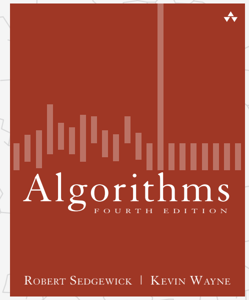
Unrelated things

Observations from the weekend:

- Bitcoins are a really good plot point in the cyberunk dystopian future we live in.
  - Ditto Snapchat.
- That Lorde song is pretty good.
  - But autotune is another harbringer of dystopia.
- Lots of people make their first submissions even with an extension on the day an assignment is due.
- Kerbal space program is amazing!
- Writing database software for four hours that saves you 30 minutes is a perfectly fine tradeoff.
- People around trash fires want to know about mergesort.



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).
- Disk drives (useful in forensics, see COS 432).
- …

" 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

Unicode characters

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

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

I (heart) Unicode

The String data type

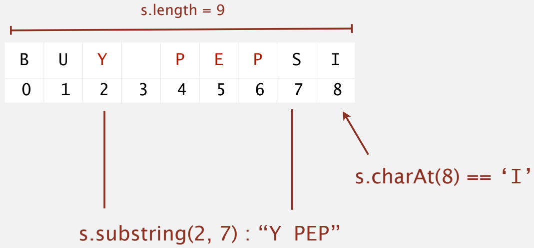
String data type in Java. Sequence of characters (immutable).

Length. Number of characters.

Indexing. Get the i^{th} character.

Substring extraction. Get a contiguous subsequence of characters.

String concatenation. Append characters to end of string.



The String data type: Java implementation

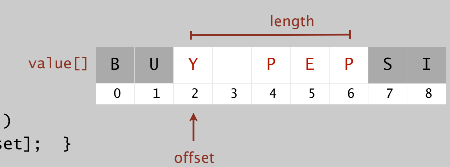
```
public final class String implements Comparable<String>
{
   private char[] value;  // characters
   private int offset;    // index of first char in array
   private int length;    // length of string
   private int hash;      // cache of hashCode()

   public int length()
   {  return length; }

   public char charAt(int i)
   {  return value[i + offset];  }


   private String(int offset, int length, char[] value)
   {
      this.offset = offset;
      this.length = length;
      this.value  = value;
   }

   public String substring(int from, int to)
   {  return new String(offset + from, to - from, value);  }
   …
```



Java 6: copy of reference to original char array

Java 7: new copy of value is made

The String data type: performance

String data type (in Java). Sequence of characters (immutable).
Underlying implementation. Immutable `char[]` array, offset, and length.

	String	
operation	guarantee	extra space
`length()`	1	1
`charAt()`	1	1
`substring()`	1 or N	1 or N
`concat()`, `+`	N	N

Runtimes for Java 6 and 7, respectively

"goldentoa".concat("d")
"goldentoa" + "d" → "goldentoad"

Memory. $40 + 2N$ bytes for a freshly created `String` of length N.

can use `byte[]` or `char[]` instead of `String` to save space
(but lose convenience of `String` data type)

Java 7, Update #6

```
"Shared char array backing buffers only 'win' with
very heavy use of String.substring. The negatively
impacted situations can include parsers and
compilers however current testing shows that
overall this change is beneficial."
```

	Java 7 String	
operation	guarantee	extra space
`substring()`	N	N

Tradeoffs.
- Bad: Slower substring construction. Breaks old code.
- "Good": Lazy programmer need not create new Strings to save space.
  - String smallString = new String(s.substring(a, b));

Moral of the story.
- No more easy substring construction.
- Alternate approaches are more complex (See Burrows-Wheeler assignment on Coursera)

The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).
Underlying implementation. Resizing `char[]` array and length.

	String		StringBuilder	
operation	guarantee	extra space	guarantee	extra space
`length()`	1	1	1	1
`charAt()`	1	1	1	1
`substring()`	1 or N	1 or N	N	N
S: `concat()`, `+` SB: `append()`	N	N	1 *	1 *

* amortized

sb.append("d")

Remark. `StringBuffer` data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q: Which string reversal method is more efficient?

A.

```
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
extremely common
rookie mistake!

B.

```
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

Sublinearity example: Longest common prefix

Q. How many compares to compute length of longest common prefix?

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

```
public static int lcp(String s, String t)
{
   int N = Math.min(s.length(), t.length());
   for (int i = 0; i < N; i++)
      if (s.charAt(i) != t.charAt(i))
         return i;
   return N;
}
```

linear time (worst case)
sublinear time (typical case)

Running time. Proportional to length D of longest common prefix.
Remark. Also can compute `compareTo()` in sublinear time.

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 = key compares.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`

* probabilistic

Lower bound. ~ $N \lg N$ 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.

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.

#			
5	Sandra	Vanilla	Grimes
0	Lauren	Mint	Jon Talabot
11	Lisa	Vanilla	Blue Peter
9	Dave	Chocolate	Superpope
4	JS	Fish	The Filthy Reds
7	James	Rocky Road	Robots are Supreme
3	Edith	Vanilla	My Bloody Valentine
6	Swimp	Chocolate	Sef
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
8	Lee	Vanilla	La(r)va
10	Bearman	Butter Pecan	Extrobophile

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.
  - Create new array.
  - Copy entry with key i into ith row.

#			
5	Sandra	Vanilla	Grimes
0	Lauren	Mint	Jon Talabot
11	Lisa	Vanilla	Blue Peter
9	Dave	Chocolate	Superpope
4	JS	Fish	The Filthy Reds
7	James	Rocky Road	Robots are Supreme
3	Edith	Vanilla	My Bloody Valentine
6	Swimp	Chocolate	Sef
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
8	Lee	Vanilla	La(r)va
10	Bearman	Butter Pecan	Extrobophile

#		

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.
  - Create new array.
  - Copy entry with key i into ith row.

#			
5	Sandra	Vanilla	Grimes
0	Lauren	Mint	Jon Talabot
11	Lisa	Vanilla	Blue Peter
9	Dave	Chocolate	Superpope
4	JS	Fish	The Filthy Reds
7	James	Rocky Road	Robots are Supreme
3	Edith	Vanilla	My Bloody Valentine
6	Swimp	Chocolate	Sef
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
8	Lee	Vanilla	La(r)va
10	Bearman	Butter Pecan	Extrobophile

#		
5	Sandra	...

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.
  - Create new array.
  - Copy entry with key i into ith row.

#			
5	Sandra	Vanilla	Grimes
0	Lauren	Mint	Jon Talabot
11	Lisa	Vanilla	Blue Peter
9	Dave	Chocolate	Superpope
4	JS	Fish	The Filthy Reds
7	James	Rocky Road	Robots are Supreme
3	Edith	Vanilla	My Bloody Valentine
6	Swimp	Chocolate	Sef
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
8	Lee	Vanilla	La(r)va
10	Bearman	Butter Pecan	Extrobophile

#		
0	Lauren	...
5	Sandra	...

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.
  - Create new array.
  - Copy entry with key i into ith row.

#			
5	Sandra	Vanilla	Grimes
0	Lauren	Mint	Jon Talabot
11	Lisa	Vanilla	Blue Peter
9	Dave	Chocolate	Superpope
4	JS	Fish	The Filthy Reds
7	James	Rocky Road	Robots are Supreme
3	Edith	Vanilla	My Bloody Valentine
6	Swimp	Chocolate	Sef
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
8	Lee	Vanilla	La(r)va
10	Bearman	Butter Pecan	Extrobophile

#		
0	Lauren	...
5	Sandra	...
11	Lisa	...

Sublinearithmic Sort

Simplest Case.

- Keys are unique integers from 0 to 11.
  - Create new array.
  - Copy entry with key i into ith row.
  - Throw away old table.

N rows

#			
0	Lauren	Mint	Jon Talabot
1	Delbert	Strawberry	Ronald Jenkees
2	Glaser	Cardamom	Rx Nightly
3	Edith	Vanilla	My Bloody Valentine
4	JS	Fish	The Filthy Reds
5	Sandra	Vanilla	Grimes
6	Swimp	Chocolate	Sef
7	James	Rocky Road	Robots are Supreme
8	Lee	Vanilla	La(r)va
9	Dave	Chocolate	Superpope
10	Bearman	Butter Pecan	Extrobophile
11	Lisa	Vanilla	Blue Peter

- Order of growth of running time: N

Sublinearithmic Sorts

Simplest Case.

- Keys are unique integers from 0 to N-1.

More Complex Cases.

- Non-unique keys.
- Non-consecutive keys.
- Non-numerical keys.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

Sublinearithmic Sorts

Alphabet Case.

- Keys belong to a finite ordered alphabet.
  - Example: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

pollEv.com/jhug text to **37607**

Q: What will be the index of the first ♥?

`Text 686853 followed by #####.`

`Example: “686853 11” would mean first ♥ will be at index 11.`

Sublinearithmic Sorts

Alphabet Case.

- Keys belong to a finite ordered alphabet.
  - Example: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

0	♣	
1	♣	
2	♣	
3	♠	
4	♠	
5		
6		
7		
8		
9		
10		
11		

pollEv.com/jhug text to **37607**

Q: What will be the index of the first ♥?

There are 3 ♣s and 2 ♠s. These will take up the slots 0 through 4, so the first ♥ goes in 5.

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→

Starting Points

0	♣	0
1	♠	3
2	♥	5
3	♦	9

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→

Starting Points

0	♣	0
1	♠	4
2	♥	5
3	♦	9

→

0		
1		
2		
3	♠	Lauren
4		
5		
6		
7		
8		
9		
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→

Starting Points

0	♣	0
1	♠	4
2	♥	6
3	♦	9

→

0		
1		
2		
3	♠	Lauren
4		
5	♥	Delbert
6		
7		
8		
9		
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	0
1	♠	4
2	♥	6
3	♦	10

Starting Points

→

0		
1		
2		
3	♠	Lauren
4		
5	♥	Delbert
6		
7		
8		
9	♦	Glaser
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	1
1	♠	4
2	♥	6
3	♦	10

Starting Points

→

0	♣	Edith
1		
2		
3	♠	Lauren
4		
5	♥	Delbert
6		
7		
8		
9	♦	Glaser
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	1
1	♠	5
2	♥	6
3	♦	10

Starting Points

→

0	♣	Edith
1		
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6		
7		
8		
9	♦	Glaser
10		
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	1
1	♠	5
2	♥	6
3	♦	11

Starting Points

→

0	♣	Edith
1		
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6		
7		
8		
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→ Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→ Starting Points

0	♣	1
1	♠	5
2	♥	7
3	♦	11

→

0	♣	Edith
1		
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7		
8		
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→ Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→ Starting Points

0	♣	1
1	♠	5
2	♥	8
3	♦	11

→

0	♣	Edith
1		
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7	♥	James
8		
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→ Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→ Starting Points

0	♣	2
1	♠	5
2	♥	8
3	♦	11

→

0	♣	Edith
1	♣	Lee
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7	♥	James
8		
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→ Counts

0	♣	3
1	♠	2
2	♥	4
3	♦	3

→ Starting Points

0	♣	2
1	♠	5
2	♥	9
3	♦	11

→

0	♣	Edith
1	♣	Lee
2		
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7	♥	James
8	♥	Dave
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	3
1	♠	5
2	♥	9
3	♦	11

Starting Points

→

0	♣	Edith
1	♣	Lee
2	♣	Bearman
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7	♥	James
8	♥	Dave
9	♦	Glaser
10	♦	Sandra
11		

Key-indexed counting

Example

- Alphabet: {♣, ♠, ♥, ♦}

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	3
1	♠	5
2	♥	9
3	♦	12

Starting Points

→

0	♣	Edith
1	♣	Lee
2	♣	Bearman
3	♠	Lauren
4	♠	JS
5	♥	Delbert
6	♥	Swimp
7	♥	James
8	♥	Dave
9	♦	Glaser
10	♦	Sandra
11	♦	Lisa

Memory Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	3
1	♠	2
2	♥	4
3	♦	3

Counts

→

0	♣	0
1	♠	3
2	♥	5
3	♦	9

Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	
1	♠	
2	♥	
3	♦	

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	
1	♠	3
2	♥	
3	♦	

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	
1	♠	3
2	♥	2
3	♦	4
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	
1	♠	3
2	♥	2
3	♦	4
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	0
1	♠	3
2	♥	2
3	♦	4
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	0
1	♠	3
2	♥	2
3	♦	4
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	0
1	♠	3
2	♥	5
3	♦	4
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	0
1	♠	3
2	♥	5
3	♦	9
		3

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.

♠	Lauren
♥	Delbert
♦	Glaser
♣	Edith
♠	JS
♦	Sandra
♥	Swimp
♥	James
♣	Lee
♥	Dave
♣	Bearman
♦	Lisa

→

0	♣	0
1	♠	3
2	♥	5
3	♦	9
		12

Counts And Starting Points

→

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

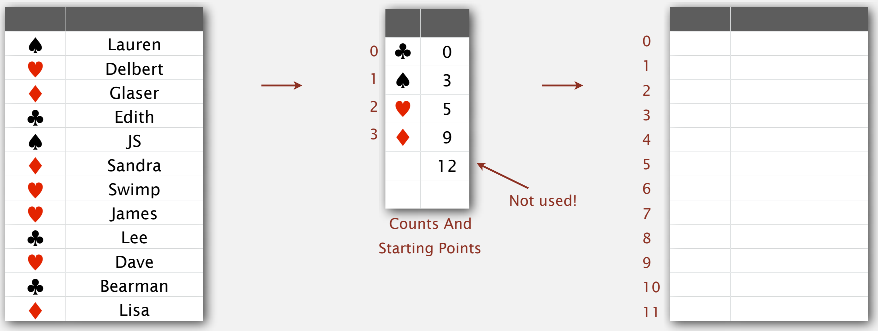
Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.



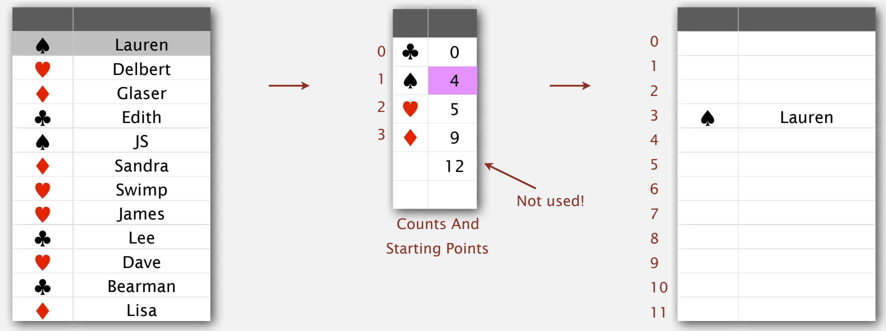
Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.



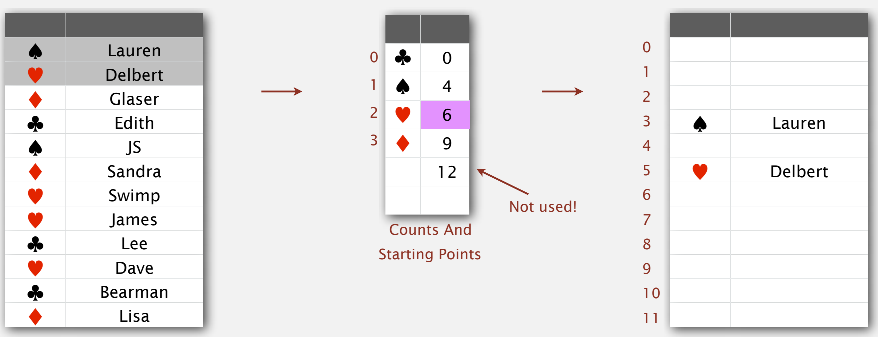
Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Optimization

Can save memory

- Replace our two helper arrays by one array that does both jobs.



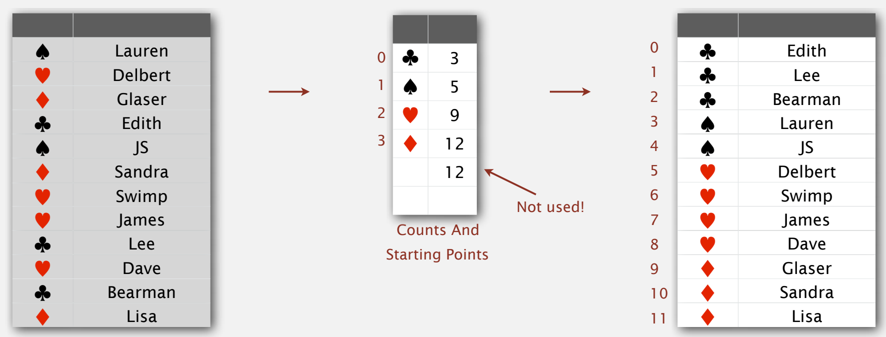
Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Key-indexed counting

Can save memory

- Replace our two helper arrays by one array that does both jobs.



Two phase construction

- Create counts as before, but offset by 1 position.
- Convert count array into a cumulant array.

Key-indexed counting: Book Implementation

Assumption. Keys are integers between 0 and $R - 1$.
Implication. Can use key as an array index.

Reminder. char datatype is really just an int in disguise.

- `System.out.println('a' == 97).`
  - Prints true

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: analysis

Proposition. Key-indexed counting uses $\sim 11\,N + 4\,R$ array accesses to sort N items whose keys are integers between 0 and $R - 1$.

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

String Keys

Alphabet Case. Can use key-indexed counting directly.

- Keys belong to a finite ordered alphabet.

String Case. ← Key insight: Can repeatedly use key-indexed counting!

- Keys are a sequence of characters from a finite ordered alphabet.

horse	Lauren
elf	Delbert
cat	Glaser
crab	Edith
monkey	JS
rhino	Sandra
raccoon	Swimp
cat	James
fish	Lee
tree	Dave
virus	Bearman
human	Lisa

letters

♠♠	Lauren
♥♦	Delbert
♦♣	Glaser
♣♥	Edith
♠♥	JS
♦♣	Sandra
♥♠	Swimp
♥♦	James
♣♠	Lee
♥♣	Dave
♣♠	Bearman
♦♠	Lisa

suits

42387	Lauren
34163	Delbert
123	Glaser
43415	Edith
9918	JS
767	Sandra
3	Swimp
634	James
724	Lee
2346	Dave
457	Bearman
312	Lisa

decimal integers

LSD Sort Example

String Case.

- Keys are a sequence from a finite ordered alphabet.
  - Example: {♣, ♠, ♥, ♦}

♠♠	Lauren
♥♦	Delbert
♦♣	Glaser
♣♥	Edith
♠♥	JS
♦♣	Sandra
♥♠	Swimp
♥♦	James
♣♠	Lee
♥♣	Dave
♣♠	Bearman
♦♠	Lisa

→

♦♣	Glaser
♦♣	Sandra
♥♣	Dave
♥♠	Swimp
♠♠	Lauren
♣♠	Lee
♣♠	Bearman
♦♠	Lisa
♠♥	JS
♣♥	Edith
♥♦	James
♥♦	Delbert

♣♠	Lee
♣♠	Bearman
♣♥	Edith
♠♠	Lauren
♠♥	JS
♥♣	Dave
♥♠	Swimp
♥♦	James
♥♦	Delbert
♦♣	Glaser
♦♣	Sandra
♦♠	Lisa

- LSD Sort
  - Sort by each digit independently, starting with the least significant.
  - Each sort is performed with key-indexed counting.

LSD Sort Example

String Case.

- Keys are a sequence from a finite ordered alphabet.
  - Example: {1, 2, 3, 4}

11	Lauren
24	Delbert
41	Glaser
13	Edith
23	JS
41	Sandra
32	Swimp
34	James
12	Lee
31	Dave
12	Bearman
42	Lisa

→

41	Glaser
41	Sandra
31	Dave
32	Swimp
22	Lauren
12	Lee
12	Bearman
42	Lisa
23	JS
13	Edith
34	James
34	Delbert

12	Lee
12	Bearman
13	Edith
22	Lauren
23	JS
31	Dave
32	Swimp
34	James
34	Delbert
41	Glaser
41	Sandra
42	Lisa

- LSD Sort
  - Sort by each digit independently, starting with the least significant.
  - Each sort is performed with key-indexed counting.

LSD Sort Example

String Case.

- Keys are a sequence from a finite ordered alphabet.
  - Example: {1, 2, 3, 4}

11	Lauren
24	Delbert
41	Glaser
13	Edith
23	JS
41	Sandra
32	Swimp
34	James
12	Lee
31	Dave
12	Bearman
42	Lisa

→

41	Glaser
41	Sandra
31	Dave
32	Swimp
22	Lauren
12	Lee
12	Bearman
42	Lisa
23	JS
13	Edith
34	James
34	Delbert

12	Lee
12	Bearman
13	Edith
22	Lauren
23	JS
31	Dave
32	Swimp
34	James
34	Delbert
41	Glaser
41	Sandra
42	Lisa

pollEv.com/jhug text to **37607**

Q: If we used heapsort instead of key-indexed counting, would LSD sort still work?

A. Yes. [689723]

B. No. [689734]

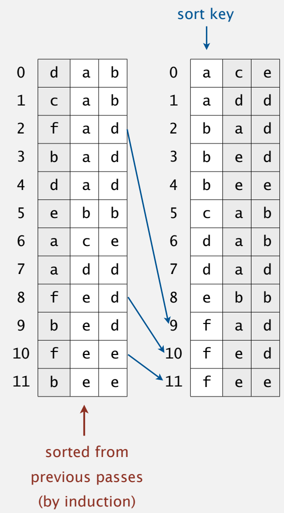
LSD string sort: correctness proof

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.



Proposition. LSD sort is stable.

LSD and fixed length strings

Q. What do we do if the strings are of different lengths?

- A1. Pad arrays with empty space at front. Treats shorter Strings as smaller.
- A2. Separately sort arrays of each observed length.
- A3. Use a different strategy than left-to-right sorting (coming up soon).

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	worst case data	random data	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	yes	compareTo()
mergesort	N lg N	N lg N	N	yes	compareTo()
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	compareTo()
heapsort	2 N lg N	2 N lg N	1	no	compareTo()
LSD †	2 W N	2 W N	N + R	yes	charAt()

* probabilistic
† fixed-length W keys

Q. How does LSD compare to Quicksort?

- Need to think about number of charAt() calls for Quicksort.

Summary of the performance of sorting algorithms

Order of growth of operation frequency.

algorithm	worst case data	random data	extra space	stable?	operations on keys
quicksort	N lg N *	N lg N	lg N	no	compareTo()
quicksort	W N lg N	N log² N	lg N	no	charAt()
LSD †	W N	W N	N + R	yes	charAt()

* probabilistic
† fixed-length W keys

charAt() is not the whole story

- Caching
- Data movement (e.g. copying aux back to a vs. partitioning)
- Experiments probably best to assess suitability to data set!

All data is strings

Consider the integer 31,992:

31992	Decimal
000000000111110011111000	Binary
7CF8	HEX

Lexicographic order may not be correct semantic order:

- Top bit should be treated differently than the rest!

-1	Decimal
111111111111111111111111	Binary
FFFFFF	HEX

String sorting interview question

Problem. Sort a billion 32-bit integers.
Ex. Google (or presidential) interview (see Coursera).

Which sorting method to use?

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

algorithm	worst case data	random data	operations on keys
LSD †	2 W N	2 W N	charAt()

pollEv.com/jhug text to **37607**

Q: If we use LSD sort to sort a billion integers, and use a 256 character alphabet, how many charAt() calls will we need to make?

A. 1 billion [689800]
B. 2 billion [689812]
C. 4 billion [689813]
D. 8 billion [689814]
E. 32 billion [689815]

String sorting interview question

Problem. Sort a billion 32-bit integers.
Ex. Google (or presidential) interview (see Coursera).

Which sorting method to use?

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

algorithm	worst case data	random data	operations on keys
LSD †	2 W N	2 W N	charAt()

pollEv.com/jhug text to **37607**

Q: If we use LSD sort to sort a billion integers, and use a 256 character alphabet, how many charAt() calls will we need to make?
C. 8 billion

256 characters is 8 bits. Treat each integer as a string of four 8-bit numbers, and thus: **W=4**. There are therefore 4 billion total characters, each of which is considered exactly twice.

Integer sorting performance summary

if we think of entire number as a digit

algorithm	worst case data	alphabet size	operations on keys	number of ops	time/op
quicksort	1.39 N lg N *	4 billion	compareTo()	160 billion	c_1
quicksort	1.39 W N lg N	256	charAt()	42 billion	c_2
LSD †	2 W N	256	charAt()	8 billion	c_2

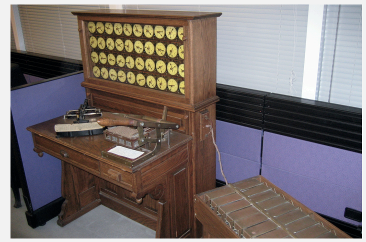
* probabilistic
† fixed-length W keys

Comparing int sorting performance of quicksort for a billion integers.
LSD Sort: 8 billion charAt() calls.
Quicksort: 1.39 10^9 lg 10^9 = 42 billion int compareTo() calls.
Quicksort with integer as String: ~160 billion charAt() calls.

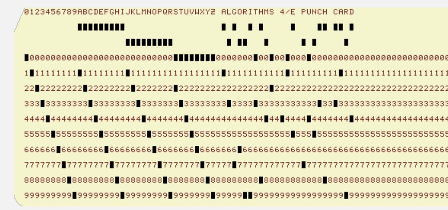
How to take a census in 1900s?

1880 Census. Took 1,500 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



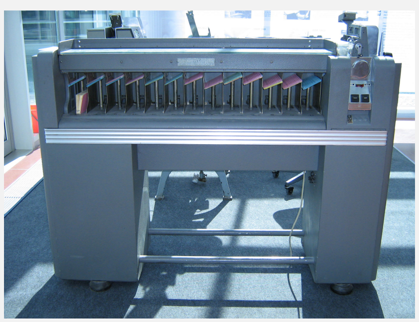
punch card (12 holes per column)

1890 Census. Finished months early 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

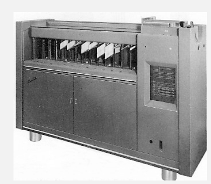
punched cards

card reader

mainframe

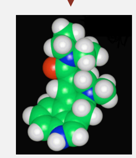
line printer

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 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

Left to right?

a	l	b	a	t	r	o	s	s	g	o	d

u	m	r	e	l	l	a	e	l	l	a	s

Moving left to right

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

sort key → sort key → sort key

Correct?

Moving left to right

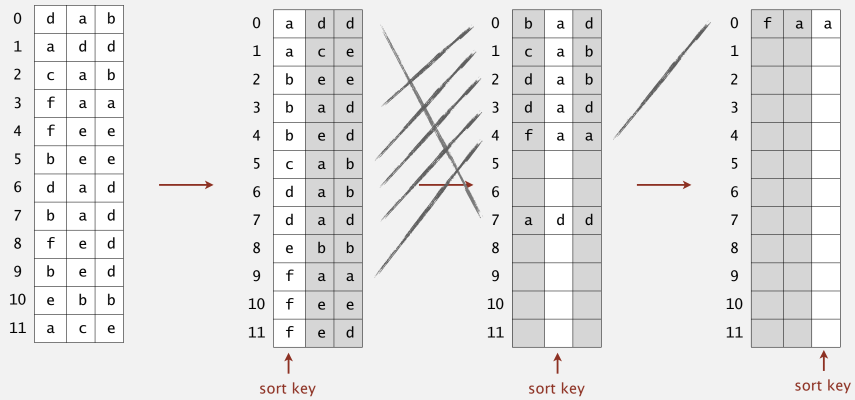
				→				
0	d	a	b		0	a	d	d
1	a	d	d		1	a	c	e
2	c	a	b		2	b	e	e
3	f	a	a		3	b	a	d
4	f	e	e		4	b	e	d
5	b	e	e		5	c	a	b
6	d	a	d		6	d	a	b
7	b	a	d		7	d	a	d
8	f	e	d		8	e	b	b
9	b	e	d		9	f	a	a
10	e	b	b		10	f	e	e
11	a	c	e		11	f	e	d

sort key → sort key → sort key

Correct?

Q: If we sort by the most significant digit (as shown above), then the middle digit, then finally the least significant digit, will we arrive at a correct result?
A. Yes.
B. No.
C. Depends on whether the sort is stable.

Moving left to right

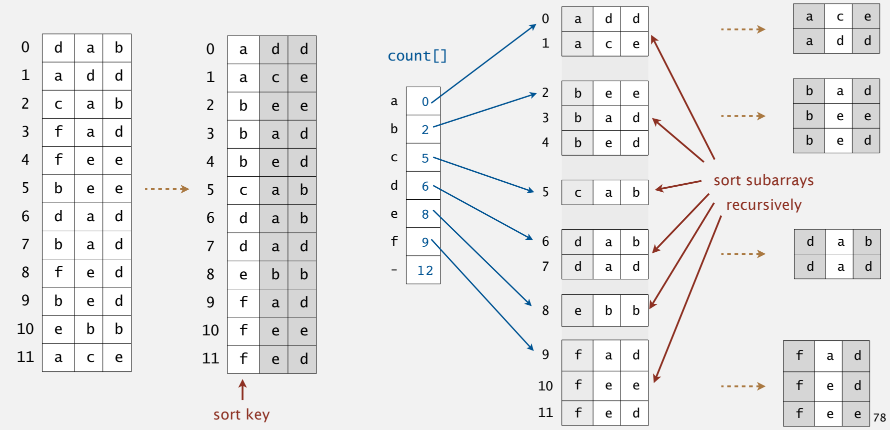


Q: If we sort by the most significant digit (as shown above), then the middle digit, then finally the least significant digit, will we arrive at a correct result?
B. No.

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

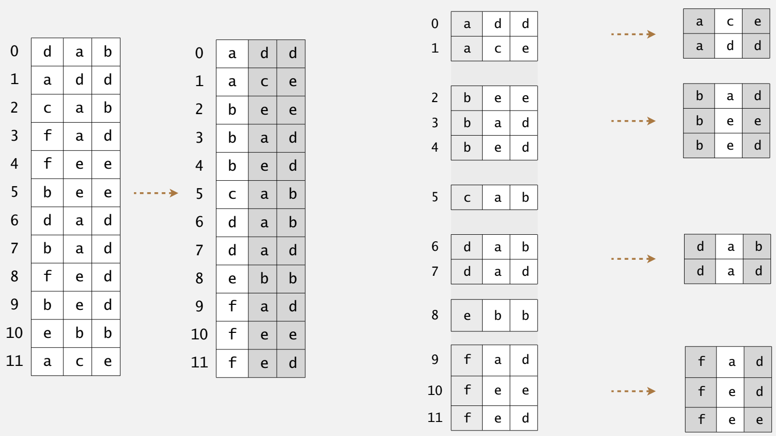
```
private static int charAt(String s, int d)
{
   if (d < s.length()) return s.charAt(d);
   else return -1;
}
```

C strings. Have extra char '\0' at end ⇒ no extra work needed.

Most-significant-digit-first string sort

pollEv.com/jhug text to **37607**

Q: If we used a **non-stable sort** instead of key indexed counting, would MSD still work?
A. Yes. [72755]
B. No. [72827]

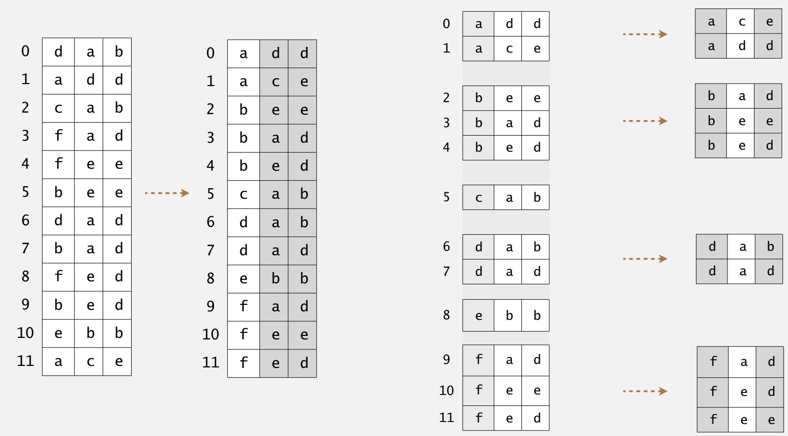


Most-significant-digit-first string sort

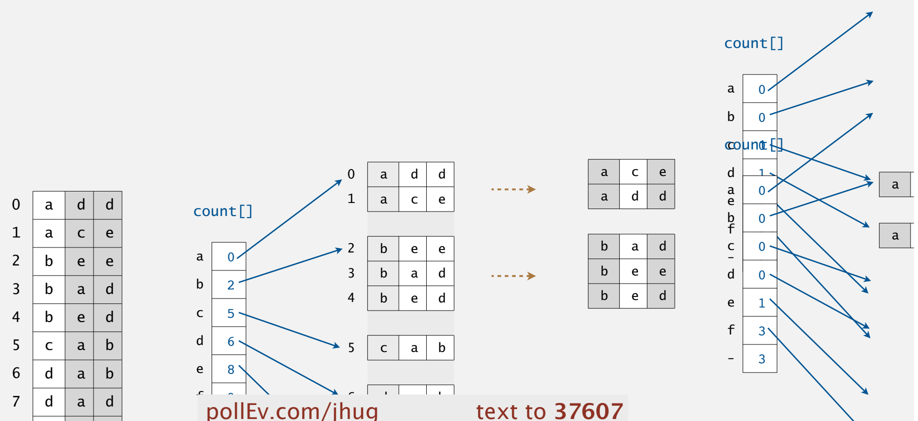
pollEv.com/jhug text to **37607**

Q: If we used a **non-stable version** of key-indexed counting, would MSD still work?
A. Yes.

Each little array is sorted independently!



Most-significant-digit-first string sort

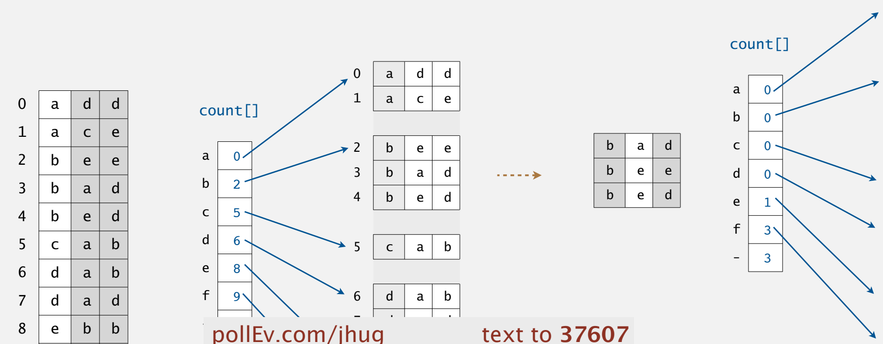


pollEv.com/jhug text to **37607**

Q: In the worst case, how much memory will our count arrays use? (Order of growth)

```
1. R N        [73779]          3. R log N      [73815]
2. R W        [73790]          4. R log W      [73819]
                               5. R N W        [73824]
Let: N = number of strings. W = width of widest string. R = radix.
```


Most-significant-digit-first string sort



pollEv.com/jhug text to **37607**

Q: In the worst case, how much memory will our count arrays use?

```
B. R W

Number of count arrays = recursion depth = W.
Size of count arrays = R.
```

MSD string sort: Java implementation

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

can recycle aux[] array but not count[] array

key-indexed counting

sort R subarrays recursively

MSD string sort: potential for disastrous performance

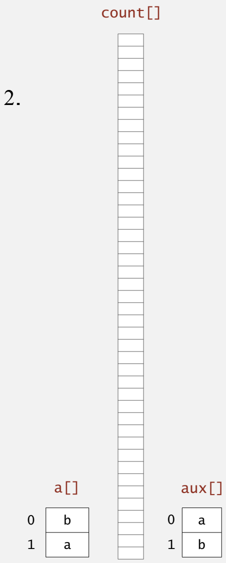
Observation 1. Much too slow for small subarrays.

- 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.

Consider hi = 1, lo = 0

```
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];
```



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.
- Implement `less()` so that it compares starting at d^{th} character.

```
public 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);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

Warning: In Java 7, this could be very slow!

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! ← Here, input size is total number of characters.

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
quicksort	W N lg N	N log² N	lg N	no	charAt()
LSD †	N W	N W	N + R	yes	charAt()
MSD ‡	N W	N log $_R$ N	N + D R	yes	charAt()

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

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

charAt() is not the whole story

- Caching
- Creating count arrays
- Data movement (e.g. copying aux back to a vs. partitioning)

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Extra space for aux[].
- Extra space for count[].
  - Really bad if you have long prefix matches!
- Inner loop has a lot of instructions.
- Accesses memory "randomly" (cache inefficient).

R D space, D is longest match

Disadvantage of quicksort.

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

Doesn't rescan anything! Doesn't create counting arrays!

Goal. Combine advantages of MSD and quicksort.

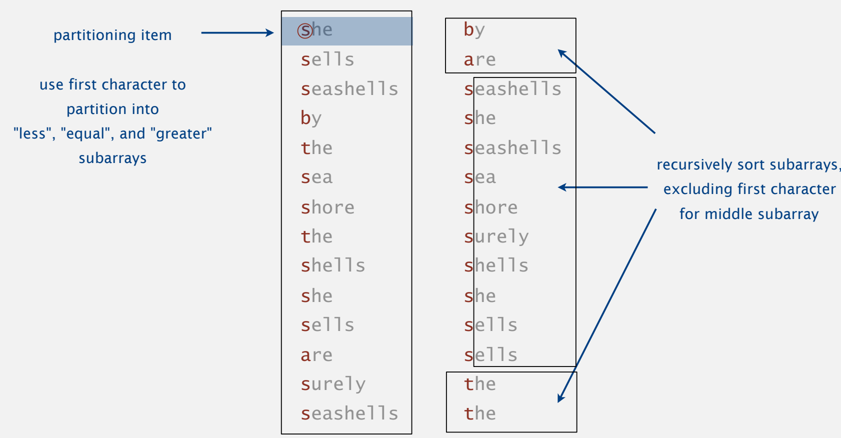
5.1 String Sorts

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

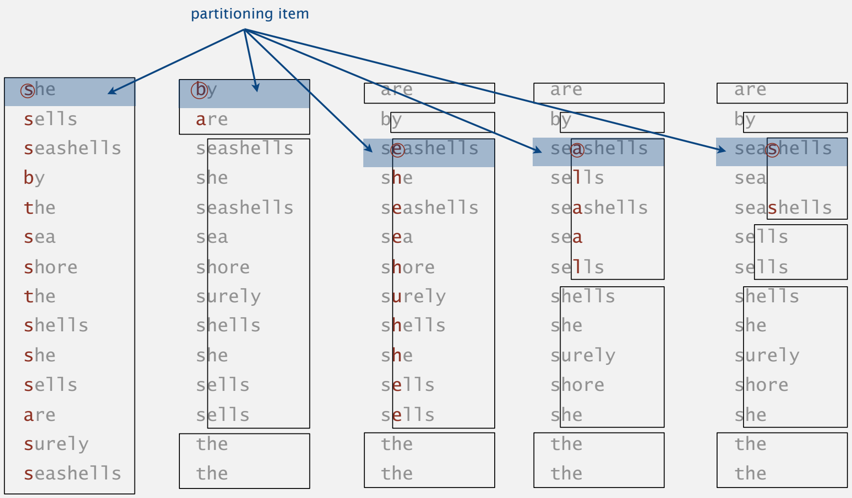
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the d^{th} character. ← Instead of sorting!

- 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

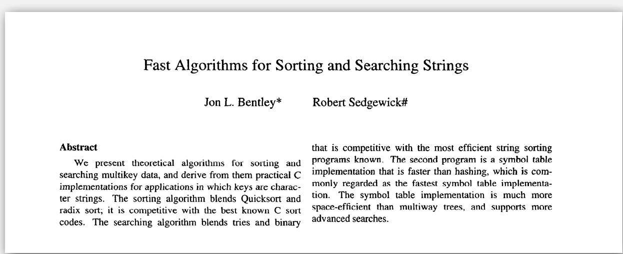
3-way string quicksort vs. standard quicksort

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

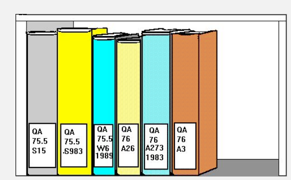
MSD string sort.

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

3-way string quicksort.

- Has a short inner loop.
- Is cache-friendly.
- Is in-place.
- Performs more `charAt()` calls.
  - But this doesn't matter!

library of Congress call numbers



Flipped lecture this week:
Experiments to see when this is true.

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
quicksort	W N lg N	N $\log^2$ N	lg N	no	charAt()
LSD †	N W	N W	N + R	yes	charAt()
MSD ‡	N W	N $\log_R$ N	N + D R	yes	charAt()
3-way string quicksort	1.39 W N lg R *	1.39 N lg N	log N + W	no	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*

Challenge #1: 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      ← characters of surrounding context
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
```

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		sort suffixes to bring repeated substrings together	
0	i t w a s b e s t i t w a s w	3	a s b e s t
1	t w a s b e s t i t w a s w	12	a s w
2	w a s b e s t i t w a s w	5	b e s t i t w a s w
3	a s b e s t i t w a s w	6	e s t i t w a s w
4	s b 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
5	b e s t i t w a s w	9	i t w a s w
6	e s t 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	7	s t i t w a s w
8	t i t w a s w	13	s w
9	i t w a s w	8	t i t w a s w
10	t w a s w	1	t w a s b e s t i t w a s w
11	w a s w	10	t w a s w
12	a s w	14	w
13	s w	2	w a s b e s t i t w a s w
14	w	11	w a s w

Non-trivial Java task! See Burrows-Wheeler assignment on Coursera if you're interested.

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 …
	⋮

Challenge #2: Longest repeated substring

Given a string of N characters, find the longest repeated substring.

```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a c t c t a t a t c t a t a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, …

Challenge #2: Longest repeated substring: a musical application

Visualize repetitions in music. `http://www.bewitched.com`

Mary Had a Little Lamb

Bach's Goldberg Variations

Very cool algorithm! See Coursera for details

Simple Solution: Form sorted suffixes array and scan. D^2N

Linearithmic Solution: Use special sequence of sorts (Manber-Myers): $N \lg N$

String sorting summary

We can develop linear-time sorts (e.g. LSD).

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

We can develop sublinear-time sorts (e.g. MSD, 3-way radix quicksort).

- 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.