5.1 STRING SORTS

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

String processing

String. Sequence of characters.

Important fundamental abstraction.
- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- ...

"The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology." — M. V. Olson

The char data type

C char data type. Typically an 8-bit integer.
- Supports 7-bit ASCII.
- Can represent only 256 characters.

Java char data type. A 16-bit unsigned integer.
- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).
I (heart) Unicode

The String data type

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

Length. Number of characters.
Indexing. Get the $i^{th}$ character.
Substring extraction. Get a contiguous subsequence of characters.
String concatenation. Append one character to end of another string.

The String data type: Java implementation

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

    public int length() {
        return length;
    }

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

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

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

The String data type: performance

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Guarantee</th>
<th>Extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

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

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


<table>
<thead>
<tr>
<th>operation</th>
<th>String</th>
<th>StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>length()</td>
<td>guarantee</td>
</tr>
<tr>
<td></td>
<td>charAt()</td>
<td>guarantee</td>
</tr>
<tr>
<td></td>
<td>substring()</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>concat()</td>
<td>N</td>
</tr>
</tbody>
</table>

* amortized

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

String vs. StringBuilder

Q. How to efficiently reverse a string?

A. public static String reverse(String s) {
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}

B. public static String reverse(String s) {
    StringBuilder rev = new StringBuilder(s);
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}

String challenge: array of suffixes

Q. 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;
}

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;
}
5.1 STRING SORTS

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

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

Alphabets

Digital key. Sequence of digits over fixed alphabet.
Radix. Number of digits $R$ in alphabet.

<table>
<thead>
<tr>
<th>name</th>
<th>$R()$</th>
<th>$\lg R()$</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>0023456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0023456789ABCDEF</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>2</td>
<td>ACTG</td>
</tr>
<tr>
<td>LOWERCASE</td>
<td>26</td>
<td>5</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>BASE64</td>
<td>64</td>
<td>6</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ascii characters</td>
</tr>
<tr>
<td>EXTENDED_ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 N \lg N$</td>
<td>$2 N \lg N$</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Lower bound. $\sim N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?
A. Yes, if we don’t depend on key compares.
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. [stay tuned]

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

### Key-indexed counting demo

**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 items.
- Copy back into original array.

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Key-indexed counting demo

**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 items.
- Copy back into original array.

```java
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];
```

```
0  a
1  b
2  c
3  d
4  e
5  f
6  g
7  h
8  i
9  j
10 k
11 l
```

Key-indexed counting: analysis

**Proposition.** Key-indexed counting uses \( \sim N + 4R \) array accesses to sort `N` items whose keys are integers between `0` and `R - 1`.

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

5.1 STRING SORTS

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
Least-significant-digit-first string sort

LSD string (radix) sort.

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

**LSD string sort: correctness proof**

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

**Pf.** [by induction on $i$]

After pass $i$, strings are sorted by last $i$ characters.

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

**Proposition.** LSD sort is stable.

**Summary of the performance of sorting algorithms**

**Frequency of operations.**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{4} N^2$</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2N \log N$</td>
<td>$2N \log N$</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

LSD †

2 W N 2 W N $N + R$ yes charAt()

* probabilistic
† fixed-length W keys

Q. What if strings do not have same length?
String sorting interview question


Which sorting method to use?

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

String sorting interview question

Google CEO Eric Schmidt interviews Barack Obama

1890 Census. Finished months early and under budget

Herman Hollerith. Developed counting and sorting machine to automate.

- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time into one of 12 bins.
- Typical question: how many women are between 20 and 30?

1880 Census. Took 1,500 people 7 years to manually process data.

Hollerith's company later merged with 3 others to form Computing.

Tabulating Recording Corporation (CTR); company renamed in 1924.

Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.
- Hollerith tabulating machine and sorter

How to take a census in 1900s?

- Took 1,500 people 7 years to manually process data.
- Developed counting and sorting machine to automate.
- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time into one of 12 bins.
- Typical question: how many women are between 20 and 30?

How to get rich sorting in 1900s?

- Hollerith's company later merged with 3 others to form Computing.
- Tabulating Recording Corporation (CTR); company renamed in 1924.

- Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

- Hollerith tabulating machine and sorter
Most-significant-digit-first string sort

**MSD string (radix) sort.**
- Partition array into $R$ pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).

```
0  d  a  b
1  b  a  c
d  a  b  a
c  a  b  e
d  e  d
f  e  d

0  a  d  d
1  a  c  e
2  b  e  e
3  b  a  d
c  a  e
5  d  a  d
6  b  d
d  a  d
e  b  e
f  d

count[]

0  a  d  d
1  a  c  e
2  b  e  e
3  b  a  d
c  a  e
5  d  a  d
6  b  d
d  a  d
e  b  e
f  d

0  a  d  d
e  b  e
f  d

0  a  d  d
e  b  e
f  d

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

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

Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

5.1 STRING SORTS

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

Treat strings as if they had an extra char at end (smaller than any char).

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

MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.
- Each function call needs its own count[] array.
- ASCII (256 counts): 100x slower than copy pass for $N=2$.
- Unicode (65,536 counts): 32,000x slower for $N=2$.

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

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.
- Insertion sort, but start at $d^h$ character.
- Implement less() so that it compares starting at $d^h$ character.
MSD string sort: performance

Number of characters examined.
- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!

compareTo() based sorts can also be sublinear!

<table>
<thead>
<tr>
<th>Characters examined by MSD string sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E10402</td>
</tr>
<tr>
<td>1HYL490</td>
</tr>
<tr>
<td>1ROZ572</td>
</tr>
<tr>
<td>2HKE734</td>
</tr>
<tr>
<td>2YE230</td>
</tr>
<tr>
<td>2XOR846</td>
</tr>
<tr>
<td>3CB573</td>
</tr>
<tr>
<td>3CV7720</td>
</tr>
<tr>
<td>3GJ3339</td>
</tr>
<tr>
<td>3KNA382</td>
</tr>
<tr>
<td>3TAV879</td>
</tr>
<tr>
<td>4CQP781</td>
</tr>
<tr>
<td>4QG284</td>
</tr>
<tr>
<td>4VW229</td>
</tr>
</tbody>
</table>

MSD string sort vs. quicksort for strings

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

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

Goal. Combine advantages of MSD and quicksort.

Summary of the performance of sorting algorithms

Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( \frac{1}{4} N^2 )</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
<td>yes</td>
<td>compareTo()</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>( 2 N \lg N )</td>
<td>( 2 N \lg N )</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD ( \dagger )</td>
<td>2 N W</td>
<td>2 N W</td>
<td>N + R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ( \dagger )</td>
<td>2 N W</td>
<td>N \log_a N</td>
<td>N + D R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

\( \dagger \) probabilistic
\( \dagger \) fixed-length W keys
\( \dagger \) average-length W keys

5.1 STRING SORTS

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the \(d^{th}\) character.
- Less overhead than \(K\)-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char (but does re-examine characters not equal to the partitioning char).

3-way string quicksort: Java implementation

```java
private static void sort(String[] a)
{    sort(a, 0, a.length - 1, 0); }

private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = charAt(a[lo], d);
    int i = lo + 1;
    while (i <= gt)
    {
        int t = charAt(a[i], d);
        if (t < v) exch(a, lt++, i++);
        else if (t > v) exch(a, i, gt--);
        else i++;
    }
    sort(a, lo, lt-1, d);
    if (v >= 0) sort(a, lt, gt, d+1);
    sort(a, gt+1, hi, d);
}
```

3-way string quicksort: trace of recursive calls

Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

3-way string quicksort vs. standard quicksort

Standard quicksort.
- Uses \(\sim 2N \ln N\) string compares on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.
- Uses \(\sim 2N \ln N\) character compares on average for random strings.
- Avoids re-comparing long common prefixes.

Fist Algorithms for Sorting and Searching Strings

Joe L. Bentley* Robert Sedgewick$

Abstract

We present alternatives algorithms for sorting and searching multibyte data, and derive from them practical C implementations for applications that match keys structure of strings. The string algorithm blends Quicksort and radix sort, is competitive with the best known C implementations, and supports more advanced searches.
3-way string quicksort vs. MSD string sort

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

3-way string quicksort.
- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

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

Summary of the performance of sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>½ N^2</td>
<td>¾ N^2</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Merge sort</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Heapsort</td>
<td>2 N lg N</td>
<td>2 N lg N</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>2 N W</td>
<td>2 N W</td>
<td>N + R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>2 N W</td>
<td>N log a N</td>
<td>N + D R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>1.39 W N lg N</td>
<td>1.39 N lg N</td>
<td>log N + W</td>
<td>no</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

Keyword-in-context search

Given a text of \( N \) characters, preprocess it to enable fast substring search (find all occurrences of query string context).

```plaintext
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
```

Applications. Linguistics, databases, web search, word processing, ....
Keyword-in-context search

Given a text of N characters, preprocess it to enable fast substring search (find all occurrences of query string context).

Keyword-in-context search: suffix-sorting solution

- Preprocess: suffix sort the text.
- Query: binary search for query; scan until mismatch.

Applications. Linguistics, databases, web search, word processing, ....

Suffix sort

input string

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>it was best it was w</td>
<td>t w a s b e s t i t w a s w</td>
<td>w a s b e s t i t w a s w</td>
<td>a s b e s t i t w a s w</td>
<td>s b e s t i t w a s w</td>
<td>b e s t i t w a s w</td>
<td>e s t i t w a s w</td>
<td>s i t w a s w</td>
<td>t i t w a s w</td>
<td>i t w a s w</td>
<td>t w a s w</td>
<td>a s w</td>
<td>w s w</td>
<td>w</td>
<td></td>
</tr>
</tbody>
</table>

sort suffixes to bring repeated substrings together

Keyword-in-context search: suffix-sorting solution

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>
| sealed my letter and ... | search for contraband her unavailing search for your father and gone in search of her husband provinces in search of impoverished dispersing in search of other carriers 

better things

||
|---|---|---|---|---|---|---|---|
| 632698 | 713727 | 660598 | 67610 | 4430 | 42705 | 499797 | 182045 |

KWIC search for "search" in Tale of Two Cities

Longest repeated substring

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

Applications. Bioinformatics, cryptanalysis, data compression, ...
**Longest repeated substring: a musical application**

Visualize repetitions in music. [http://www.bewitched.com](http://www.bewitched.com)

**Longest repeated substring**

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.

```
\[
\begin{array}{c}
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{t} & \text{t} & \text{t} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{c} \\
\end{array}
\]
```

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

**Longest repeated substring: a sorting solution**

Longest repeated substring: a musical application

Mary Had a Little Lamb

Bach's Goldberg Variations

Longest repeated substring: a sorting solution

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

b c d e f g h i j k cb cc cd ce cf

Compute longest prefix between adjacent suffixes

```
\[
\begin{array}{c}
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{t} & \text{t} & \text{t} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{c} \\
\end{array}
\]
```

Arrays.sort(suffixes);

```
\[
\begin{array}{c}
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{t} & \text{t} & \text{t} & \text{a} & \text{c} & \text{a} & \text{a} & \text{g} & \text{c} \\
\end{array}
\]
```

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++)
        { 
            int len = lcp(suffixes[i], suffixes[i+1]);
            if (len > lrs.length())
                lrs = suffixes[i].substring(0, len);
        }
    return lrs;
}
```

% java LRS < moby dick.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.

$✓$ $E$ uses suffix sorting solution with 3-way string quicksort.

but only if LRS is not long ($\dagger$)

**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 $\dagger$ | 7.6 sec | 79
chromosome11.txt | 7.1 million | 2 months $\dagger$ | 61 sec | 12,567
pi.txt | 10 million | 4 months $\dagger$ | 84 sec | 14
pipi.txt | 20 million | forever $\dagger$ | ?? | 10 million

$\dagger$ estimated

Suffix sorting: worst-case input

**Bad input:** longest repeated substring very long.

- Ex: same letter repeated $N$ times.
- Ex: two copies of the same Java codebase.

<table>
<thead>
<tr>
<th>form suffixes</th>
<th>sorted suffixes</th>
</tr>
</thead>
</table>
| 0 twinstwins | 9 ins 
| 1 twinstwins | 8 ins 
| 2 twinstwins | 7 ins 
| 3 twinstwins | 6 ins 
| 4 twinstwins | 5 ins 
| 5 twinstwins | 4 twinstwins 
| 6 twinstwins | 3 twinstwins 
| 7 twinstwins | 2 twinstwins 
| 8 twinstwins | 1 twinstwins 
| 9 twinstwins | 0 twinstwins |

LRS needs at least $1 + 2 + 3 + \ldots + D$ character compares, where $D =$ length of longest match.

**Running time.** Quadratic (or worse) in $D$ for LRS (and also for sort).

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. $\rightarrow$ Manber-Myers algorithm
- Linear. $\rightarrow$ suffix trees (beyond our scope)
- Nobody knows.
Suffix sorting in linearithmic time

Manber-Myers MSD algorithm overview.
- 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.  \(N \lg N\).
- Finishes after \(\lg N\) phases.
- Can perform a phase in linear time. (!) [ahead]

---

Linearithmic suffix sort example: phase 0

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>key-indexed counting sort (first character)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b b a a a a b c b</td>
<td>b a a a a a b c b a a a a a 0</td>
</tr>
<tr>
<td>a b a a a a b c b</td>
<td>a b a a a a b c b a a a a a 0</td>
</tr>
<tr>
<td>b a a a c b b a a a</td>
<td>b a a a c b b a a a a a 0</td>
</tr>
<tr>
<td>a a b c a b b a a a</td>
<td>a a b c a b b a a a a a 0</td>
</tr>
<tr>
<td>c a b b b b a a a</td>
<td>c a b b b b a a a a a 0</td>
</tr>
<tr>
<td>b a a a a a b c</td>
<td>b a a a a a b c a a a a a 0</td>
</tr>
<tr>
<td>a a a a a a b c</td>
<td>a a a a a a b c a a a a a 0</td>
</tr>
</tbody>
</table>

---

Linearithmic suffix sort example: phase 1

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first two characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b b a a a a b c b</td>
<td>b b a a a a b c b a a a a a 0</td>
</tr>
<tr>
<td>b a a a c b b a a a</td>
<td>b a a a c b b a a a a a 0</td>
</tr>
<tr>
<td>c b c b b a a a a</td>
<td>c b c b b a a a a a 0</td>
</tr>
<tr>
<td>b c b b a a a a</td>
<td>b c b b a a a a 0</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>a a a a a a a 0</td>
</tr>
<tr>
<td>b b b b a a a a</td>
<td>b b b b a a a a 0</td>
</tr>
<tr>
<td>a a a a a a</td>
<td>a a a a a a 0</td>
</tr>
<tr>
<td>a a a a</td>
<td>a a a a 0</td>
</tr>
</tbody>
</table>

---

Linearithmic suffix sort example: phase 2

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first four characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b b a a a a b c b</td>
<td>b b a a a a b c b a a a a a 0</td>
</tr>
<tr>
<td>b a a a c b b a a a</td>
<td>b a a a c b b a a a a a 0</td>
</tr>
<tr>
<td>c b c b b a a a a</td>
<td>c b c b b a a a a a 0</td>
</tr>
<tr>
<td>b c b b a a a a</td>
<td>b c b b a a a a 0</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>a a a a a a a 0</td>
</tr>
<tr>
<td>b b b b a a a a</td>
<td>b b b b a a a a 0</td>
</tr>
<tr>
<td>a a a a</td>
<td>a a a a 0</td>
</tr>
<tr>
<td>a a a</td>
<td>a a a 0</td>
</tr>
<tr>
<td>a a</td>
<td>a a 0</td>
</tr>
<tr>
<td>a</td>
<td>a 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---
Linearithmic suffix sort example: phase 3

original suffixes
0 b b a a a a b c b a b a a a a 0
1 a b a a a a b c b a b a a a a 0
2 b a a a a a b c b a b a a a a 0
3 a a a a a a b c b a b a a a a 0
4 a a b c b a b a a a a a 0
5 a a a a b c b a b 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 0
14 a a 0
15 a 0
16 a
17 0

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

finished (no equal keys)

Constant-time string compare by indexing into inverse

original suffixes
0 b b a a a a b c b a b a a a a 0
1 a b a a a a b c b a b a a a a 0
2 b a a a a a b c b a b a a a a 0
3 a a a a a a b c b a b a a a a 0
4 a a b c b a b a a a a a 0
5 a a a a b c b a b 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 0
14 a a 0
15 a 0
16 a
17 0

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


String sorting summary

We can develop linear-time sorts.
- Key compares not necessary for string keys.
- Use characters as index in an array.

We can develop sublinear-time sorts.
- Input size is amount of data in keys (not number of keys).
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.
- 1.39 \( N \lg N \) chars for random data.

Long strings are rarely random in practice.
- Goal is often to learn the structure!
- May need specialized algorithms.