5. Strings

Algorithms in Java, 4th Edition · Robert Sedgewick and Kevin Wayne · *· January 26, 2010 8:15:36 AM*

String processing

String. Sequence of characters.

Important fundamental abstraction.
- Java programs.
- Natural languages.
- 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.
- Need more bits to represent certain 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

Java char data type. A 16-bit unsigned integer.
- Supports original 16-bit Unicode.
- Awkwardly supports 21-bit Unicode 3.0.

The String data type

Character extraction. Get the i^{th} character.
Substring extraction. Get a contiguous sequence of characters from a string.
String concatenation. Append one character to end of another string.

s	t	r	i	n	g	s
0	1	2	3	4	5	6

```
String s = "strings";           // s = "strings"
char   c = s.charAt(2);         // c = 'r'
String t = s.substring(2, 6);   // t = "ring"
String u = t + c;               // u = "ringr"
```

Implementing strings in Java

Java strings are immutable ⇒ two strings can share underlying `char[]` array.

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

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

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

   public char charAt(int index)
   {  return value[index + offset];  }
   …
}
```

java.lang.String

constant time

Implementing strings in Java

```
public String concat(String that)
{
   char[] buffer = new char[this.length() + that.length()];
   for (int i = 0; i < this.length(); i++)
       buffer[i] = this.value[i];
   for (int j = 0; j < that.length(); j++)
       buffer[this.length() + j] = that.value[j];
   return new String(0, this.length() + that.length(), buffer);
}
```

Memory. 40 + 2N bytes for a virgin `String` of length N.

use `byte[]` or `char[]` instead of `String` to save space

operation	guarantee	extra space
`charAt()`	1	1
`substring()`	1	1
`concat()`	N	N

`String` vs. `StringBuilder`

`String`. [immutable] Constant substring, linear concatenation.

`StringBuilder`. [mutable] Linear substring, constant (amortized) append.

Ex. Reverse a `String`.

```
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

```
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

String challenge: array of suffixes

Challenge. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a a c a a g t t t a c a a g c
1	a c a a g t t t a c a a g c
2	c a a g t t t a c a a g c
3	a a g t t t a c a a g c
4	a g t t t a c a a g c
5	g t t t a c a a g c
6	t t t a c a a g c
7	t t a c a a g c
8	t a c a a g c
9	a c a a g c
10	c a a g c
11	a a g c
12	a g c
13	g c
14	c

String challenge: array of suffixes

Challenge. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);
   return suffixes;
}
```

← linear time and space

B.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   StringBuilder sb = new StringBuilder(s);
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = sb.substring(i, N);
   return suffixes;
}
```

← quadratic time and space!

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*

Standard alphabets

6.1 Sorting Strings

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string 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	$N^2/2$	$N^2/4$	no	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	no	no	`compareTo()`

* probabilistic

Lower bound. ~ N lg N compares are required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?
A. Yes, if we don't depend on compares.

- **key-indexed counting**
- LSD string sort
- MSD string sort
- 3-way radix quicksort
- longest repeated substring

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.

Remark. Keys may have associated data ⇒ 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

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

- Count frequencies of each letter using key as index.
-
-
-

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

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.
-
-

compute cumulates →

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

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

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

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

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

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

r	count[r]
a	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

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 records.*
-

```
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 records →

i	a[i]	aux[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

Key-indexed counting

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 records.*
-

```
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 records →

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

r	count[r]
a	0
b	2
c	5
d	7
e	8
f	9
-	12

Key-indexed counting

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 records.*
-

```
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 records →

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

r	count[r]
a	1
b	2
c	5
d	7
e	8
f	9
-	12

Key-indexed counting

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 records.*
-

```
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 records →

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

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	9
-	12

Key-indexed counting

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 records.
-

```
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 records →

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	1
b	2
c	6
d	7
e	8
f	10
-	12

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

Key-indexed counting

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 records.
-

```
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 records →

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	1
b	2
c	6
d	7
e	8
f	11
-	12

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

Key-indexed counting

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 records.
-

```
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 records →

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	1
b	3
c	6
d	7
e	8
f	11
-	12

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

Key-indexed counting

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 records.
-

```
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 records →

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	1
b	3
c	6
d	8
e	8
f	11
-	12

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

Key-indexed counting

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 records.
-

```
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 records →

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

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	11
-	12

Key-indexed counting

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 records.
-

```
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 records →

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

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	12
-	12

Key-indexed counting

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 records.
-

```
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 records →

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

r	count[r]
a	1
b	5
c	6
d	8
e	8
f	12
-	12

Key-indexed counting

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 records.
-

```
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 records →

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

r	count[r]
a	1
b	5
c	6
d	8
e	9
f	12
-	12

Key-indexed counting

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 records.
-

```
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 records →

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

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 records.
-

```
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 records →

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

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 records.
- 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

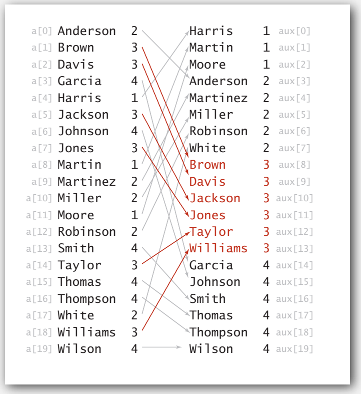
i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting: analysis

Proposition. Key-indexed counting takes time proportional to N + R to sort N records whose keys are integers between 0 and R-1.

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

Stable? Yes!



Least-significant-digit-first radix sort

LSD string sort.

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

		sort key ↓	sort key ↓	sort key ↓
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 must be stable
(arrows do not cross)

LSD string sort: correctness proof

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

Pf. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.

	in order by previous passes	sort key ↓
0	d a b	a c e
1	c a b	a d d
2	f a d	b a d
3	b a d	b e d
4	d a d	b e e
5	e b b	c a b
6	a c e	d a b
7	a d d	d a d
8	f e d	e b b
9	b e d	f a d
10	f e e	f e d
11	b e e	f e e

LSD string sort: Java implementation

```
public class LSD
{
   public static void sort(String[] a, int W)      ← fixed-length W strings
   {
      int R = 256                                  ← radix R
      int N = a.length;
      String[] aux = new String[N];
      for (int d = W-1; d >= 0; d--)               ← do key-indexed counting
      {                                              for each digit from right to left
         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];                ← key-indexed counting
         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];
      }
   }
}
```

LSD string sort: example

Input	d = 6	d = 5	d = 4	d = 3	d = 2	d = 1	d = 0	Output
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	2RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Summary of the performance of sorting algorithms

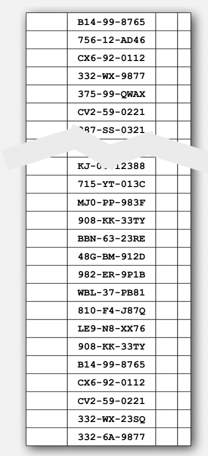
Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2/2$	$N^2/4$	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

Sorting challenge 1

Problem. Sort a huge commercial database on a fixed-length key field.
Ex. Account number, date, SS number, ...



Which sorting method to use?
- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ LSD string sort.

256 (or 65536) counters;
Fixed-length strings sort in W passes.

Sorting challenge 2a

Problem. Sort 1 million 32-bit integers.
Ex. Google interview or presidential interview.

Which sorting method to use?
- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

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

card punch

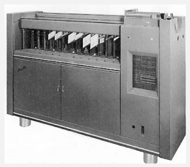
punched cards

card reader

mainframe

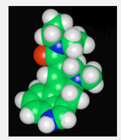
line printer

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)

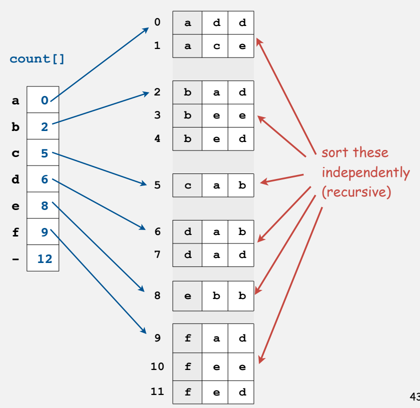
Most-significant-digit-first string sort

MSD string sort.
- Partition file 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).

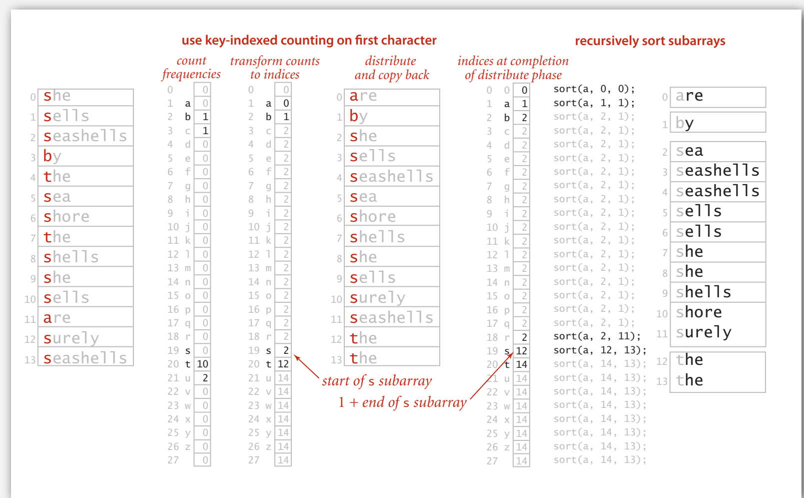
0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

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

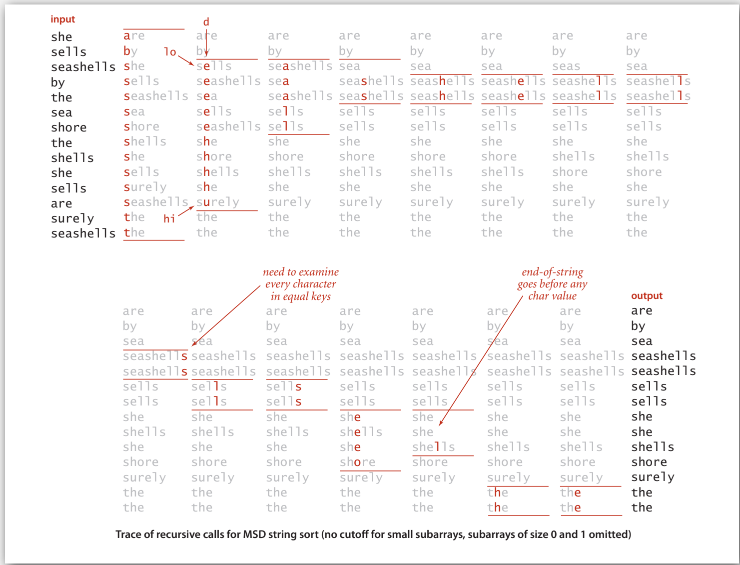
sort key



MSD string sort: top level trace



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).

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.

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[] but not count[]

key-indexed counting

recursively sort subarrays

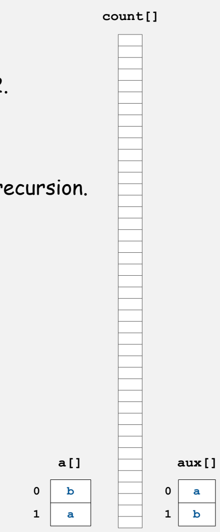
MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- The count[] array must be re-initialized.
- ASCII (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 32,000x slower for N = 2.

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

Solution. Cutoff to insertion sort for small N.



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small N.

- 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;  }
```

in Java, forming and comparing substrings is faster than directly comparing chars with `charAt()` !

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!

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	$N^2/2$	$N^2/4$	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 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`

stack depth D = 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.

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

Disadvantage of quicksort.

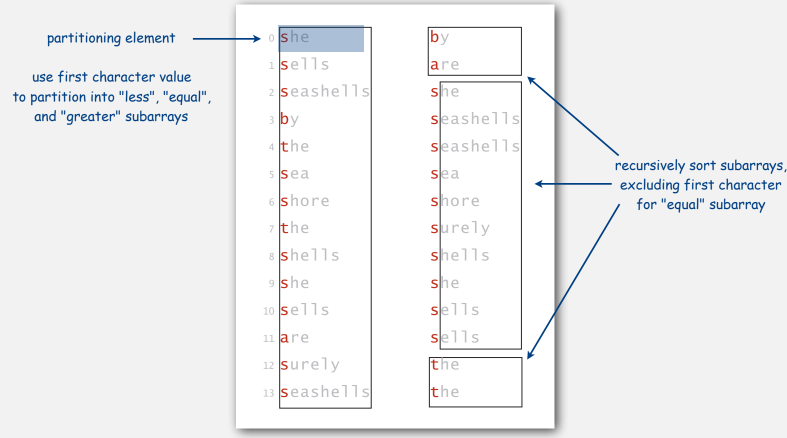
- Linearithmic number of string compares (not linear).
- Has to rescan long keys for compares.
  [but stay tuned]

- key-indexed counting
- LSD string sort
- MSD string sort
- **3-way string quicksort**
- suffix arrays

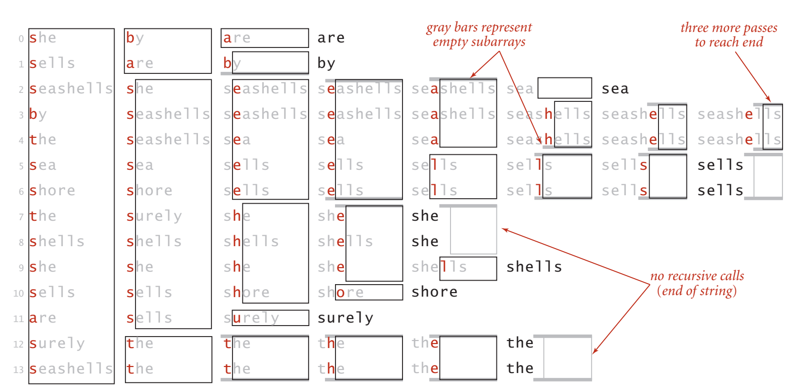
3-way string quicksort (Bentley and Sedgewick, 1997)

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

- Cheaper than R-way partitioning of MSD string sort.
- Need not examine again characters equal to the partitioning char.



3-way string quicksort: trace of recursive calls



Trace of recursive calls for 3-way string quicksort (no cutoff for small subarrays)

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)
{
   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

sort 3 pieces recursively

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses 2N ln N string compares on average.
- Costly for long keys that differ only at the end (and this is a common case!)

3-way string quicksort.

- Uses 2 N ln N character compares on average for random strings.
- Avoids recomparing initial parts of the string.
- Adapts to data: uses just "enough" characters to resolve order.
- Sublinear when strings are long.

Proposition. 3-way string quicksort is optimal (to within a constant factor); no sorting algorithm can (asymptotically) examine fewer chars.

Pf. Ties cost to entropy. Beyond scope of 226.

3-way string quicksort vs. MSD string sort

MSD string sort.

- Has a long inner loop.
- Is cache-inefficient.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

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

library call numbers

```
WUS-------10706-----7---10
WUS-------12692-----4---27
WLSOC------2542----30
LTK--6015-P-63-1988
LDS---361-H-4
...
```

Bottom line. 3-way string quicksort is the 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	$N^2 /2$	$N^2 /4$	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 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`
3-way string quicksort	1.39 W N lg N *	1.39 N lg N	log N + W	no	`charAt()`

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

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way radix quicksort
- **suffix arrays**

Warmup: longest common prefix

LCP. Given two strings, find the longest substring that is a prefix of both.

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

```
public static String 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 s.substring(0, i);
   }
   return s.substring(0, n);
}
```

Running time. Linear-time in length of prefix match.
Space. Constant extra space.

Longest repeated substring

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

Ex.

```
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, ...

Longest repeated substring: a musical application

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

Mary Had a Little Lamb

Bach's Goldberg Variations

Longest repeated substring

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

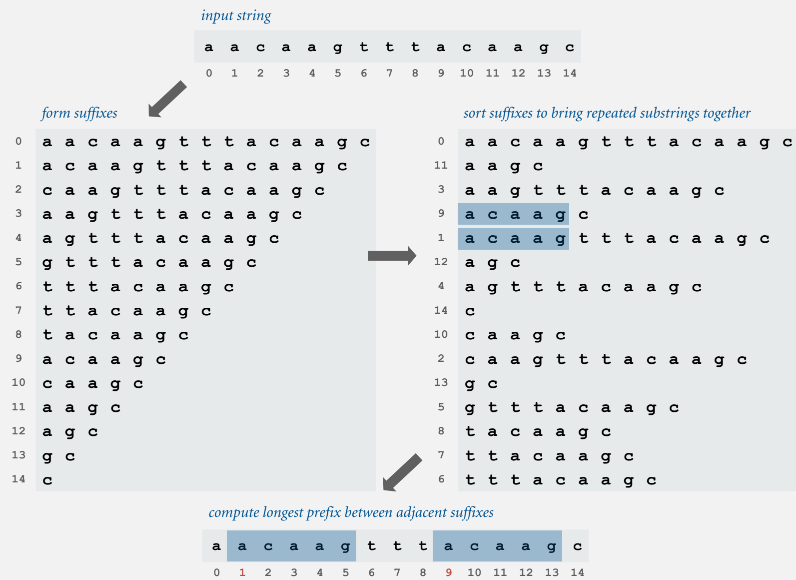
Brute force algorithm.

- Try all indices `i` and `j` for start of possible match.
- Compute longest common prefix (LCP) for each pair.

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
	i								j					

Analysis. Running time ≤ M N^2 , where M is length of longest match.

Longest repeated substring: a sorting solution



Longest repeated substring: Java implementation

```
public String lrs(String s)
{
   int N = s.length();

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

   Arrays.sort(suffixes);

   String lrs = "";
   for (int i = 0; i < N-1; i++)
   {
      String x = lcp(suffixes[i], suffixes[i+1]);
      if (x.length() > lrs.length()) lrs = x;
   }
   return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between suffixes that are adjacent after sorting

```
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```


Sorting challenge

Problem. Five scientists A, B, C, D, and E are looking for long repeated substring in a genome with over 1 billion nucleotides.

- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- ✓ E uses suffix sorting solution with 3-way string quicksort.

only if LRS is not long (!)

Q. Which one is more likely to lead to a cure cancer?

Longest repeated substring: empirical analysis

input file	characters	brute	suffix sort	length of LRS
LRS.java	2,162	0.6 sec	0.14 sec	73
amendments.txt	18,369	37 sec	0.25 sec	216
aesop.txt	191,945	1.2 hours	1.0 sec	58
mobydick.txt	1.2 million	43 hours †	7.6 sec	79
chromosome11.txt	7.1 million	2 months †	61 sec	12,567
pi.txt	10 million	4 months †	84 sec	14

† estimated

Suffix sorting: worst-case input

Longest repeated substring not long. Hard to beat 3-way string quicksort.

Longest repeated substring very long.

- Radix sorts are quadratic in the length of the longest match.
- Ex: two copies of Aesop's fables.

```
% more abcdefgh2.txt
abcdefgh
abcdefghabcdefgh
bcdefgh
bcdefghabcdefgh
cdefgh
cdefghabcdefgh
defgh
efghabcdefgh
efgh
fghabcdefgh
fgh
ghabcdefgh
fh
habcdefgh
h
```

algorithm	time to suffix sort (seconds): mobydick.txt	time to suffix sort (seconds): aesopaesop.txt
brute-force	36,000 †	4000 †
quicksort	9.5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way string quicksort	2.8	400

† estimated

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length N.

Q. What is worst-case running time of best algorithm for problem?

- Quadratic.
- ✓ Linearithmic. ← Manber's algorithm
- ✓ Linear. ← suffix trees (see COS 423)
- Nobody knows.

Suffix sorting in linearithmic time

Manber's MSD algorithm.

- Phase 0: sort on first character using key-indexed counting sort.
- Phase i: given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters.

Worst-case running time. N log N.

- Finishes after lg N phases.
- Can perform a phase in linear time. (!) [stay tuned]

Linearithmic suffix sort example: phase 0

	original suffixes		*key-indexed counting sort (first character)*
0	b a b a a a a b c b a b a a a a a 0	17	0
1	a b a a a a b c b a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0	16	a 0
3	a a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0	5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0	6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0	15	a a 0
8	c b a b a a a a a 0	14	a a a 0
9	b a b a a a a a 0	13	a a a a 0
10	a b a a a a a 0	12	a a a a a 0
11	b a a a a a 0	10	a b a a a a a 0
12	a a a a a 0	0	b a b a a a a b c b a b a a a a a 0
13	a a a a 0	9	b a b a a a a a 0
14	a a a 0	11	b a a a a a 0
15	a a 0	7	b c b a b a a a a a 0
16	a 0	2	b a a a a b c b a b a a a a a 0
17	0	8	c b a b a a a a a 0

↑ *sorted*

Linearithmic suffix sort example: phase 1

	original suffixes		*index sort (first two characters)*
0	b a b a a a a b c b a b a a a a a 0	17	0
1	a b a a a a b c b a b a a a a a 0	16	a 0
2	b a a a a b c b a b a a a a a 0	12	a a a a a 0
3	a a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0	5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0	13	a a a a 0
7	b c b a b a a a a a 0	15	a a 0
8	c b a b a a a a a 0	14	a a a 0
9	b a b a a a a a 0	6	a b c b a b a a a a a 0
10	a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0
11	b a a a a a 0	10	a b a a a a a 0
12	a a a a a 0	0	b a b a a a a b c b a b a a a a a 0
13	a a a a 0	9	b a b a a a a a 0
14	a a a 0	11	b a a a a a 0
15	a a 0	2	b a a a a b c b a b a a a a a 0
16	a 0	7	b c b a b a a a a a 0
17	0	8	c b a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 2

	original suffixes		*index sort (first four characters)*
0	b a b a a a a b c b a b a a a a a 0	17	0
1	a b a a a a b c b a b a a a a a 0	16	a 0
2	b a a a a b c b a b a a a a a 0	15	a a 0
3	a a a a b c b a b a a a a a 0	14	a a a 0
4	a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0	12	a a a a a 0
6	a b c b a b a a a a a 0	13	a a a a 0
7	b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0
9	b a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0
10	a b a a a a a 0	10	a b a a a a a 0
11	b a a a a a 0	6	a b c b a b a a a a a 0
12	a a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0
13	a a a a 0	11	b a a a a a 0
14	a a a 0	0	b a b a a a a b c b a b a a a a a 0
15	a a 0	9	b a b a a a a a 0
16	a 0	7	b c b a b a a a a a 0
17	0	8	c b a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 3

	original suffixes		*index sort (first eight characters)*
0	b a b a a a a b c b a b a a a a a 0	17	0
1	a b a a a a b c b a b a a a a a 0	16	a 0
2	b a a a a b c b a b a a a a a 0	15	a a 0
3	a a a a b c b a b a a a a a 0	14	a a a 0
4	a a a b c b a b a a a a a 0	13	a a a a 0
5	a a b c b a b a a a a a 0	12	a a a a a 0
6	a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0
7	b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0
9	b a b a a a a a 0	10	a b a a a a a 0
10	a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0
11	b a a a a a 0	6	a b c b a b a a a a a 0
12	a a a a a 0	11	b a a a a a 0
13	a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0
14	a a a 0	9	b a b a a a a a 0
15	a a 0	0	b a b a a a a b c b a b a a a a a 0
16	a 0	7	b c b a b a a a a a 0
17	0	8	c b a b a a a a a 0

sorted

FINISHED! (no equal keys)

Achieve constant-time string compare by indexing into inverse

	original suffixes		*index sort (first four characters)*		*inverse*
0	b a b a a a a b c b a b a a a a a 0	17	0	0	14
1	a b a a a a b c b a b a a a a a 0	16	a 0	1	9
2	b a a a a b c b a b a a a a a 0	15	a a 0	2	12
3	a a a a b c b a b a a a a a 0	14	a a a 0	3	4
4	a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0	4	7
5	a a b c b a b a a a a a 0	12	a a a a a 0	5	8
6	a b c b a b a a a a a 0	13	a a a a 0	6	11
7	b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0	7	16
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0	8	17
9	b a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0	9	15
10	a b a a a a a 0	10	a b a a a a a 0	10	10
11	b a a a a a 0	6	a b c b a b a a a a a 0	11	13
12	a a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0	12	5
13	a a a a 0	11	b a a a a a 0	13	6
14	a a a 0	0	b a b a a a a b c b a b a a a a a 0	14	3
15	a a 0	9	b a b a a a a a 0	15	2
16	a 0	7	b c b a b a a a a a 0	16	1
17	0	8	c b a b a a a a a 0	17	0

0 + 4 = 4

9 + 4 = 13

$\texttt{suffixes}_4\texttt{[13]} \le \texttt{suffixes}_4\texttt{[4]}$ (*because* $\texttt{inverse[13]} < \texttt{inverse[4]}$)

so $\texttt{suffixes}_8\texttt{[9]} \le \texttt{suffixes}_8\texttt{[0]}$

Suffix sort: experimental results

algorithm	time to suffix sort (seconds) `mobydick.txt`	`aesopaesop.txt`
brute-force	36.000 †	4000 †
quicksort	9.5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way string quicksort	2.8	400
Manber MSD	17	8.5

† estimated

String sorting summary

We can develop linear-time sorts.

- Compares not necessary for string keys.
- Use digits to index an array.

We can develop sublinear-time sorts.

- Should measure 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.