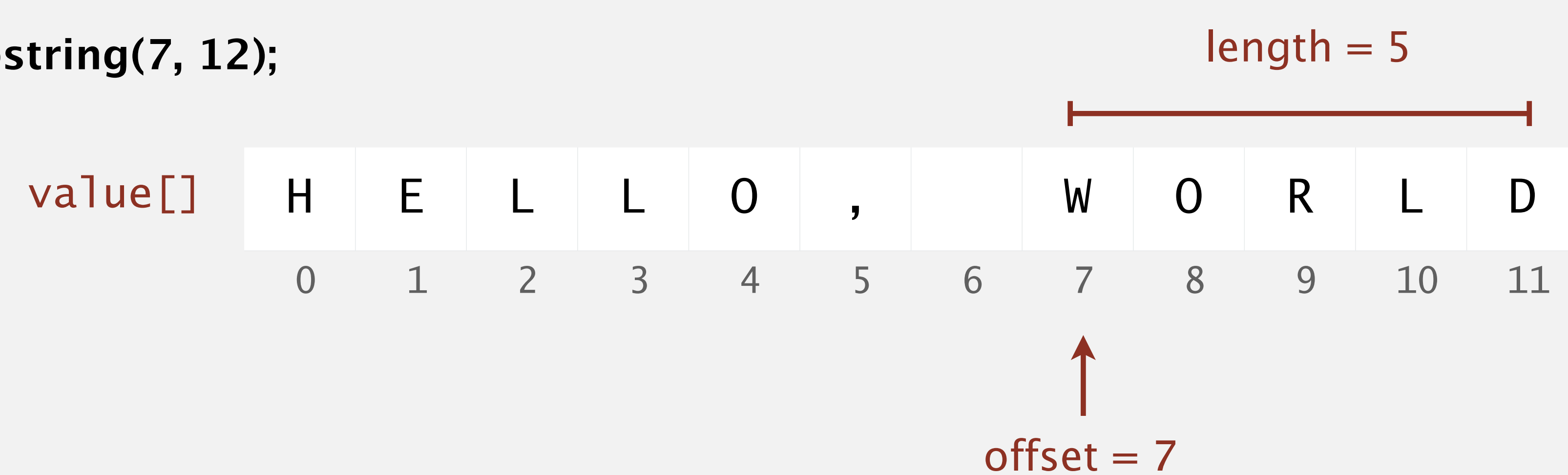
The String data type: Java 7u5 implementation

```
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);



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);

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
substring extraction	1	N
concatenation	$M + N$	$M + N$
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`.

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

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

Arrays.sort(suffixes);
```

4th printing

Lessons learned

Lesson 1. Put performance guarantees in API.

Lesson 2. If API has no performance guarantees, don't rely upon any!

Corollary. May want to avoid `String` data type for huge strings.

- Are you sure `charAt()` and `length()` take constant time?
- If lots of calls to `charAt()`, overhead for function calls is large.
- If lots of small strings, memory overhead of `String` is large.

Ex. Our optimized algorithm for suffix arrays is 5x faster and uses 32x less memory than our original solution in Java 7u5!

Suffix Arrays: theory

Q. What is worst-case running time of our suffix arrays algorithm?

- Quadratic.
- Linearithmic.
- Linear.
- None of the above. ← $N^2 \log N$

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

Suffix Arrays: theory

Q. What is complexity of suffix arrays?

- Quadratic.
- Linearithmic. ← Manber-Myers algorithm (see video)
- ✓ Linear. ← suffix trees (beyond our scope)
- Nobody knows.

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.

Suffix arrays:
A new method for on-line string searches

Udi Manber[1]
Gene Myers[2]

Department of Computer Science
University of Arizona
Tucson, AZ 85721

May 1989
Revised August 1991

Abstract

A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type, "Is W a substring of A?" to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst case, versus $O(N \log N)$ time for suffix arrays. However, we give an augmented algorithm that, regardless of the alphabet size, constructs suffix arrays in $O(N)$ ***expected*** *time, albeit with lesser space efficiency. We believe that suffix arrays will prove to be better in practice than suffix trees for many applications.*

Suffix Arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Many ingenious algorithms.

- Memory footprint very important.
- State-of-the art still changing.

year	algorithm	worst case	memory
1990	**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	$6\ N$

good choices
(Yuta Mori)

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 amount of data in keys (not number of keys).
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.

- $1.39 \, N \lg N$ chars for random data.

Long strings are rarely random in practice.

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