5.1 **STRING SORTS**

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
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.**
- Programming systems (e.g., Java code).
- Communication systems (e.g., email).
- Information processing.
- Genomic sequences.
- ...

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

C char data type. Typically an 8-bit integer (between 0 and 255).
- Supports 7-bit ASCII.
- Represents only $2^8 = 256$ characters.

<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>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
<td>DLE</td>
</tr>
<tr>
<td>1</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
<td>SP</td>
</tr>
<tr>
<td>2</td>
<td>@</td>
<td>#</td>
<td>$</td>
<td>&amp;</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>&lt;</td>
<td>&gt;</td>
<td>?</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>7</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td>}</td>
<td>~</td>
<td>DEL</td>
<td></td>
</tr>
</tbody>
</table>

all $2^7 = 128$ ASCII characters

can use as an index into an array

Java char data type. A 16-bit unsigned integer (between 0 and 65,535).
- Supports 16-bit Unicode 1.0.1.
- Supports 21-bit Unicode 10.0.0 (awkwardly via UTF-8).
I 💖 Unicode
**The String data type (in Java 11)**

**String data type.** Immutable sequence of characters.

**Java 11 representation.** A fixed-length `char[]` array.

![String representation](image)

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
<th>Java</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>length</strong></td>
<td>number of characters</td>
<td><code>s.length()</code></td>
<td>1</td>
</tr>
<tr>
<td><strong>indexing</strong></td>
<td>character at index i</td>
<td><code>s.charAt(i)</code></td>
<td>1</td>
</tr>
<tr>
<td><strong>concatenation</strong></td>
<td>concatenate one string to the end of the other</td>
<td><code>s + t</code></td>
<td><code>len(s) + len(t)</code></td>
</tr>
<tr>
<td><strong>comparison</strong></td>
<td>compare two strings lexicographically</td>
<td><code>s.compareTo(t)</code></td>
<td><code>lcp(s, t)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>K</th>
<th>A</th>
<th>T</th>
<th>D</th>
<th>A</th>
<th>W</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>s.length()</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>

- **indexing**
- **concatenation**
- **comparison**

- Allocates new `char[]`
- Length of longest common prefix
Q. How to build a long string, one character at a time?

String performance trap

StringBuilder data type. Mutable sequence of characters.
Java representation. A resizing char[] array.

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

quadratic time \((1 + 2 + 3 + \ldots + n)\)

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

linear time \(n + (1 + 2 + 4 + 8 + 16 + \ldots + n)\)

Alternatively,
```
new StringBuilder(s).reverse().toString()
```
Q. Why are Java strings immutable?

Many compelling reasons!

・ Streamlines tracing/debugging.
・ Simplifies programming.
・ Maintains data structure invariants.
・ Strengthens security.
・ Improves performance.

Immutable strings.
Java, C#, Python, JavaScript, Scala, Go, ...

Mutable strings.
C, C++, Matlab, Ruby, PHP, …
Alphabets

**Digital key.** Sequence of digits over a given alphabet.

**Radix.** Number of digits $R$ in alphabet.

<table>
<thead>
<tr>
<th>name</th>
<th>$R()$</th>
<th>$\log_2 R()$</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>ACDEFGHIJKLMNOPQRSTUVWXYZWY</td>
</tr>
<tr>
<td>BASE64</td>
<td>64</td>
<td>6</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/</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>

**Note.** We use extended ASCII alphabet in this lecture (but analyze in terms of radix $R$).
5.1 **String Sorts**

- string in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
Frequency of calls to `compareTo()`. 

<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>$\Theta(1)$</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$\Theta(n)$</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 n \log_2 n$ *</td>
<td>$1.39 n \log_2 n$ *</td>
<td>$\Theta(\log n)$ *</td>
<td></td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 n \log_2 n$</td>
<td>$2 n \log_2 n$</td>
<td>$\Theta(1)$</td>
<td></td>
<td><code>compareTo()</code></td>
</tr>
</tbody>
</table>

* probabilistic

**Sorting lower bound.** In the worst case, any compare-based sorting algorithm makes $\Omega(n \log n)$ compares. 

**Q.** Can we sort strings faster (despite lower bound)?

**A.** Yes, by exploiting access to individual characters.

use characters to make $R$-way decisions (instead of binary decisions)
Key-indexed counting: assumptions about keys

Assumption. Each key is an integer between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort class roster by section number.
- Sort phone numbers by area code.
- Sort playing cards by suit.
- Sort string by first letter.
- Use as a subroutine in string sorting algorithm.

Remark. Keys typically have associated data ⇒ can’t simply count keys of each value.

<table>
<thead>
<tr>
<th>input name</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>Garcia</td>
<td>4</td>
</tr>
<tr>
<td>Harris</td>
<td>1</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>Johnson</td>
<td>4</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>Thomas</td>
<td>4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>White</td>
<td>2</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>

sorted result (by section)

<table>
<thead>
<tr>
<th>Harris</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>Garcia</td>
<td>4</td>
</tr>
<tr>
<td>Johnson</td>
<td>4</td>
</tr>
<tr>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>Thomas</td>
<td>4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>White</td>
<td>4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>
Key-indexed counting demo

**Goal.** Sort an array \( a[] \) of \( n \) characters between 0 and \( R - 1 \).

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- 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];
```

\( R = 6 \)

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

*use a for 0, b for 1, c for 2, d for 3, e for 4, f for 5*
Key-indexed counting demo

**Goal.** Sort an array $a[]$ of $n$ characters between 0 and $R - 1$.
- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- 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];
```

offset by 1 [stay tuned]
**Goal.** Sort an array `a[]` of `n` characters between 0 and `R – 1`.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- 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];
```

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

<table>
<thead>
<tr>
<th>r</th>
<th>count[r]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>9</td>
</tr>
<tr>
<td>-</td>
<td>12</td>
</tr>
</tbody>
</table>

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` characters between 0 and `R – 1`.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- 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];
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
<th>i</th>
<th>aux[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>7</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td>e</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td>f</td>
</tr>
</tbody>
</table>
**Goal.** Sort an array `a[]` of `n` characters between 0 and `R – 1`.

- Compute character frequencies.
- Compute cumulative frequencies.
- Distribute items to auxiliary array using cumulative frequencies.
- 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]]++;

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];
```
Which of the following are properties of key-indexed counting?

A. $\Theta(n + R)$ time.
B. $\Theta(n + R)$ extra space.
C. Stable.
D. All of the above.
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 (LSD) radix sort

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

<table>
<thead>
<tr>
<th>$d$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$a$</th>
<th>$b$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
<th>$i$</th>
<th>$j$</th>
<th>$k$</th>
<th>$l$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>0</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>0</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>0</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
</tr>
<tr>
<td>1</td>
<td>$a$</td>
<td>$d$</td>
<td>$d$</td>
<td>1</td>
<td>$c$</td>
<td>$a$</td>
<td>$b$</td>
<td>1</td>
<td>$c$</td>
<td>$a$</td>
<td>$b$</td>
<td>1</td>
<td>$a$</td>
<td>$d$</td>
<td>$d$</td>
</tr>
<tr>
<td>2</td>
<td>$c$</td>
<td>$a$</td>
<td>$b$</td>
<td>2</td>
<td>$e$</td>
<td>$b$</td>
<td>$b$</td>
<td>2</td>
<td>$f$</td>
<td>$a$</td>
<td>$d$</td>
<td>2</td>
<td>$b$</td>
<td>$a$</td>
<td>$d$</td>
</tr>
<tr>
<td>3</td>
<td>$f$</td>
<td>$a$</td>
<td>$d$</td>
<td>3</td>
<td>$a$</td>
<td>$d$</td>
<td>$d$</td>
<td>3</td>
<td>$b$</td>
<td>$a$</td>
<td>$d$</td>
<td>3</td>
<td>$b$</td>
<td>$e$</td>
<td>$d$</td>
</tr>
<tr>
<td>4</td>
<td>$f$</td>
<td>$e$</td>
<td>$e$</td>
<td>4</td>
<td>$f$</td>
<td>$a$</td>
<td>$d$</td>
<td>4</td>
<td>$d$</td>
<td>$a$</td>
<td>$d$</td>
<td>4</td>
<td>$b$</td>
<td>$e$</td>
<td>$e$</td>
</tr>
<tr>
<td>5</td>
<td>$b$</td>
<td>$a$</td>
<td>$d$</td>
<td>5</td>
<td>$b$</td>
<td>$a$</td>
<td>$d$</td>
<td>5</td>
<td>$e$</td>
<td>$b$</td>
<td>$b$</td>
<td>5</td>
<td>$c$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>6</td>
<td>$d$</td>
<td>$a$</td>
<td>$d$</td>
<td>6</td>
<td>$d$</td>
<td>$a$</td>
<td>$d$</td>
<td>6</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
<td>6</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>7</td>
<td>$b$</td>
<td>$e$</td>
<td>$e$</td>
<td>7</td>
<td>$f$</td>
<td>$e$</td>
<td>$d$</td>
<td>7</td>
<td>$a$</td>
<td>$d$</td>
<td>$d$</td>
<td>7</td>
<td>$d$</td>
<td>$a$</td>
<td>$d$</td>
</tr>
<tr>
<td>8</td>
<td>$f$</td>
<td>$e$</td>
<td>$d$</td>
<td>8</td>
<td>$b$</td>
<td>$e$</td>
<td>$d$</td>
<td>8</td>
<td>$f$</td>
<td>$e$</td>
<td>$d$</td>
<td>8</td>
<td>$e$</td>
<td>$b$</td>
<td>$b$</td>
</tr>
<tr>
<td>9</td>
<td>$b$</td>
<td>$e$</td>
<td>$d$</td>
<td>9</td>
<td>$f$</td>
<td>$e$</td>
<td>$d$</td>
<td>9</td>
<td>$b$</td>
<td>$e$</td>
<td>$d$</td>
<td>9</td>
<td>$f$</td>
<td>$a$</td>
<td>$d$</td>
</tr>
<tr>
<td>10</td>
<td>$e$</td>
<td>$b$</td>
<td>$b$</td>
<td>10</td>
<td>$b$</td>
<td>$e$</td>
<td>$e$</td>
<td>10</td>
<td>$f$</td>
<td>$e$</td>
<td>$e$</td>
<td>10</td>
<td>$f$</td>
<td>$e$</td>
<td>$d$</td>
</tr>
<tr>
<td>11</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
<td>11</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
<td>11</td>
<td>$b$</td>
<td>$e$</td>
<td>$e$</td>
<td>11</td>
<td>$f$</td>
<td>$e$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

sort is stable (arrows do not cross)

strings sorted!
**LSD string sort: correctness proof**

**Proposition.** LSD sorts any array of \( n \) strings, each of length \( w \), in \( \Theta(w(n + R)) \) time.

**Pf of correctness.** [ by induction on # passes ]

- Inductive hypothesis: after pass \( i \), strings are sorted by last \( i \) characters.
- After pass \( i + 1 \), string are sorted by last \( i + 1 \) last characters because...
  - if two strings differ on sort key, key-indexed counting puts them in proper relative order
  - if two strings agree on sort key, stability of key-indexed counting keeps them in proper relative order

**Proposition.** LSD sort is stable.

**Pf.** Key-indexed counting is stable.
LSD string sort (for fixed-length strings): Java implementation

```java
public class LSD {
    public static void sort(String[] a, int w) {
        int R = 256;  // radix R
        int n = a.length;
        String[] aux = new String[n];

        for (int d = w - 1; d >= 0; d--) {  // do key-indexed counting for each digit from right to left
            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 calls to `compareTo()` and `charAt()`.

<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>( \Theta(1) )</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>( n \log_2 n )</td>
<td>( n \log_2 n )</td>
<td>( \Theta(n) )</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>( 1.39 n \log_2 n ) *</td>
<td>( 1.39 n \log_2 n ) *</td>
<td>( \Theta(\log n) ) *</td>
<td></td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>( 2 n \log_2 n )</td>
<td>( 2 n \log_2 n )</td>
<td>( \Theta(1) )</td>
<td></td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD sort †</td>
<td>( 2w n )</td>
<td>( 2w n )</td>
<td>( \Theta(n + R) )</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

one call to `compareTo()` can involve as many as \( 2w \) calls to `charAt()`

and \( \Theta(w(n+R)) \) array accesses

* probabilistic
† fixed-length \( w \) keys
Google CEO Eric Schmidt interviews Barack Obama in November 2007
Radix sorting: quiz 2

Which algorithm below is fastest for sorting 1 million 32-bit integers?

A. Mergesort.
B. Quicksort.
C. Arrays.sort().
D. LSD sort.
5.1 **String Sorts**

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

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

<table>
<thead>
<tr>
<th>String</th>
<th>Sort Key ($d = 0$)</th>
<th>Sort Key ($d = 1$)</th>
<th>Sort Key ($d = 2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abda</td>
<td>daad</td>
<td>baad</td>
<td>baad</td>
</tr>
<tr>
<td>adad</td>
<td>cace</td>
<td>caab</td>
<td>caab</td>
</tr>
<tr>
<td>caba</td>
<td>beae</td>
<td>beae</td>
<td>beae</td>
</tr>
<tr>
<td>fabd</td>
<td>bdb</td>
<td>adad</td>
<td>adad</td>
</tr>
<tr>
<td>bead</td>
<td>ce</td>
<td>caab</td>
<td>caab</td>
</tr>
<tr>
<td>bdea</td>
<td>eb</td>
<td>cace</td>
<td>cace</td>
</tr>
<tr>
<td>fedb</td>
<td>ed</td>
<td>badd</td>
<td>badd</td>
</tr>
<tr>
<td>bede</td>
<td>ed</td>
<td>badd</td>
<td>badd</td>
</tr>
<tr>
<td>ebbe</td>
<td>ee</td>
<td>cace</td>
<td>cace</td>
</tr>
<tr>
<td>acec</td>
<td>ee</td>
<td>fadd</td>
<td>fadd</td>
</tr>
</tbody>
</table>

strings not sorted!
Most-significant-digit-first (MSD) radix sort

Overview.

- Partition array into \( R \) subarrays according to first character.  
- Recursively sort all strings that start with each character.  

(excluding the first characters in subsequent sorts)

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

sort key \((d = 0)\)  

```
0  a  d  d
1  a  c  e
```

count[]

```
0  a  d  d
1  a  c  e
```

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

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

sort subarrays recursively  
(excluding first characters)

use key-indexed counting

key-indexed counts delineate subarray boundaries
MSD string sort (for fixed-length strings): Java implementation

```java
public static void sort(String[] a, int w) {
    aux = new String[a.length];
    sort(a, aux, w, 0, a.length - 1, 0);
}

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

    sort(a, aux, w, lo, lo + count[0] - 1, d+1); // sort R subarrays recursively
    for (int r = 1; r < R; r++)
        sort(a, aux, w, lo + count[r-1], lo + count[r] - 1, d+1);
}
```

Key-indexed counting (using character d)

- The `sort` method sorts the input array `a` into the `aux` array.
- It first checks if the subarray is of length 0 or 1, or if all characters match.
- It uses an auxiliary array `aux` to store temporary values.
- The method recursively sorts subarrays of length 0 or 1; or all `w` characters match.

Recycling `aux[]` array but not `count[]` array.
Variable-length strings

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

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>e</th>
<th>a</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>e</td>
<td>a</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>e</td>
<td>a</td>
<td>s</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>e</td>
<td></td>
<td>l</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>l</td>
</tr>
<tr>
<td>6</td>
<td>s</td>
<td>h</td>
<td>o</td>
<td>r</td>
</tr>
</tbody>
</table>

why smaller?

"she" before "shells"

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

**C strings.** Terminated with null character (`'\0'`) ⇒ no extra work needed.
For which family of inputs is MSD sort likely to be faster than LSD sort?

A. Random strings.
B. All equal strings.
C. Both A and B.
D. Neither A nor B.

<table>
<thead>
<tr>
<th>random</th>
<th>all equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 E I 0 4 0 2</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>1 H Y L 4 9 0</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>1 R O Z 5 7 2</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>2 H X E 7 3 4</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>2 I Y E 2 3 0</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>2 X O R 8 4 6</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>3 C D B 5 7 3</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>3 C V P 7 2 0</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>3 I G J 3 1 9</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>3 K N A 3 8 2</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>3 T A V 8 7 9</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>4 C Q P 7 8 1</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>4 Q G I 2 3 4</td>
<td>1 D N B 3 7 7</td>
</tr>
<tr>
<td>4 Y H V 2 2 9</td>
<td>1 D N B 3 7 7</td>
</tr>
</tbody>
</table>
MSD string sort: performance

Observation. MSD examines just enough character to sort the keys.

Proposition. For random strings, MSD examines $\Theta(n \log_R n)$ characters on average.

Remark. This can be sublinear in the input size $\Theta(n w)$.

Proposition. In the worst case, MSD requires $\Theta(n + wR)$ extra space.
Summary of the performance of sorting algorithms

Frequency of calls to `compareTo()` and `charAt()`.

<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>$\Theta(1)$</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$\Theta(n)$</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 n \log_2 n^*$</td>
<td>$1.39 n \log_2 n^*$</td>
<td>$\Theta(\log n)^*$</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 n \log_2 n$</td>
<td>$2 n \log_2 n$</td>
<td>$\Theta(1)$</td>
<td></td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD sort †</td>
<td>$2w n$</td>
<td>$2w n$</td>
<td>$\Theta(n + R)$</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD sort ‡</td>
<td>$2w n$</td>
<td>$n \log_R n$</td>
<td>$\Theta(n + wR)$</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

but can make $\Theta(wnR)$ array accesses

$n / 2$ pairs of duplicate keys

* probabilistic
† fixed-length $w$ keys
‡ average-length $w$ keys
Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.
- MSD is much too slow for small subarrays.
- Essential for performance.

Optimization 1. Replace recursion with explicit stack.
- Push subarrays to be sorted onto stack.
- One count[] array now suffices.

Optimization 2. Do R-way partitioning in place.
- Eliminates aux[] array.
- Sacrifices stability.

American national flag problem  Dutch national flag problem

ABSTRACT: Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place “American flag” sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.
5.1 String Sorts

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

- Partition array into 3 subarrays according to first character of pivot.
- Recursively sort 3 subarrays.

3-way string quicksort

use Dijkstra 3-way partitioning algorithm

partition array into 3 subarrays

recursively sort 3 subarrays
### 3-way string quicksort: trace of recursive calls

Trace of first few recursive calls for 3-way string quicksort (subarrays of length 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;  // subarrays of length 0 or 1
    int pivot = charAt(a[lo], d);

    int lt = lo, gt = hi;
    int i = lo + 1;
    while (i <= gt)
    {
        int c = charAt(a[i], d);
        if       (c < pivot) exch(a, lt++, i++);
        else if (c > pivot) exch(a, i, gt--);
        else                   i++;
    }

    sort(a, lo, lt-1, d);  // sort 3 subarrays recursively
    if (pivot != -1) sort(a, lt, gt, d+1);
    sort(a, gt+1, hi, d);
}
```

Dijkstra 3-way partitioning (using character at index d)
3-way string quicksort vs. competitors

3-way string quicksort vs. MSD sort.
• In-place; short inner loop; cache-friendly.
• Not stable.

3-way string quicksort vs. standard quicksort.
• Typically uses $\sim 2n \ln n$ character compares (instead of $\sim 2n \ln n$ string compares).
• Faster for keys with long common prefixes (and this is a common case!)

Bottom line. 3-way string quicksort is often the method of choice for sorting strings.
# Summary of the performance of sorting algorithms

Frequency of calls to `compareTo()` and `charAt()`.

<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>(\Theta(1))</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
<td>(\Theta(n))</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>(1.39 n \log_2 n)</td>
<td>(1.39 n \log_2 n)</td>
<td>(\Theta(\log n))</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>(2 n \log_2 n)</td>
<td>(2 n \log_2 n)</td>
<td>(\Theta(1))</td>
<td>✔</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD sort †</td>
<td>(2w n)</td>
<td>(2w n)</td>
<td>(\Theta(n + R))</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD sort ‡</td>
<td>(2w n)</td>
<td>(n \log_R n)</td>
<td>(\Theta(n + w R))</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>(1.39 w n \log_2 R)</td>
<td>(1.39 n \log_2 n)</td>
<td>(\Theta(\log n + w))</td>
<td>✔</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length \(w\) keys
‡ 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
Keyword-in-context search

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

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

**input string**

```
it was best it was w
```

**form suffixes**

```
0  it was best it was w
1  twas best it was w
2  was best it was w
3  as best it was w
4  sbest it was w
5  est it was w
6  st it was w
7  tit was w
8  it was w
9  twas w
10  w as w
11  as w
12  sw
13  w
```

**sort suffixes to bring query strings together**

```
3  as best it was w
12  as w
12  best it was w
5  est it was w
6  sbest it was w
9  it was w
0  it was best it was w
9  it was w
4  sbest it was w
7  st it was w
13  sw
8  st it was w
13  sw
1  twas best it was w
10  twas w
14  w
2  w as best it was w
11  w as w
```

**array of suffix indices**

(in sorted order)
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

... sealed my letter and ...
Radix sorting: quiz 4

How much memory as a function of $n$?

A. $\Theta(1)$
B. $\Theta(n)$
C. $\Theta(n \log n)$
D. $\Theta(n^2)$

String[] suffixes = new String[n];
for (int i = 0; i < n; i++)
    suffixes[i] = s.substring(i, n);
Arrays.sort(suffixes);

3rd printing (2012)
Q. How to efficiently form (and sort) the $n$ suffixes?

```java
String[] suffixes = new String[n];
for (int i = 0; i < n; i++)
    suffixes[i] = s.substring(i, n);
Arrays.sort(suffixes);
```

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>Java 7u5</th>
<th>Java 7u6</th>
</tr>
</thead>
<tbody>
<tr>
<td>amendments.txt</td>
<td>18 K</td>
<td>0.25 sec</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>192 K</td>
<td>1.0 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>mobyduck.txt</td>
<td>1.2 M</td>
<td>7.6 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 M</td>
<td>61 sec</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

$\Theta(n^2)$ time and space to form suffixes!
The String data type: Java 7u6 implementation

```java
public final class String implements Comparable<String>
{
    private char[] value; // sequence of characters in string
    private int hash;     // cache of hashCode()
    ...
}
```

String `s = "Hello, World";`

```
value[]  H E L L O , W O R L D
         0 1 2 3 4 5 6 7 8 9 10 11
```

String `t = s.substring(7, 12);` (allocates new char[] array ⇒ linear extra memory)

```
value[]  W O R L D
         0 1 2 3 4
```
The String data type: Java 7u5 implementation

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

String `s = "Hello, World";`

- `value[]`: `H E L L O , W O R L D`
- `offset = 0`
- `length = 12`

String `t = s.substring(7, 12);`

- Reuses original `char[]` array ⇒ constant extra memory
- `value[]`: `H E L L O , W O R L D`
- `offset = 7`
- `length = 5`
The String data type: performance summary

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

Java 7u5. Immutable char[] array, offset, length, hash cache.
Java 7u6. Immutable char[] array, hash cache.

<table>
<thead>
<tr>
<th>operation</th>
<th>Java 7u5</th>
<th>Java 7u6</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>indexing</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>(m + n)</td>
<td>(m + n)</td>
</tr>
<tr>
<td>substring extraction</td>
<td>1</td>
<td>(n)</td>
</tr>
<tr>
<td>immutable?</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>memory</td>
<td>(64 + 2n)</td>
<td>(56 + 2n)</td>
</tr>
</tbody>
</table>
A Reddit exchange

I'm the author of the substring() change. As has been suggested in the analysis here there were two motivations for the change

- Reduce the size of String instances. Strings are typically 20-40% of common apps footprint.
- Avoid memory leakage caused by retained substrings holding the entire character array.

Changing this function, in a bugfix release no less, was totally irresponsible. It broke backwards compatibility for numerous applications with errors that didn't even produce a message, just freezing and timeouts... All pain, no gain. Your work was not just vain, it was thoroughly destructive, even beyond its immediate effect.

http://www.reddit.com/r/programming/comments/1qw73v/til_oracle_changed_the_internal_string
Q. How to efficiently form (and sort) suffixes in Java 7u6?
A. Define `Suffix` class à la Java 7u5 String representation.

```java
public class Suffix implements Comparable<Suffix> {
    private final String text;
    private final int offset;

    public Suffix(String text, int offset) {
        this.text = text;
        this.offset = offset;
    }

    public int length() {
        return text.length() - offset;
    }
    public char charAt(int i) {
        return text.charAt(offset + i);
    }
    public int compareTo(Suffix that) {
        /* see textbook */
    }
}
```
**Suffix sort**

**Q.** How to efficiently form (and sort) suffixes in Java 7u6?

**A.** Define `Suffix` class à la Java 7u5 String representation.

```java
Suffix[] suffixes = new Suffix[n];
for (int i = 0; i < n; i++)
    suffixes[i] = new Suffix(s, i);
Arrays.sort(suffixes);
```

**Optimizations.** [5× faster and 32× less memory than Java 7u5 version]

- Use 3-way string quicksort instead of `Arrays.sort()`.
- Manipulate suffix offsets directly instead of via explicit `Suffix` objects.
Conjecture. [Knuth 1970] Impossible to compute suffix array in \( \Theta(n) \) time.

Proposition. [Weiner 1973] Can solve in \( \Theta(n) \) time (suffix trees).

“has no practical virtue... but a historic monument in the area of string processing.”
Suffix arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Many ingenious algorithms.

• Constants and memory footprint very important.
• State–of–the art still changing.

<table>
<thead>
<tr>
<th>year</th>
<th>algorithm</th>
<th>worst case</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>Manber–Myers</td>
<td>$n \log n$</td>
<td>$8n$</td>
</tr>
<tr>
<td>1999</td>
<td>Larsson–Sadakane</td>
<td>$n \log n$</td>
<td>$8n$</td>
</tr>
<tr>
<td>2003</td>
<td>Kärkkäinen–Sanders</td>
<td>$n$</td>
<td>$13n$</td>
</tr>
<tr>
<td>2003</td>
<td>Ko–Aluru</td>
<td>$n$</td>
<td>$10n$</td>
</tr>
<tr>
<td>2008</td>
<td>divsufsort2</td>
<td>$n \log n$</td>
<td>$5n$</td>
</tr>
<tr>
<td>2010</td>
<td>sais</td>
<td>$n$</td>
<td>$6n$</td>
</tr>
</tbody>
</table>

see lecture videos
about 10× faster than Manber–Myers

good choices (libdivsufsort)
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 = total number of characters (not number of strings).
- Not all of the characters have to be examined.

Long strings are rarely random in practice.

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