Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
String sorts
- 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.
- Need more bits to represent certain characters.

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI
1 DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM GS RS US
2 SP ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 :)<=>?
3 @ A B C D E F G H I J K L M N O
4 P Q R S T U V W X Y Z [ \ ] ^ _
5 ` a b c d e f g h i j k l m n o
6 p q r s t u v w x y z { | } ~ DEL
```

**Hexadecimal to ASCII conversion table**

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

<table>
<thead>
<tr>
<th>A</th>
<th>á</th>
<th>ą</th>
<th>ő</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0041</td>
<td>U+00E1</td>
<td>U+2202</td>
<td>U+1D50A</td>
</tr>
</tbody>
</table>

**Unicode characters**
I (heart) Unicode
The String data type

String data type. Sequence of characters (immutable).

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

```
s.length()
s.charAt(3)
s.substring(7, 11)
```
public final class String implements Comparable<String> {
    private char[] val; // 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[] val) {
        this.offset = offset;
        this.length = length;
        this.val = val;
    }

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

    ...
The String data type: performance

String data type. Sequence of characters (immutable).

Design Choice. Immutable, cache or share the backing array

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**.

**StringBuilder data type.** Sequence of characters (mutable).

**Design Choice.** Easier to update, can’t cache or share array.

**Underlying implementation.** Resizing char[] array and length.

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

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

Actually as of Java 1.7 this is O(n) for String as well. Before 1.7 the initial String and substring shared the backing array (no need to copy!)
**String vs. StringBuilder**

**Q. How to efficiently reverse a string?**

**A.**

```java
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**
- String concatenation creates a new String and all chars in backing array are copied to new one.

**B.**

```java
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**
- The backing array is updated. Sometimes may need to expand the array but amortised cost is $O(1)$
Q. How to efficiently form array of suffixes?

input string

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>g</th>
<th>t</th>
<th>t</th>
<th>t</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>g</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

suffixes
String vs. StringBuilder

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;
}

Since Strings are immutable, the backing array of larger String can be shared with substring. In Java 1.7 they changed it, now cost is the same as below!

The array of StringBuilder can change, so can't share with substring.
Longest common prefix

Q. How long to compute length of longest common prefix?

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.

Complexity of some algorithms will depend on this.

<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>0123456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0123456789ABCDEF</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>ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/</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>
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
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>$N^2/2$</td>
<td>$N^2/4$</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>2 $N \log N$</td>
<td>2 $N \log N$</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

* probabilistic

Lower bound. $\sim N \log N$ 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 ⇒ can't just count up number of keys of each value.
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];
```
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];
```
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];
```

6 keys < $d$, 8 keys < $e$
so $d$'s go in $a[6]$ and $a[7]$.}

20
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];
```
**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];
```
**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];
```
**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];
```
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];
```
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];
```
**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];
```
**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];
```
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];
```
**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];
```
**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];
```
**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];
```
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];
```
**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];
```
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];
```
**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];
```
**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];
```

6 keys < d, 8 keys < e so d's go in a[6] and a[7]
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];
```
Goal. Sort an array \( \text{a[]} \) 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];
```
Key-indexed counting: analysis

**Proposition.** Key-indexed counting uses \( \sim 11N + 4R \) array accesses to sort \( N \) items whose keys are integers between 0 and \( R - 1 \).

**Proposition.** Key-indexed counting uses extra space proportional to \( N + R \).

Stable? ✔️

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Garcia</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Harris</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Johnson</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>Thomas</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>White</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aux[0]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>aux[1]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>aux[2]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>aux[3]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>aux[4]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>aux[5]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>aux[6]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>aux[7]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>aux[8]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>aux[9]</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>aux[10]</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>aux[11]</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>aux[12]</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>aux[13]</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>aux[14]</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>aux[15]</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>aux[16]</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>aux[17]</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>aux[18]</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>aux[19]</td>
<td></td>
</tr>
</tbody>
</table>

Depends on the Alphabet size / Max integer value
String Sorts

- 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).

<table>
<thead>
<tr>
<th>d=2</th>
<th>d=1</th>
<th>d=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>d=1</td>
<td>c=2</td>
<td>e=0</td>
</tr>
<tr>
<td>d=1</td>
<td>e=2</td>
<td>d=0</td>
</tr>
<tr>
<td>d=2</td>
<td>c=2</td>
<td>a=1</td>
</tr>
<tr>
<td>d=2</td>
<td>e=2</td>
<td>d=0</td>
</tr>
<tr>
<td>d=2</td>
<td>a=2</td>
<td>c=1</td>
</tr>
<tr>
<td>d=2</td>
<td>d=2</td>
<td>d=0</td>
</tr>
<tr>
<td>d=2</td>
<td>f=2</td>
<td>e=1</td>
</tr>
<tr>
<td>d=2</td>
<td>b=2</td>
<td>d=1</td>
</tr>
<tr>
<td>d=2</td>
<td>e=2</td>
<td>e=0</td>
</tr>
<tr>
<td>d=2</td>
<td>b=2</td>
<td>e=1</td>
</tr>
</tbody>
</table>

Sort must be stable (arrows do not cross)
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.

- [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.

<table>
<thead>
<tr>
<th></th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td>a</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>e</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td>f</td>
<td>e</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>e</td>
<td>e</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sort key

sorted from previous passes (by induction)
public class LSD
{
    public static void sort(String[] a, int W)
    {
        int R = 256;
        int N = a.length;
        String[] aux = 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++)
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
### 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>N² / 2</td>
<td>N² / 4</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 †</td>
<td>2 W N</td>
<td>2 W N</td>
<td>N + R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length W keys

**Q.** What if strings do not have same length?
Problem. Sort a huge commercial database on a fixed-length key.

Ex. Account number, date, Social Security number, ...

Which sorting method to use?

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

256 (or 65,536) counters;
Fixed-length strings sort in $W$ passes.
String sorting challenge 2a

**Problem.** Sort one million 32-bit integers.

**Ex.** Google (or presidential) interview. Obama answered “Bubble Sort is not the way to go”

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

Google CEO Eric Schmidt interviews Barack Obama
String sorting challenge 2a

Problem. Sort one million 32-bit integers.

Can view 32-bit integers as:
- Strings of length \( W = 1 \) over alphabet of size \( R = 2^{32} \)
- Strings of length \( W = 2 \) over alphabet of size \( R = 2^{16} \)
- Strings of length \( W = 3 \) over alphabet of size \( R = 2^8 \)
  ...

- Each LSD sort out of \( W \) takes \( N + R \)
- If \( R = 2^{16} \) then we can ignore \( R \), and reduce to \( O(N) \)
Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.
Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.

✓ LSD string sort.

Divide each word into eight 16-bit “chars”

$2^{16} = 65,536$ counters.
Sort in 8 passes.
String sorting challenge 2b

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

Ex. Supercomputer sort, internet router.

Which sorting method to use?

✓ • Insertion sort.
   • Mergesort.
   • Quicksort.
   • Heapsort.
✓ • LSD string sort.

Divide each word into eight 16-bit “chars”

$2^{16} = 65,536$ counters

LSD sort on leading 32 bits in 2 passes

Finish with insertion sort

Examines only ~25% of the data
How to take a census in 1900s?

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

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 of age 20 to 30?

1890 Census. Finished months early and under budget!
How to get rich sorting in 1900s?

**Punch cards. [1900s to 1950s]**
- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

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

card sorter
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
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).
MSD string sort: example

Input:
- She sells seashells by the sea.
- The shells she sells are surely seashells.
- She sells seashells by the sea.
- The shells she sells are surely seashells.

Output:
- Are are by by sea seashells seashells seashells seashells seashells seashells seashells.
- Are are by by sea seashells seashells seashells seashells seashells seashells seashells.
- Are are by by sea seashells seashells seashells seashells seashells seashells seashells.
- Are are by by sea seashells seashells seashells seashells seashells seashells seashells.

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted).
Variable-length strings

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

```
private static int charAt(String s, int d) {
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

C strings. Have extra char `'\0'` at end ⇒ no extra work needed.

why smaller?

she before shells

0  sea -1
1  seas shells l s -1
2  sell s -1
3  she -1
4  she -1
5  shell ls -1
6  shore e -1
7  surely y -1
public static void sort(String[] a)
{
    aux = new String[a.length];
sort(a, aux, 0, a.length, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int[] count = new int[R+2];
    for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 2]++;
    for (int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]++] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

    for (int r = 0; r < R; r++)
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
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 \(d^{th}\) character.
- Implement `less()` so that it compares starting at \(d^{th}\) character.

```java
public static void sort(String[] a, int lo, int hi, int d)
{
    for (int i = lo; i <= hi; i++)
        for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
            exch(a, j, j-1);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

In Java, forming and comparing substrings is faster than directly comparing chars with `charAt()`
**MSD string sort: performance**

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

- compareTo() based sorts can also be sublinear!

<table>
<thead>
<tr>
<th>Random (sublinear)</th>
<th>Non-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1EI0402</td>
<td>are</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1HYL490</td>
<td>by</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1ROZ572</td>
<td>sea</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2HXE734</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2IYE230</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2X0R846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CD8573</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CVP720</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3IGJ319</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3KNA382</td>
<td>shells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TAV879</td>
<td>shore</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4CQP781</td>
<td>surely</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4QGI284</td>
<td>the</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4YHV229</td>
<td>the</td>
<td>1DNB377</td>
</tr>
</tbody>
</table>

Characters examined by MSD string sort
## 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>$N^2 / 2$</td>
<td>$N^2 / 4$</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 †</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>$N \ log \ R \ N$</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)

* probabilistic
† fixed-length W keys
‡ average-length W keys
MSD string sort vs. quicksort for strings

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

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

Goal. Combine advantages of MSD and quicksort.
String Sorts

- 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 $R$-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).

- Partitioning item
- Use first character to partition into "less", "equal", and "greater" subarrays
- Recursively sort subarrays, excluding first character for middle subarray
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: 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 partitioning (using d\textsuperscript{th} character) to handle variable-length strings

sort 3 subarrays recursively
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.

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort, it is competitive with the best known C sort codes. The searching algorithm blends tries and binary tree representations that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, 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 the method of choice for sorting strings.
## 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>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>1</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</td>
<td><code>compareTo()</code></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><code>compareTo()</code></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><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD †</td>
<td>$2 , N , W$</td>
<td>$2 , N , W$</td>
<td>$N + R$</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD ‡</td>
<td>$2 , N , W$</td>
<td>$N \log R , N$</td>
<td>$N + D , R$</td>
<td>yes</td>
<td><code>charAt()</code></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><code>charAt()</code></td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length $W$ keys
‡ average-length $W$ keys
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
Keyword-in-context search

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

```
% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
dispersing in search of other carri
n that bed and search the straw hold

better thing
 t is a far far better thing that i do than
 some sense of better things else forgotte
 was capable of better things mr carton ent
```

**Applications.** Linguistics, databases, web search, word processing, ....
Suffix sort

**input string**

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**form suffixes**

|    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 0  | a  | a  | c  | a  | a  | g  | t  | t  | t  | t  |
| 1  | a  | c  | a  | a  | g  | t  | t  | t  | t  | t  |
| 2  | a  | a  | g  | t  | t  | t  | t  | t  | a  | c  |
| 3  | a  | a  | g  | t  | t  | t  | t  | t  | a  | c  |
| 4  | a  | g  | t  | t  | t  | t  | a  | c  | a  | a  |
| 5  | g  | t  | t  | a  | c  | a  | a  | g  | c  |
| 6  | 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  | 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  | t  |
| 1  | a  | a  | g  | c  |
| 2  | a  | a  | g  | t  | t  | t  | t  | a  | c  |
| 3  | a  | a  | g  | t  | t  | t  | t  | a  | c  |
| 4  | a  | c  | a  | a  | g  | c  |
| 5  | g  | 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  | g  | c  |
| 12 | a  | g  | c  |
| 13 | g  | c  |
| 14 | c  |
Keyword-in-context search: suffix-sorting solution

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

KWIC search for "search" in Tale of Two Cities

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>632698</td>
</tr>
<tr>
<td>713727</td>
</tr>
<tr>
<td>660598</td>
</tr>
<tr>
<td>67610</td>
</tr>
<tr>
<td>4430</td>
</tr>
<tr>
<td>42705</td>
</tr>
<tr>
<td>499797</td>
</tr>
<tr>
<td>182045</td>
</tr>
<tr>
<td>143399</td>
</tr>
<tr>
<td>411801</td>
</tr>
<tr>
<td>158410</td>
</tr>
<tr>
<td>691536</td>
</tr>
<tr>
<td>536569</td>
</tr>
<tr>
<td>484763</td>
</tr>
</tbody>
</table>
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


Mary Had a Little Lamb

Bach's Goldberg Variations
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.

Analysis. Running time $\leq D N^2$, where $D$ is length of longest match.
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
13 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
0  a  a  c  a  a  g  t  t  t  a  c  a  a  g  c
1  2  3  4  5  6  7  8  9 10 11 12 13 14
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 < mobyduck.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
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 (!)

Q. Which one is more likely to lead to a cure cancer?
### Longest repeated substring: empirical analysis

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>brute</th>
<th>suffix sort</th>
<th>length of LRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRS.java</td>
<td>2.162</td>
<td>0.6 sec</td>
<td>0.14 sec</td>
<td>73</td>
</tr>
<tr>
<td>amendments.txt</td>
<td>18.369</td>
<td>37 sec</td>
<td>0.25 sec</td>
<td>216</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>191.945</td>
<td>1.2 hours</td>
<td>1.0 sec</td>
<td>58</td>
</tr>
<tr>
<td>mobyduck.txt</td>
<td>1.2 million</td>
<td>43 hours †</td>
<td>7.6 sec</td>
<td>79</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>2 months †</td>
<td>61 sec</td>
<td>12.567</td>
</tr>
<tr>
<td>pi.txt</td>
<td>10 million</td>
<td>4 months †</td>
<td>84 sec</td>
<td>14</td>
</tr>
<tr>
<td>pipi.txt</td>
<td>20 million</td>
<td>forever †</td>
<td>???</td>
<td>10 million</td>
</tr>
</tbody>
</table>

† estimated
Bad input: longest repeated substring very long.

- Ex: same letter repeated \( N \) times.
- Ex: two copies of the same Java codebase.

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

Running time. Quadratic (or worse) in the length of the longest match.
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 (beyond our scope)
   • Nobody knows.
Suffix sorting in linearithmic time

Manber's 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>0: b a b a a a a b c b a b a a a a 0</td>
<td>17: 0</td>
</tr>
<tr>
<td>1: a b a a a a b c b a b a a a a 0</td>
<td>16: a 0</td>
</tr>
<tr>
<td>2: b a a a a b c b a b a a a a 0</td>
<td>15: a a 0</td>
</tr>
<tr>
<td>3: a a a a b c b a b a a a a 0</td>
<td>14: a a a 0</td>
</tr>
<tr>
<td>4: a a b c b a b a a a a 0</td>
<td>13: a a a 0</td>
</tr>
<tr>
<td>5: a a b c b a b a a a a 0</td>
<td>12: a a a a 0</td>
</tr>
<tr>
<td>6: b c b a b a a a a a 0</td>
<td>11: a b a a a a a 0</td>
</tr>
<tr>
<td>7: b c b a b a a a a a 0</td>
<td>10: a b a a a a a 0</td>
</tr>
<tr>
<td>8: c b a b a a a a a 0</td>
<td>9: a a a a 0</td>
</tr>
<tr>
<td>9: b a b a a a a a 0</td>
<td>8: a a a 0</td>
</tr>
<tr>
<td>10: a b a a a a a 0</td>
<td>7: a a a 0</td>
</tr>
<tr>
<td>11: b a a a a a 0</td>
<td>6: b a a a a a 0</td>
</tr>
<tr>
<td>12: a a a a a 0</td>
<td>5: b a a a a a 0</td>
</tr>
<tr>
<td>13: a a a 0</td>
<td>4: b a a a a a 0</td>
</tr>
<tr>
<td>14: a a 0</td>
<td>3: b c b a b a a a a 0</td>
</tr>
<tr>
<td>15: a a 0</td>
<td>2: b a a a a b c b a b a a a a 0</td>
</tr>
<tr>
<td>16: a 0</td>
<td>1: a b a a a a b c b a b a a a a 0</td>
</tr>
<tr>
<td>17: 0</td>
<td>0: b a b a a a a b c b a b a a a a 0</td>
</tr>
</tbody>
</table>

sorted
### 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>0: \texttt{b a a a a a a b c b a b a a a a 0}</td>
<td>17: 0</td>
</tr>
<tr>
<td>1: \texttt{a b a a a b c b a b a a a a 0}</td>
<td>16: a 0</td>
</tr>
<tr>
<td>2: \texttt{b a a a a b c b a b a a a a 0}</td>
<td>15: a a a a 0</td>
</tr>
<tr>
<td>3: \texttt{a a a a b c b a b a a a a 0}</td>
<td>14: a a a a 0</td>
</tr>
<tr>
<td>4: \texttt{a a a b c b a b a a a a 0}</td>
<td>13: a a a a 0</td>
</tr>
<tr>
<td>5: \texttt{a b c b a b a a a a a 0}</td>
<td>12: a a a a a 0</td>
</tr>
<tr>
<td>6: \texttt{b c b a b a a a a a 0}</td>
<td>11: a a a a a a 0</td>
</tr>
<tr>
<td>7: \texttt{c b a b a a a a a 0}</td>
<td>10: a a a a a a 0</td>
</tr>
<tr>
<td>8: \texttt{b a b a a a a a 0}</td>
<td>9: a a a a a a 0</td>
</tr>
<tr>
<td>9: \texttt{a b a a a a a a a 0}</td>
<td>8: b a a a a a a a 0</td>
</tr>
<tr>
<td>10: \texttt{b a a a a a a a a 0}</td>
<td>7: b a a a a a a a 0</td>
</tr>
<tr>
<td>11: \texttt{a a a a a a a 0}</td>
<td>6: b a a a a a a a 0</td>
</tr>
<tr>
<td>12: \texttt{a a a a a a 0}</td>
<td>5: b a a a a a a 0</td>
</tr>
<tr>
<td>13: \texttt{a a a a 0}</td>
<td>4: b a a a a a 0</td>
</tr>
<tr>
<td>14: \texttt{a a a 0}</td>
<td>3: b a a a a a 0</td>
</tr>
<tr>
<td>15: \texttt{a a 0}</td>
<td>2: b a a a a a a a 0</td>
</tr>
<tr>
<td>16: \texttt{a 0}</td>
<td>1: b a a a a a a a a 0</td>
</tr>
<tr>
<td>17: \texttt{0}</td>
<td>0: b a a a a a a a a a 0</td>
</tr>
</tbody>
</table>

```
\textbf{sorted}
```
### 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 b c b a b a a a a 0 |
| 6 | a b c b a b 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 | a a 0 |
| 14 | a a a 0 |
| 13 | a a a a a 0 |
| 12 | a a a a a 0 |
| 11 | a a a a a 0 |
| 10 | a a a a a 0 |
| 9 | b a a a a a a 0 |
| 8 | b a a a a a a 0 |
| 7 | b a a a a a a 0 |
| 6 | b a a a a a 0 |
| 5 | b a a a a a 0 |
| 4 | b a a a a a 0 |
| 3 | b a a a a a 0 |
| 2 | b a a a a a 0 |
| 1 | b a a a a a 0 |

Sorted
## Linearithmic suffix sort example: phase 3

**original suffixes**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b b a a a a a a b c b a b a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a b a a a a a b c b a b a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b a b a a a a b c b a b a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a a a a b c b a b a a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a a a b c b a b a a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a a b c b a b a a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>b c b a b a a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>b c b a b a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>c b a b a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b b b a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a b a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>b a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a a a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>a a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**index sort (first eight characters)**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

finished (no equal keys)
### Constant-time string compare by indexing into inverse

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first four characters)</th>
<th>inverse frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a b a a a a b c b a b a a a a 0</td>
<td>17 0</td>
<td>0 14</td>
</tr>
<tr>
<td>a b a a a a b c b a b a a a a 0</td>
<td>16 a 0</td>
<td>1 9</td>
</tr>
<tr>
<td>b a a a a a b c b a b a a a a 0</td>
<td>15 a a 0</td>
<td>2 12</td>
</tr>
<tr>
<td>a a a a a b c b a b a a a a 0</td>
<td>14 a a a 0</td>
<td>3 4</td>
</tr>
<tr>
<td>a a a b c b a b a a a a 0</td>
<td>13 a a a a 0</td>
<td>4 7</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>12 a a a a 0</td>
<td>5 8</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>11 a a a a 0</td>
<td>6 11</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>10 a a a a 0</td>
<td>7 16</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>9 a a a a 0</td>
<td>8 17</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>8 a a a a 0</td>
<td>9 15</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>7 a a a a 0</td>
<td>10 10</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>6 a a a a 0</td>
<td>11 13</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>5 a a a a 0</td>
<td>12 5</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>4 a a a a 0</td>
<td>13 6</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>3 a a a a 0</td>
<td>14 3</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>2 a a a a 0</td>
<td>15 2</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>1 a a a a 0</td>
<td>16 1</td>
</tr>
<tr>
<td>a a a a b c b a b a a a a 0</td>
<td>0 a a a a 0</td>
<td>17 0</td>
</tr>
</tbody>
</table>

To find the index of a prefix, shifted 4 times:

- \( b a b a a a a b c b a b a a a a 0 \):
  - \( 0 + 4 = 4 \)
- \( b a a a a a b c b a b a a a a 0 \):
  - \( 9 + 4 = 13 \)

**To do this, inverse-index should be computed for the previous phase. May use for only the last phase.**

\[
\text{suffixes}_4[13] \leq \text{suffixes}_4[4] \quad (\text{because inverse}[13] < \text{inverse}[4])
\]

\[
SO \text{ suffixes}_8[9] \leq \text{suffixes}_8[0]
\]
# Suffix sort: experimental results

## time to suffix sort (seconds)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobyduck.txt</th>
<th>aesop-aesop.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36.000 †</td>
<td>4000 †</td>
</tr>
<tr>
<td>quicksort</td>
<td>9,5</td>
<td>167</td>
</tr>
<tr>
<td>LSD</td>
<td>not fixed length</td>
<td>not fixed length</td>
</tr>
<tr>
<td>MSD</td>
<td>395</td>
<td>out of memory</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6,8</td>
<td>162</td>
</tr>
<tr>
<td>3-way string 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>

† estimated
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.
- Should measure amount of data in keys, not number of keys.
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.
- $1.39 \cdot N \log N$ chars for random data.

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