Radix Sorts

- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- application: LRS

References:
  Algorithms in Java, Chapter 10
  http://www.cs.princeton.edu/introalgsds/61sort

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	operations on keys
insertion sort	$N^2/2$	$N^2/4$	no	`compareTo()`
selection sort	$N^2/2$	$N^2/2$	no	`compareTo()`
mergesort	N lg N	N lg N	N	`compareTo()`
quicksort	1.39 N lg N	1.39 N lg N	c lg N	`compareTo()`

lower bound: N lg N -1.44 N compares are required by any algorithm

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

Digital keys

Many commonly-use key types are inherently digital (sequences of fixed-length characters)

Examples

- Strings
- 64-bit integers

example interface

```
interface Digital
{
  public int charAt(int k);
  public int length(int);
}
```

This lecture:

- refer to fixed-length vs. variable-length strings
- **R** different characters for some fixed value **R**.
- assume key type implements `charAt()` and `length()` methods
- code works for `String`

Widely used in practice

- low-level bit-based sorts
- string sorts

- **key-indexed counting**
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- application: LRS

Key-indexed counting: assumptions about keys

Assume that keys are integers between **`0`** and **`R-1`**
Implication: Can use key as an array index

Examples:

- **`char`** (**`R = 256`**)
- **`short`** with fixed **`R`**, enforced by client
- **`int`** with fixed **`R`**, enforced by client

Reminder: equal keys are not uncommon in sort applications

Applications:

- sort phone numbers by area code
- sort classlist by precept
- Requirement: sort must be stable
- Ex: Full sort on primary key, then stable radix sort on secondary key

Key-indexed counting

Task: sort an array `a[]` of N integers between 0 and R-1

Plan: produce sorted result in array `temp[]`

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates
3. Access cumulates using key as index to find record positions.
4. Copy back into original array

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

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

for (int k = 1; k < 256; k++)        // compute cumulates
   count[k] += count[k-1];

for (int i = 0; i < N; i++)          // move records
   temp[count[a[i]++]] = a[i]

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

a[]

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

count[]

a	2
b	5
c	6
d	8
e	9
f	12

temp[]

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

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	operations on keys
insertion sort	$N^2/2$	$N^2/4$	no	`compareTo()`
selection sort	$N^2/2$	$N^2/2$	no	`compareTo()`
mergesort	N lg N	N lg N	N	`compareTo()`
quicksort	1.39 N lg N	1.39 N lg N	c lg N	`compareTo()`
key-indexed counting	N + R	N + R	N + R	use as array index

inplace version is possible and practical

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

A: Yes, if we do not depend on comparisons

- key-indexed counting
- **LSD radix sort**
- MSD radix sort
- 3-way radix quicksort
- application: LRS

Least-significant-digit-first radix sort

LSD radix sort.

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

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

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

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

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

sort must be stable
arrows do not cross

LSD radix sort: Why does it work?

Pf 1. [thinking about the past]

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

Pf 2. [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.

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

sort key

in order by previous passes

LSD radix sort implementation

Use k-indexed counting on characters, moving right to left

```
public static void lsd(String[] a)
{
   int N = a.length;
   int W = a[0].length;
   for (int d = W-1; d >= 0; d--)
   {
      int[] count = new int[R];
      for (int i = 0; i < N; i++)
         count[a[i].charAt(d) + 1]++;
      for (int k = 1; k < 256; k++)
         count[k] += count[k-1];
      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];
   }
}
```

key-indexed counting

count frequencies

compute cumulates

move records

copy back

Assumes fixed-length keys (length = W)

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

algorithm	guarantee	average	extra space	assumptions on keys
insertion sort	$N^2 /2$	$N^2 /4$	no	Comparable
selection sort	$N^2 /2$	$N^2 /2$	no	Comparable
mergesort	N lg N	N lg N	N	Comparable
quicksort	1.39 N lg N	1.39 N lg N	c lg N	Comparable
LSD radix sort	WN	WN	N + R	digital

Sorting Challenge

Problem: sort a huge commercial database on a fixed-length key field

Ex: account number, date, SS number

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort

	B14-99-8765		
	756-12-AD46		
	CX6-92-0112		
	332-WX-9877		
	375-99-QWAX		
	CV2-59-0221		
	[illegible]7-SS-032[illegible]		
	KJ-[illegible]388		
	715-YT-013C		
	MJ0-PP-983F		
	908-KK-33TY		
	BBN-63-23RE		
	48G-BM-912D		
	982-ER-9P1B		
	WBL-37-PB81		
	810-F4-J87Q		
	LE9-N8-XX76		
	908-KK-33TY		
	B14-99-8765		
	CX6-92-0112		
	CV2-59-0221		
	332-WX-23SQ		
	332-6A-9877		

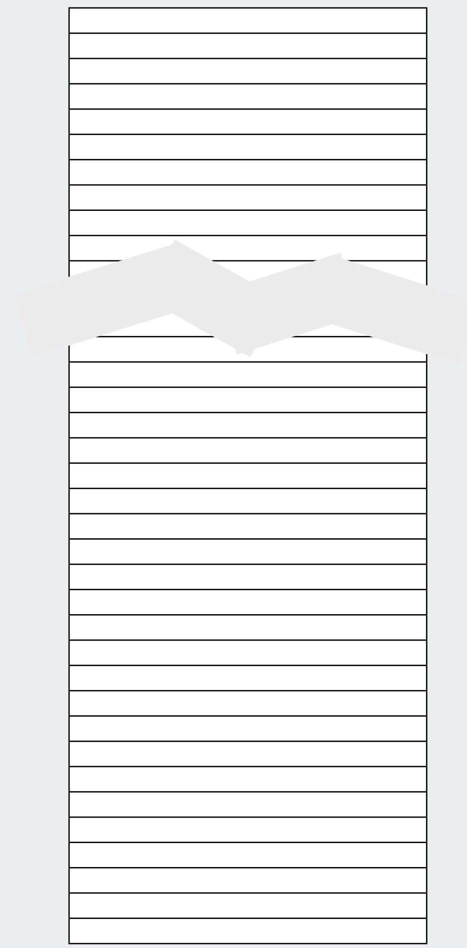
Sorting Challenge

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort



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

card punch

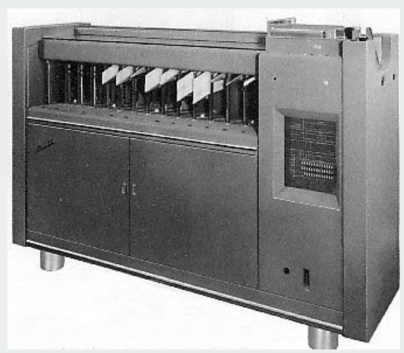
punched cards

card reader

mainframe

line printer

To sort a card deck
1. start on right column
2. put cards into hopper
3. machine distributes into bins
4. pick up cards (stable)
5. move left one column
6. continue until sorted



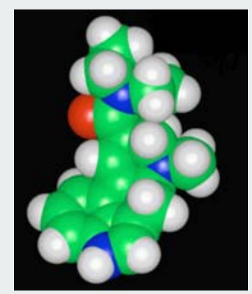
card sorter

LSD radix sort actually predates computers

LSD not related to sorting

↓

"Lucy in the Sky with Diamonds"



Lysergic Acid Diethylamide

- key-indexed counting
- LSD radix sort
- **MSD radix sort**
- 3-way radix quicksort
- application: LRS

MSD Radix Sort

Most-significant-digit-first 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 files 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

	sort key		
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

count[]

a	0
b	2
c	5
d	6
e	8
f	9

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 these independently (recursive)

MSD radix sort implementation

Use key-indexed counting on first character, recursively sort subfiles

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

private static void msd(String[] a, int lo, int hi, int d)
{
   if (hi <= lo + 1) return;
   int[] count = new int[256+1];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   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];
   for (int i = 0; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

key-indexed counting →

← count frequencies

← compute cumulates

← move records

← copy back

MSD radix sort: potential for disastrous performance

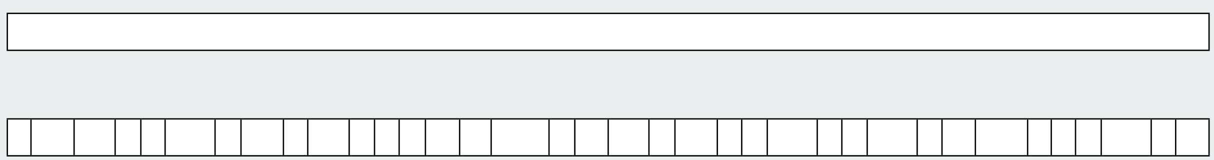
Observation 1: Much too slow for small files

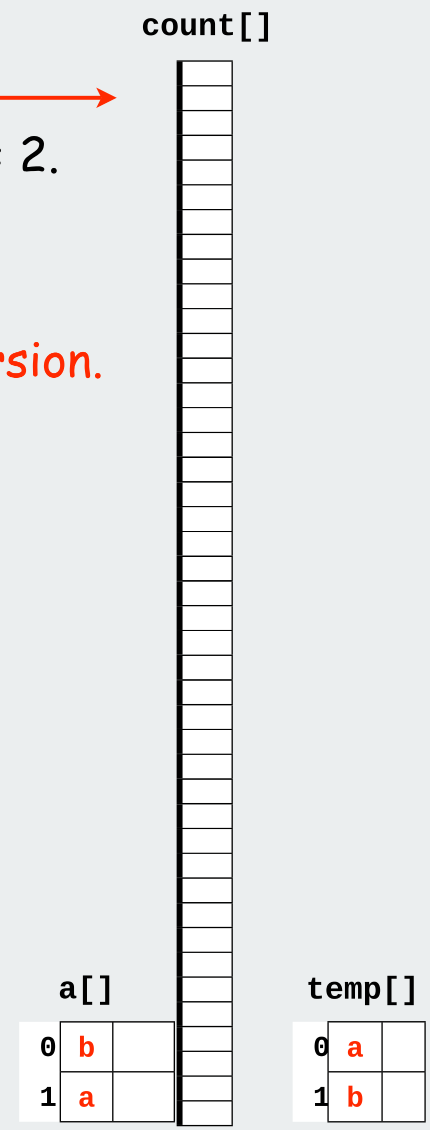
- all counts must be initialized to zero
- ASCII (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 30,000x slower for N = 2

Observation 2: Huge number of small files because of recursion.

- keys all different: up to N/2 files of size 2
- ASCII: 100x slower than copy pass for all N.
- Unicode: 30,000x slower for all N

switch to Unicode might be a big surprise!





Solution. Switch to insertion sort for small N.

MSD radix sort bonuses

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

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d

← 19/24 ≈ 80% of the characters examined

Bonus 2: Works for variable-length keys (**`String`** values)

0	a	c	e	t	o	n	e	\0		
1	a	d	d	i	t	i	o	n	\0	
2	b	a	d	g	e	\0				
3	b	e	d	a	z	z	l	e	d	\0
4	b	e	e	h	i	v	e	\0		
5	c	a	b	i	n	e	t	r	y	\0
6	d	a	b	b	l	e	\0			
7	d	a	d	\0						

← 19/64 ≈ 30% of the characters examined

Implication: sublinear sorts (!)

MSD string sort implementation

Use key-indexed counting on first character, recursively sort subfiles

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

private static void msd(String[] a, int l, int r, int d)
{
   if (r <= l + 1) return;
   int[] count = new int[256];
   for (int i = 0; i < N; i++)
      count[a[i].charAt(d) + 1]++;
   for (int k = 1; k < 256; k++)
      count[k] += count[k-1];
   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];
   for (int i = 1; i < 255; i++)
      msd(a, l + count[i], l + count[i+1], d+1);
}
```

key-indexed counting

don't sort strings that start with '\0' (end of string char)

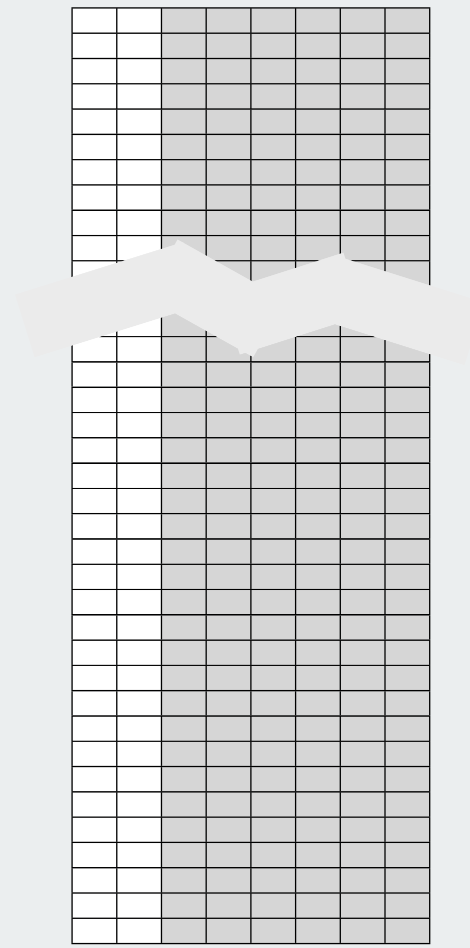
Sorting Challenge (revisited)

Problem: sort huge files of random 128-bit numbers

Ex: supercomputer sort, internet router

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort on MSDs ✓



2^{16} = 65536 counters

divide each word into 16-bit "chars"

sort on leading 32 bits in 2 passes

finish with insertion sort

examines only ~25% of the data

MSD radix sort versus quicksort for strings

Disadvantages of MSD radix sort.

- Accesses memory "randomly" (cache inefficient)
- Inner loop has a lot of instructions.
- Extra space for counters.
- Extra space for temp (or complicated inplace key-indexed counting).

Disadvantage of quicksort.

- N lg N, not linear.
- Has to rescan long keys for compares
- [but stay tuned]

- key-indexed counting
- LSD radix sort
- MSD radix sort
- **3-way radix quicksort**
- application: LRS

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

Idea. Do 3-way partitioning on the dth 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	d

partition 0th char on **b**

0	b	e	e
1	b	a	d
2	a	c	e
3	a	d	d
4	f	e	e
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	b	e	d

swap **b**'s to ends as in 3-way quicksort

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

3-way partition on **b**

0	a	d	d
1	a	c	e

`qsortX(0, 2, 0)`

2	b	a	d
3	b	e	e
4	b	e	d

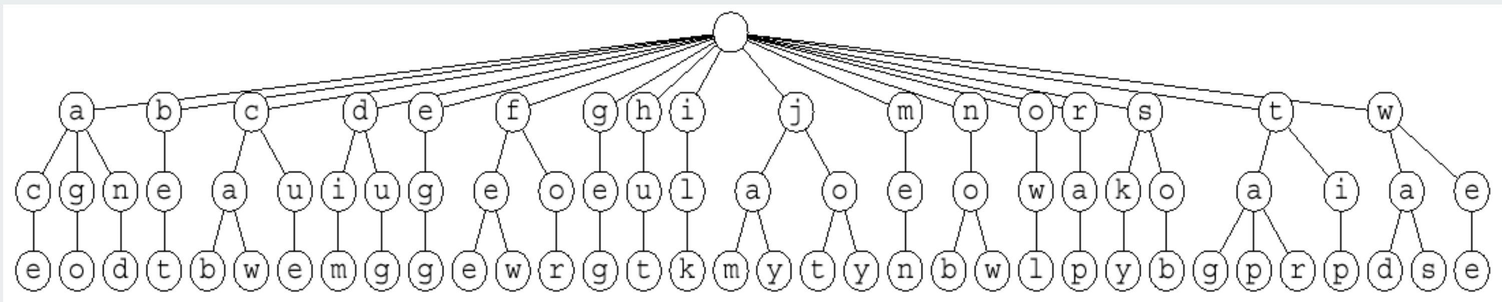
`qsortX(2, 5, 1)`

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

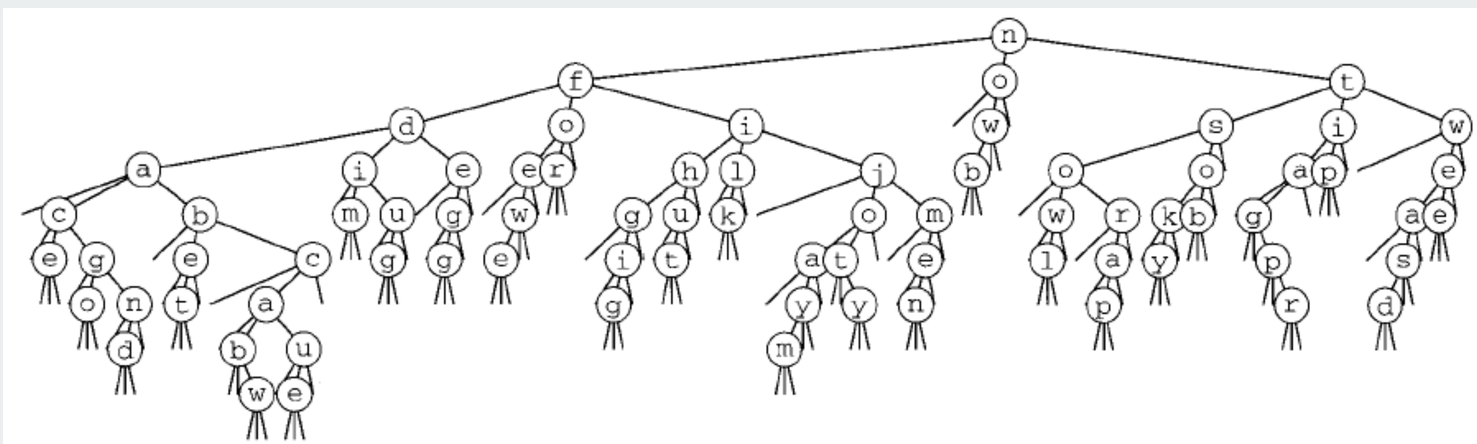
`qsortX(5, 12, 0)`

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)

3-Way radix quicksort

```
private static void quicksortX(String a[], int lo, int hi, int d)
{
   if (hi - lo <= 0) return;
   int i = lo-1, j = hi;
   int p = lo-1, q = hi;
   char v = a[hi].charAt(d);
   while (i < j)
   {
      while (a[++i].charAt(d) < v) if (i == hi) break;
      while (v < a[--j].charAt(d)) if (j == lo) break;
      if (i > j) break;
      exch(a, i, j);
      if (a[i].charAt(d) == v) exch(a, ++p, i);
      if (a[j].charAt(d) == v) exch(a, j, --q);
   }

   if (p == q)
   {
      if (v != '\0') quicksortX(a, lo, hi, d+1);
      return;
   }

   if (a[i].charAt(d) < v) i++;
   for (int k = lo; k <= p; k++) exch(a, k, j--);
   for (int k = hi; k >= q; k--) exch(a, k, i++);

   quicksortX(a, lo, j, d);
   if ((i == hi) && (a[i].charAt(d) == v)) i++;
   if (v != '\0') quicksortX(a, j+1, i-1, d+1);
   quicksortX(a, i, hi, d);
}
```

4-way partition with equals at ends

special case for all equals

swap equals back to middle

sort 3 pieces recursively

3-Way Radix quicksort vs. standard quicksort

standard quicksort.

- uses 2N ln N string comparisons on average.
- uses costly compares for long keys that differ only at the end, and this is a common case!

3-way radix quicksort.

- avoids re-comparing initial parts of the string.
- adapts to data: uses just "enough" characters to resolve order.
- uses 2 N ln N character comparisons on average for random strings.
- is sub-linear when strings are long

to within a constant factor

Theorem. Quicksort with 3-way partitioning is OPTIMAL.
No sorting algorithm can examine fewer chars on any input

asymptotically

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

3-Way Radix quicksort vs. MSD radix sort

MSD radix sort

- has a long inner loop
- is cache-inefficient
- repeatedly initializes counters for long stretches of equal chars, and this is a common case!

Ex. Library call numbers

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

3-way radix quicksort

- uses one compare for equal chars.
- is cache-friendly
- adapts to data: uses just "enough" characters to resolve order.

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

- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- **application: LRS**

Longest repeated substring

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
```

Longest repeated substring

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
```

String processing

String. Sequence of characters.

Important fundamental abstraction

Natural languages, Java programs, 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

String concatenation: append one string to end of another string.

Substring: extract a contiguous list 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 = s + t;              // u = "stringsring"
```

Implementing Strings In Java

Memory. 40 + 2N bytes for a virgin string!

could use byte array instead of String to save space

```
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);  }
   …
}
```

`java.lang.String`

String vs. StringBuilder

`String`. [immutable] Fast substring, slow concatenation.

`StringBuilder`. [mutable] Slow substring, fast (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

Warmup: longest common prefix

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

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

```
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);
}
```

linear time

Would be quadratic with **`StringBuilder`**

Lesson: cost depends on implementation

This lecture: need constant-time **`substring()`**, use **`String`**

Longest repeated substring

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

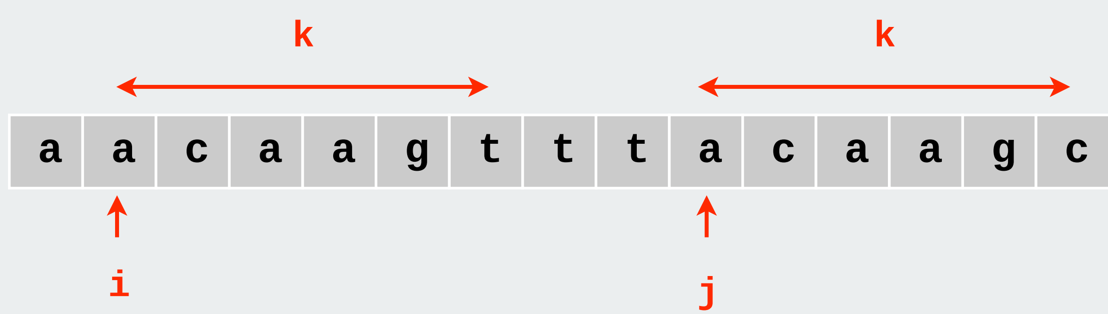
Classic string-processing problem.

Ex: `a a c a a g t t t a c a a g c`

`1` `9`

Applications
- bioinformatics.
- cryptanalysis.

Brute force.
- Try all indices `i` and `j` for start of possible match, and check.
- Time proportional to $M N^2$, where M is length of longest match.



Longest repeated substring

Suffix sort solution.

- form N suffixes of original string.
- sort to bring longest repeated substrings together.
- check LCP of adjacent substrings to find longest match

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														

sorted suffixes

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						

Suffix Sorting: Java Implementation

```
public class LRS {
   public static void main(String[] args) {
      String s = StdIn.readAll();
      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;
      }
      System.out.println(lrs);
   }
}
```

read input

create suffixes (linear time)

sort suffixes

find LCP

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

Sorting Challenge

Problem: suffix sort a long string

Ex. Moby Dick ~1.2 million chars

Which sorting method to use?

1. insertion sort
2. mergesort
3. quicksort
4. LSD radix sort
5. MSD radix sort
6. ✓ 3-way radix quicksort

only if LRS is not long (!)

Suffix sort experimental results

algorithm	time to suffix-sort Moby Dick (seconds)
brute-force	36.000 (est.)
quicksort	9.5
LSD	not fixed-length
MSD	395
MSD with cutoff	6.8
3-way radix quicksort	2.8

Suffix Sorting: Worst-case input

Longest match not long:

- hard to beat 3-way radix quicksort.

Longest match very long:

- radix sorts are quadratic in the length of the longest match
- Ex: two copies of Moby Dick.

Can we do better? linearithmic? linear?

Observation. Must find longest repeated substring while suffix sorting to beat N^2.

```
abcdefghi
abcdefghiabcdefghi
bcdefghi
bcdefghiabcdefghi
cdefghi
cdefghiabcdefgh
defghi
efghiabcdefghi
efghi
fghiabcdefghi
fghi
ghiabcdefghi
fhi
hiabcdefghi
hi
iabcdefghi
i
```

Input: "abcdeghiabcdefghi"

Fast suffix sorting

Manber's MSD algorithm

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

Running time

- finishes after lg N phases
- obvious upper bound on growth of total time: $O(N (\lg N)^2)$
- actual growth of total time (proof omitted): ~N lg N.

not many subfiles if not much repetition
3-way quicksort handles equal keys if repetition

Best algorithm in theory is linear (but more complicated to implement).

Linearithmic suffix sort example: phase 0

0	babaaaabcbabaaaaa0
1	abaaaabcbabaaaaa0
2	baaaabcbabaaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0
9	babaaaaa0
10	abaaaaa0
11	baaaaa0
12	aaaaa0
13	aaaa0
14	aaa0
15	aa0
16	a0
17	0

index sort

17	0
1	abaaaabcbabaaaaa0
16	a0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
15	aa0
14	aaa0
13	aaaa0
12	aaaaa0
10	abaaaaa0
0	babaaaabcbabaaaaa0
9	babaaaaa0
11	baaaaa0
7	bcbabaaaaa0
2	baaaabcbabaaaaa0
8	cbabaaaaa0

sorted

inverse

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

Linearithmic suffix sort example: phase 1

0	babaaaabcbabaaaaa0
1	abaaaabcbabaaaaa0
2	baaaabcbabaaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
6	abcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0
9	babaaaaa0
10	abaaaaa0
11	baaaaa0
12	aaaaa0
13	aaaa0
14	aaa0
15	aa0
16	a0
17	0

index sort

17	0
16	a0
12	aaaaa0
3	aaaabcbabaaaaa0
4	aaabcbabaaaaa0
5	aabcbabaaaaa0
13	aaaa0
15	aa0
14	aaa0
6	abcbabaaaaa0
1	abaaaabcbabaaaaa0
10	abaaaaa0
0	babaaaabcbabaaaaa0
9	babaaaaa0
11	baaaaa0
2	baaaabcbabaaaaa0
7	bcbabaaaaa0
8	cbabaaaaa0

sorted

inverse

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

Linearithmic suffix sort example: phase 2

		index sort		inverse	
0	babaaaabcbabaaaaa0	17	0	0	14
1	abaaaabcbabaaaaa0	16	a0	1	9
2	baaaabcbabaaaaa0	15	aa0	2	12
3	aaaabcbabaaaaa0	14	aaa0	3	4
4	aaabcbabaaaaa0	3	aaaabcbabaaaaa0	4	7
5	aabcbabaaaaa0	12	aaaaa0	5	8
6	abcbabaaaaa0	13	aaaa0	6	11
7	bcbabaaaaa0	4	aaabcbabaaaaa0	7	16
8	cbabaaaaa0	5	aabcbabaaaaa0	8	17
9	babaaaaa0	1	abaaaabcbabaaaaa0	9	15
10	abaaaaa0	10	abaaaaa0	10	10
11	baaaaa0	6	abcbabaaaaa0	11	13
12	aaaaa0	2	baaaabcbabaaaaa0	12	5
13	aaaa0	11	baaaaa0	13	6
14	aaa0	0	babaaaabcbabaaaaa0	14	3
15	aa0	9	babaaaaa0	15	2
16	a0	7	bcbabaaaaa0	16	1
17	0	8	cbabaaaaa0	17	0

sorted

Linearithmic suffix sort example: phase 3

			index sort			inverse	
0	babaaaabcbabaaaaa0		17	0		0	15
1	abaaaabcbabaaaaa0		16	a0		1	10
2	baaaabcbabaaaaa0		15	aa0		2	13
3	aaaabcbabaaaaa0		14	aaa0		3	4
4	aaabcbabaaaaa0		3	aaaabcbabaaaaa0		4	7
5	aabcbabaaaaa0		13	aaaa0		5	8
6	abcbabaaaaa0		12	aaaaa0		6	11
7	bcbabaaaaa0		4	aaabcbabaaaaa0		7	16
8	cbabaaaaa0		5	aabcbabaaaaa0		8	17
9	babaaaaa0		10	abaaaaa0		9	14
10	abaaaaa0		1	abaaaabcbabaaaaa0		10	9
11	baaaaa0		6	abcbabaaaaa0		11	12
12	aaaaa0		11	baaaaa0		12	6
13	aaaa0		2	baaaabcbabaaaaa0		13	5
14	aaa0		9	babaaaaa0		14	3
15	aa0		0	babaaaabcbabaaaaa0		15	2
16	a0		7	bcbabaaaaa0		16	1
17	0		8	cbabaaaaa0		17	0

sorted

FINISHED! (no equal keys)

Linearithmic suffix sort: key idea

Achieve constant-time string compare by indexing into inverse

		index sort		inverse	
0	babaaaabcbabaaaaa0	17	0	0	14
1	abaaaabcbabaaaaa0	16	a0	1	9
2	baaaabcbabaaaaa0	15	aa0	2	12
3	aaaabcbabaaaaa0	14	aaa0	3	4
4	aaabcbabaaaaa0	3	aaaabcbabaaaaa0	4	7
5	aabcbabaaaaa0	12	aaaaa0	5	8
6	abcbabaaaaa0	13	aaaa0	6	11
7	bcbabaaaaa0	4	aaabcbabaaaaa0	7	16
8	cbabaaaaa0	5	aabcbabaaaaa0	8	17
9	babaaaaa0	1	abaaaabcbabaaaaa0	9	15
10	abaaaaa0	10	abaaaaa0	10	10
11	baaaaa0	6	abcbabaaaaa0	11	13
12	aaaaa0	2	baaaabcbabaaaaa0	12	5
13	aaaa0	11	baaaaa0	13	6
14	aaa0	0	babaaaabcbabaaaaa0	14	3
15	aa0	9	babaaaaa0	15	2
16	a0	7	bcbabaaaaa0	16	1
17	0	8	cbabaaaaa0	17	0

0 + 4 = 4

9 + 4 = 13

13 < 4 (because 6 < 7) so 9 < 0

Suffix sort experimental results

algorithm	time to suffix-sort Moby Dick (seconds)	time to suffix-sort AesopAesop (seconds)
brute-force	36.000 (est.)	4000 (est.)
quicksort	9.5	167
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way radix quicksort	2.8	400
Manber MSD	17	8.5

counters in deep recursion

only 2 keys in subfiles with long matches

Radix sort summary

We can develop linear-time sorts.

- comparisons not necessary for some types of keys
- use keys to index an array

We can develop sub-linear-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 examine fewer bits than 3-way radix quicksort

- 1.39 N lg N bits for random data

Long strings are rarely random in practice.

- goal is often to learn the structure!
- may need specialized algorithms

lecture acronym cheatsheet	
LSD	least significant digit
MSD	most significant digit
LCP	longest common prefix
LRS	longest repeated substring