

6 Strings

- ▶ 6.1 radix sorts
- ▶ 6.2 tries
- ▶ 6.3 substring search
- ▶ 6.4 pattern matching
- ▶ 6.5 data compression

References:
Algorithms (4th edition), Chapter 6
<http://www.cs.princeton.edu/algs4/60string>

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

Using strings in Java: review

String concatenation. Append one character to end of another string.

Substring. Extract a contiguous sequence of characters from a 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"
```

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 VS. StringBuilder

String. [immutable] Constant substring, linear concatenation.

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

Ex. Form array of suffixes.

```
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

```
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!

5

Implementing strings in Java

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

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

java.lang.String

```
public final class String implements Comparable<String>
{
    private char[] value; // characters
    private int offset; // index of first char into 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); }
    ...
}
```

6

6.1 Radix Sorts

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

References:

Algorithms in Java (3rd edition), Chapter 10
<http://www.cs.princeton.edu/algs4/61radix>

Review: summary of the performance of sorting algorithms

Number of operations on keys = compares.

algorithm	guarantee	typical	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()

Lower bound. $\sim N \lg N$ 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.

8

Digital keys

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits in alphabet.

Applications.

- DNA: sequence of a, c, g, t.
- IPv6 address: sequence of 128 bits.
- ASCII: sequence of 7-bit characters.
- Protein: sequence of amino acids A, C, ..., Y.
- English words: sequence of lowercase letters.
- Credit card number: sequence of 16 decimal digits.
- International words: sequence of Unicode characters.
- Library call numbers: sequence of letters, numbers, periods.

This lecture. `string` of ASCII characters.

9

▶ key-indexed counting

- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ longest repeated substring

10

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 precept.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm.

Remark. Keys may have associated data \Rightarrow can't just count up number of keys of each value.

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.

-
-
-

```
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++)
    temp[count[a[i]]++] = a[i];

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

count
frequencies

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

offset by 1
[stay tuned]

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.
-
-

```
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++)
    temp[count[a[i]]++] = a[i];

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

compute cumulates

13

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++)
    temp[count[a[i]]++] = a[i];

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

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

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

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

move records

14

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++)
    temp[count[a[i]]++] = a[i];

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

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	temp[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

copy back

15

Key-indexed counting: analysis

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

Running time. Takes time proportional to $N + R$.

Extra space.

- Array of size R (for counting).
- Array of size N (for rearrangement).

↑
inplace version is possible and practical

Stable? Yes!

16

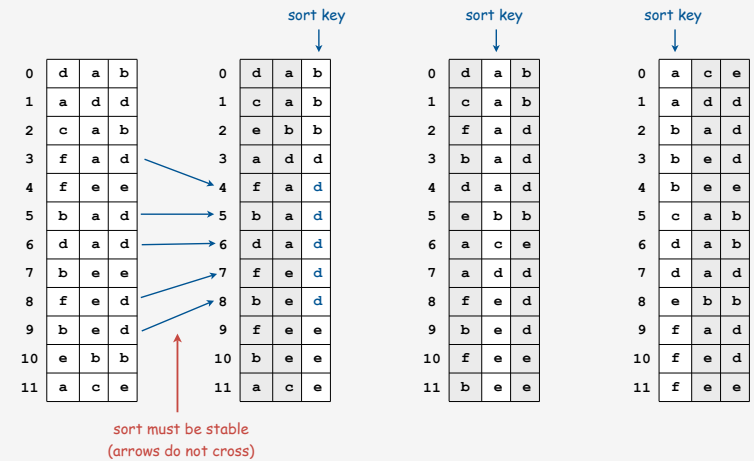
- › key-indexed counting
- › **LSD radix sort**
- › MSD radix sort
- › 3-way radix quicksort
- › longest repeated substring

17

Least-significant-digit-first radix sort

LSD radix sort.

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



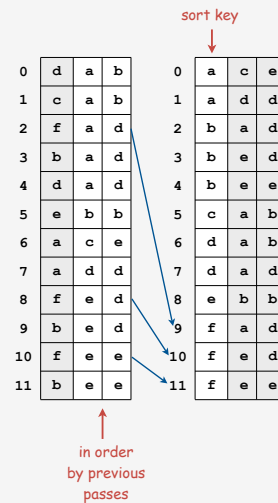
18

LSD radix 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.



19

LSD radix sort: Java implementation

```

public class LSD
{
    public static void sort(String[] a)
    {
        int R = 256;
        int W = a[0].length;
        int N = a.length;
        String[] temp = 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++)
                temp[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = temp[i];
        }
    }
}

```

radix R,
fixed-length W strings

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

key-indexed counting

20

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	typical	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	no	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	no	no	<code>compareTo()</code>
LSD †	$W(N + R)$	$W(N + R)$	$N + R$	yes	<code>charAt()</code>

† fixed-length W keys

21

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 radix sort.

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

B14-99-8765		
756-12-AD46		
CX6-92-0112		
332-WX-9877		
375-99-QWAX		
CV2-59-0221		
187-SS-0321		
KJ-0-12388		
715-YZ-013C		
MJ0-FP-983F		
908-KK-33TY		
BBN-63-23RE		
48G-EM-912D		
982-ER-9P1B		
WBL-37-PB81		
810-F4-J87Q		
LES-N8-XC76		
908-KK-33TY		
B14-99-8765		
CX6-92-0112		
CV2-59-0221		
332-WX-23SQ		
332-6A-9877		

22

Sorting challenge 2

Problem. Sort 1 million 32-bit integers.

Ex. Google interview or presidential interview.

Which sorting method to use?

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



23

Sorting challenge 2

Problem. Sort huge files of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

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

↑
 Divide each word into eight 16-bit "chars."
 $2^{16} = 65536$ counters.
 Sort in 8 passes.

011101101101101101...1011101

24

Sorting challenge 2

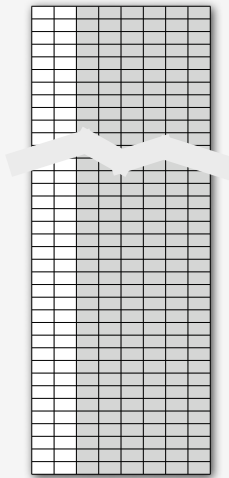
Problem. Sort huge files of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

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

Divide each word into 16-bit "chars"
 $2^{16} = 65536$ counters
 LSD sort on leading 32 bits in 2 passes
 Finish with insertion sort
 Examines only ~25% of the data



25

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



card punch



punched cards



card reader

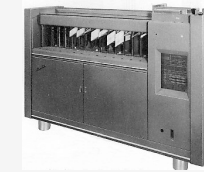


mainframe



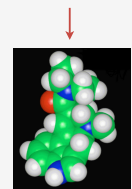
line printer

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)

26

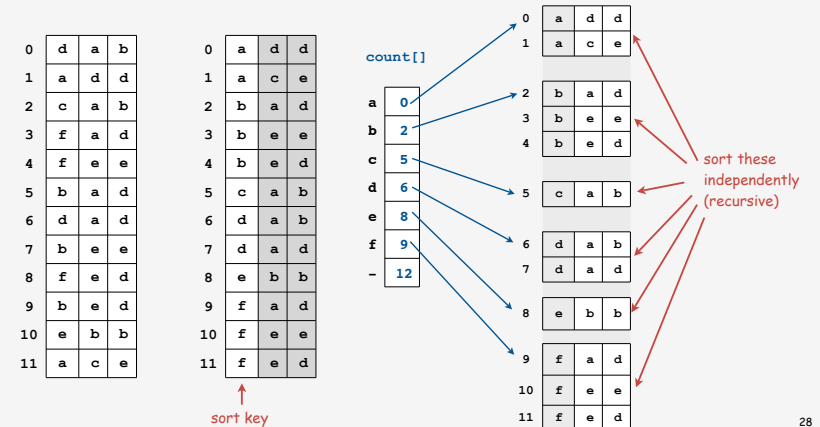
- › key-indexed counting
- › LSD radix sort
- › **MSD radix sort**
- › 3-way radix quicksort
- › longest repeated substring

27

Most-significant-digit-first radix sort

MSD radix 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).

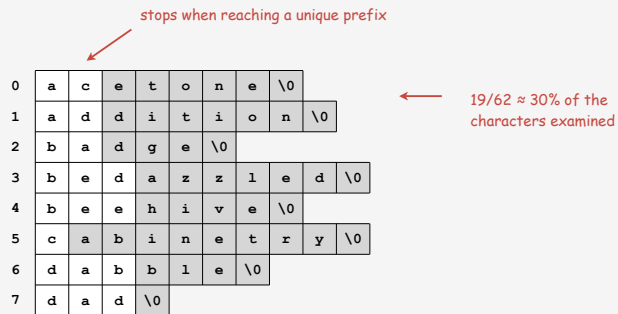


28

MSD radix sort bonuses

Bonus 1. May not have to examine all of the keys.

Bonus 2. Works for variable-length keys, e.g., '\0'-terminated string.



Implication. Sublinear-time sort (!) [assuming data is already in memory]

29

MSD radix sort: Java implementation

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

private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo + 1) return;

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

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

key-indexed counting

recursively sort R-1 subarrays

assumes strings are '\0' terminated;
don't sort substrings that start with '\0'

30

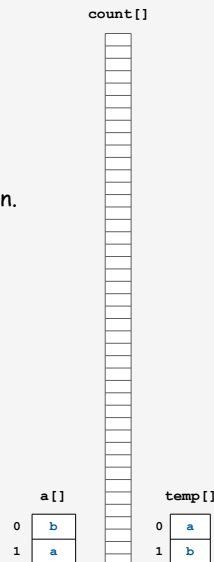
MSD radix sort: potential for disastrous performance

Observation 1. Much too slow for small files.

- 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 files because of recursion.

Solution. Cutoff to insertion sort for small N .



31

Summary of the performance of sorting algorithms

algorithm	guarantee	typical	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	no	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	no	no	<code>compareTo()</code>
LSD †	$W(N + R)$	$W(N + R)$	$N + R$	yes	<code>charAt()</code>
MSD ‡	$W(N + R)$	$c N \log N$	$D(N + R)$	yes	<code>charAt()</code>

just enough to distinguish
 N ASCII strings: $N \log_{128} N$
(usually sublinear in input size WN)

stack depth $D =$ length of
longest prefix match

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

32

MSD radix sort vs. quicksort for strings

Disadvantages of MSD radix sort.

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

Disadvantage of quicksort.

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

33

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

34

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

Idea. Do 3-way partitioning on the *d*th character.

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

`qsortX(0, 12, 0)`

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	a	c	e
10	e	b	b
11	b	e	e

↑
3-way partition
0th char on b

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

↑
result of 3-way partition

only examine string
starting at index 1

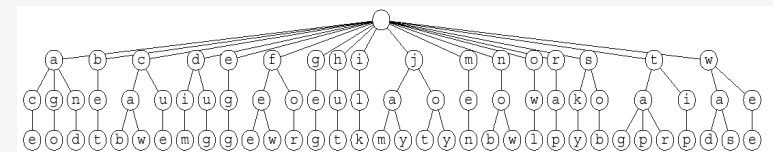
0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	f	a	d
6	d	a	d
7	c	a	b
8	f	e	d
9	d	a	b
10	e	b	b
11	f	e	e

recursively sort 3 pieces

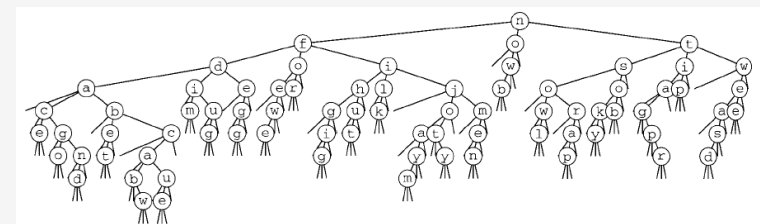
35

Recursive structure: MSD radix sort vs. 3-way radix quicksort

3-way radix quicksort collapses empty links in MSD recursion tree.



MSD radix sort recursion tree
(1035 null links, not shown)



3-way radix quicksort recursion tree
(155 null links)

36

3-way radix 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 radix quicksort.

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

Proposition. 3-way radix 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.

37

3-way radix quicksort vs. MSD radix sort

MSD radix sort.

- Has a long inner loop.
- Is cache-inefficient.
- Too much overhead reinitializing `count[]` and `temp[]` for long keys that differ only at end (and this is a common case!)

3-way radix 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
LTR--6015-P-63-1988
LDS---361-H-4
...
```

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

38

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

39

Warmup: longest common prefix

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

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

40

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

Applications. Bioinformatics, cryptanalysis, data compression, ...

41

Longest repeated substring: a musical application

Visualize repetitions in music. <http://www.bewitched.com>

Mary Had a Little Lamb



Bach's Goldberg Variations



42

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.



Analysis. Running time $\leq M N^2$, where M is length of longest match.

43

Longest repeated substring: a sorting solution

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
    
```

form 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
    
```

sort suffixes to bring repeated substrings together

```

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

compute longest prefix between adjacent suffixes

```

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
    
```

44

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

45

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 radix sort.
- ✓ • E uses suffix sorting solution with 3-way radix quicksort.

only if LRS is not long (!)

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

46

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
mobydic.txt	1.2 million	43 hours †	7.6 sec	79
chromosomell.txt	7.1 million	2 months †	61 sec	12,567
pi.txt	10 million	4 months †	84 sec	14

† estimated

47

Suffix sorting: worst-case input

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

Longest repeated substring very long.

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

```
abcdefgh
abcde fgh abcde fgh
bcde fgh
bcde fgh abcde fgh
cde fgh
cde fgh abcde fgh
de fgh
efgh abcde fgh
efgh
fgh abcde fgh
fgh
gh abcde fgh
fh
h abcde fgh
h
```

abcde fgh abcde fgh

algorithm	time to suffix sort (seconds)	
	mobydic.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 radix quicksort	2.8	400

† estimated

48

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.

49

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]

50

Linearithmic suffix sort example: phase 0

original suffixes

```

0  b 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
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
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
7  b c b a b a a a a a 0
8  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
11 b a a a a a 0
12 a a a a a 0
13 a a a a 0
14 a a a 0
15 a a 0
16 a 0
17 0
    
```

key-indexed counting sort (first character)

```

17 0
1  a 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
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
6  a b c b a b a a a a a 0
15 aa 0
14 aaa 0
13 aaaa 0
12 aaaaa 0
10 a b 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
9  b a b a a a a a 0
11 b a a a a a 0
7  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
8  c b a b a a a a a 0
    
```

↑
sorted

51

Linearithmic suffix sort example: phase 1

original suffixes

```

0  b 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
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
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
7  b c b a b a a a a a 0
8  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
11 b a a a a a 0
12 a a a a a 0
13 a a a a 0
14 a a a 0
15 a a 0
16 a 0
17 0
    
```

index sort (first two characters)

```

17 0
16 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
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
13 a a a a 0
15 aa 0
14 aaa 0
6  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
10 a b 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
9  b a b a a a a a 0
11 b a a a a a 0
2  b 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
8  c b a b a a a a a 0
    
```

↑
sorted

52

Linearithmic suffix sort example: phase 2

original suffixes

```

0  b 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
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
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
7  b c b a b a a a a a 0
8  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
11 b a a a a a 0
12 a a a a a 0
13 a a a a 0
14 a a a 0
15 a a 0
16 a 0
17 0
                
```

index sort (first four characters)

```

17  0
16  a 0
15  aa 0
14  aaa 0
3   aaaa b c b a b a a a a a 0
12  aaaaa 0
13  aaaa 0
4   aaab c b a b a a a a a 0
5   aabc b a b a a a a a 0
1   abaaa a b c b a b a a a a a 0
10  abaaaa 0
6   abcb a b a a a a a 0
2   baaa a b c b a b a a a a 0 a 0
11  baaaa a 0
0   baba a a a b c b a b a a a a a 0
9   baba a a a a 0
7   bcb a b a a a a a 0
8   cb a b a a a a a 0
                
```

↑
sorted

53

Linearithmic suffix sort example: phase 3

original suffixes

```

0  b 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
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
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
7  b c b a b a a a a a 0
8  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
11 b a a a a a 0
12 a a a a a 0
13 a a a a 0
14 a a a 0
15 a a 0
16 a 0
17 0
                
```

index sort (first eight characters)

```

17  0
16  a 0
15  aa 0
14  aaa 0
13  aaaa 0
12  aaaaa 0
3   aaaa b c b a b a a a a a 0
4   aaab c b a b a a a a a 0
5   aabc b a b a a a a a 0
10  abaaaa 0
1   abaaaa b c b a b a a a a a 0
6   abcb a b a a a a a 0
11  baaaa a 0
11  baaaa a 0
2   baaa a b c b a b a a a a 0 a 0
9   baba a a a a 0
0   baba a a a b c b a b a a a a a 0
7   bcb a b a a a a a 0
8   cb a b a a a a a 0
                
```

↑
sorted

FINISHED! (no equal keys)

54

Achieve constant-time string compare by indexing into inverse

original suffixes

```

0  b 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
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
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
7  b c b a b a a a a a 0
8  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
11 b a a a a a 0
12 a a a a a 0
13 a a a a 0
14 a a a 0
15 a a 0
16 a 0
17 0
                
```

index sort (first four characters)

```

17  0
16  a 0
15  aa 0
14  aaa 0
3   aaaa b c b a b a a a a a 0
12  aaaaa 0
13  aaaa 0
4   aaab c b a b a a a a a 0
5   aabc b a b a a a a a 0
1   abaaa a b c b a b a a a a a 0
10  abaaaa 0
6   abcb a b a a a a a 0
2   baaa a b c b a b a a a a 0 a 0
11  baaaa a 0
0   baba a a a b c b a b a a a a a 0
9   baba a a a a 0
7   bcb a b a a a a a 0
8   cb a b a a a a a 0
                
```

inverse

```

0  14
1  9
2  12
3  4
4  7
5  8
6  11
7  16
8  17
9  15
10 10
11 13
12 5
13 6
14 3
15 2
16 1
17 0
                
```

0 + 4 = 4

9 + 4 = 13

$\text{suffixes}_s[13] \leq \text{suffixes}_s[4]$ (because $\text{inverse}[13] < \text{inverse}[4]$)
 so $\text{suffixes}_s[9] \leq \text{suffixes}_s[0]$

55

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 radix quicksort	2,8	400
Manber MSD	17	8,5

† estimated

56

Radix sort summary

We can develop linear-time sorts.

- Compares not necessary for some types of keys.
- Use keys 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.

No algorithm can asymptotically examine fewer chars than 3-way radix quicksort.

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

Long strings are rarely random in practice.

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