Radix Sorts

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

References: Algorithms in Java, Chapter 10
Intro to Algs and Data Structs, Section 6.1

Digital keys

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

Examples
- Strings
- 64-bit integers

This lecture:
- refer to fixed-length vs. variable-length strings
- R different characters
- key type implements charAt() and length() methods
- code works for String and key types that implement Digital.

Widely used in practice
- low-level bit-based sorts
- string sorts

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>average</th>
<th>extra space</th>
<th>assumptions on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>no</td>
<td>Comparable</td>
</tr>
<tr>
<td>selection sort</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>no</td>
<td>Comparable</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>Comparable</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 N \lg N$</td>
<td>$1.39 N \lg N$</td>
<td>$c \lg N$</td>
<td>Comparable</td>
</tr>
</tbody>
</table>

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

Q: Can we do better (despite the lower bound)?
Key-indexed counting: assumptions about keys

Assume that keys are integers between 0 and R-1

Examples:
- char (R = 256)
- short with fixed R, enforced by client
- int with fixed R, enforced by client

Reminder: equal keys are allowed in sorts

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

Implication: Can use key as an array index

Key-indexed counting

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

Plan: produce sorted result in array \( a[] \)

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates which specify destinations

Examples:

```
int N = a.length;
int[] count = new int[R];
for (int i = 0; i < N; i++)
  count[a[i]]++; 
```

```
for (int i = 1; i < 256; i++)
  count[i] = count[i-1] + count[i];
```

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

Key-indexed counting

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

Plan: produce sorted result in array \( a[] \)

1. Count frequencies of each letter using key as index
2. Compute frequency cumulates which specify destinations
3. Access cumulates using key as index to move records.
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

Frequency of execution of instructions in the inner loop:

- **LSD radix sort**
- **MSD radix sort**
- 3-way radix quicksort

Extra space assumptions on keys:

- Comparable
- No

Review: summary of the performance of sorting algorithms

Frequency of execution of instructions in the inner loop:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Average</th>
<th>Extra Space</th>
<th>Assumptions on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>no</td>
<td>Comparable</td>
</tr>
<tr>
<td>selection sort</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>no</td>
<td>Comparable</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>Comparable</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 N \log N$</td>
<td>$1.39 N \log N$</td>
<td>$c \log N$</td>
<td>Comparable</td>
</tr>
<tr>
<td>key-indexed counting</td>
<td>$N$</td>
<td>$N$</td>
<td>$N + R$</td>
<td>integers</td>
</tr>
</tbody>
</table>

Q: Can we do better (despite the lower bound)?
A: Yes, if we do not depend on comparisons
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.

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[W];
        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];
    }
}

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 | \(N^2/2\) \(N^2/4\) \(\text{no}\) Comparable | | |
| selection sort | \(N^2/2\) \(N^2/4\) \(\text{no}\) 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
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

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

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

2^{16} = 65536 counters
divide each word into 16-bit “chars”
sort in 8 passes

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

Lysergic Acid Diethylamide

"Lucy in the Sky with Diamonds"

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

LSD radix sort actually predates computers
**MSD radix sort**

### 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)

### MSD radix sort implementation

```java
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[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 < 255; i++)
        msd(a, l + count[i], l + count[i+1], d+1);
}
```

### 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 $N$

**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 a d
- 4 b a 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 (Strings)

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

19/64 = 30% of the characters examined

**Implication:** sublinear sorts (!)

---

**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

**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.**
- \( N \log N, \) not linear.
- Has to rescan long keys for compares
- [but stay tuned]

---

**MSD string sort implementation**

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

```java
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];
    temp = new char[N];
    msd(a, l + count[i], l + count[i+1], d+1);
  }
}
```

**MSD radix sort versus quicksort for strings**

Disadvantages of MSD radix sort.
- \( 2^{16} = 65536 \) 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
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

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

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

3-way radix quicksort

private static void quicksortX(String a[], int lo, int hi, int d)
{
   if (hi - lo <= 0) return;
   int i = ... radix quicksort
sort 3 pieces
recursively
special case for
all equals
4-way partition
with equals
at ends

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);
}
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 $2N \ln N$ character comparisons on average for random strings.
- is sub-linear when strings are long.

**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!

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

---

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

---

key-indexed counting

LSD radix sort
MSD radix sort
3-way radix quicksort

application: LRS

---

Ex. Library call numbers

```
WUS------10706------7---10
WUS-------12692-------4---27
WLSOC------2542-------3D
LTK------6015-P-63-1988
LDS-----361-H-4
```
Longest repeated substring

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

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

Using Strings in Java

String concatenation: append one string to end of another string.
Substring: extract a contiguous list of characters from a string.

```
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);
   }
}
```

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

Would be quadratic with StringBuilder

Lesson: cost depends on implementation

This lecture: need constant-time substring(), use String
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);
    }
}

% 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

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? \( \Theta(N \log N) \)? \( \Theta(N) \)?

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

Input: "abcdefghiabedefghi"

Suffix Sorting: Worst Case Input

Input: "abcdefghiabedefghi
  bcdefghi
  bdefghiabedefghi
cdefghi
defghiabedefghi
efghi
efghiaabedefghi
fghi
gbiabedefghi
ghiabedefghi
hiabedefghi
iaabedefghi"
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 in linear time.
- finishes after $\lg N$ phases.
- total time proportional to $N \lg N$.

Best in theory. $O(N)$ but more complicated to implement.

Suffix sort experimental results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>time to suffix-sort Moby Dick (seconds)</th>
<th>time to suffix-sort AesopAesop (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36.000 (est.)</td>
<td>4000 (est.)</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td>MSD</td>
<td>out of memory</td>
<td></td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td>3-way radix quicksort</td>
<td>2.8</td>
<td>400</td>
</tr>
<tr>
<td>Manber MSD</td>
<td>17</td>
<td>8.5</td>
</tr>
</tbody>
</table>

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