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 programs).
- 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.
- Supports 7-bit ASCII.
- Can represent at most 256 characters.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>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>ENQ</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>
</tr>
<tr>
<td>1</td>
<td>DLE</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>
</tr>
<tr>
<td>2</td>
<td>SP</td>
<td>&quot;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</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>
</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>
</tr>
</tbody>
</table>

### ASCII characters
- Supports 7-bit ASCII.
- Can represent at most 256 characters.

### Unicode characters
- Supports 16-bit Unicode 1.0.
- Supports 21-bit Unicode 8.0 (awkwardly).

### Java char data type.
A 16-bit unsigned integer.
- Supports 16-bit Unicode 1.0.
- Supports 21-bit Unicode 8.0 (awkwardly).

---

*All 128 ASCII characters*  
*Some Unicode characters*
The String data type

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

**Length.** Number of characters.

**Indexing.** Get the \( i \)th character.

**Concatenation.** Concatenate one string to the end of another.

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
\end{array}
\]

\[s.length()\]

\[s.charAt(3)\]
The String data type: representation

Representation (Java 7 and 8). Immutable char[] array + cache of hash.

<table>
<thead>
<tr>
<th>operation</th>
<th>Java</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>s.length()</td>
<td>1</td>
</tr>
<tr>
<td>indexing</td>
<td>s.charAt(i)</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>s + t</td>
<td>m + n</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
String performance trap

Q. How to build a long string, one character at a time?

A. Use StringBuilder data type (mutable char[] resizing array).

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

```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();
}
```

quadratic time

(1 + 2 + 3 + … + n)

linear time
Q. Why are Java strings immutable?
**Alphabets**

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

<table>
<thead>
<tr>
<th>name</th>
<th>$R$</th>
<th>$\lg R$</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>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>
5.1 String Sorts

- Strings in Java
- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
“The first rule of program optimization: don't do it.
The second rule of program optimization (for experts only!): don't do it yet.” – Michael A. Jackson
Review: summary of the performance of sorting algorithms

Frequency of operations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{4} n^2$</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>$n$</td>
<td>✔</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>✔</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></td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

* probabilistic

compareTo() not constant time for strings

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

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)
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 string sorting algorithm.

Remark. Keys may have associated data ⇒ can’t simply count keys of each value.
Key-indexed counting demo

**Goal.** Sort an array `a[]` of `n` integers between 0 and `R – 1`.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```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>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>

Hints:
- `count` array for new cumulates.
- `aux` array for moving items.

Use:
- `a` for 0
- `b` for 1
- `c` for 2
- `d` for 3
- `e` for 4
- `f` for 5

**R = 6**
Key-indexed counting demo

**Goal.** Sort an array $a[]$ of $n$ integers between 0 and $R - 1$.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

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

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

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

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

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

**Goal.** Sort an array $a[]$ of $n$ integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

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

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

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

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

for (int i = 0; i < n; i++)
    a[i] = aux[i];
```

<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>

<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>
</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 $n$ 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];
```

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

move items

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];
```
Radix sorting: quiz 1

Which of the following are properties of key–indexed counting?

A. Running time proportional to $n + R$.

B. Extra space proportional to $n + R$.

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 $d^{th}$ character as the key (using key-indexed counting).

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

sort is stable
(arrows do not cross)
**LSD string sort: correctness proof**

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

**Pf.** [ by induction on \( i \) ]

After pass \( i \), strings are sorted by last \( i \) characters.
- If two strings differ on sort key, key-indexed 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.
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];
        }
    }
}

fixed-length W strings
radix R
do key-indexed counting for each digit from right to left
key-indexed counting
Summary of the performance of sorting algorithms

Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{4} n^2$</td>
<td>1</td>
<td>✔️</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>$n$</td>
<td>✔️</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>✔️</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>✔️</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort †</td>
<td>$2 W (n + R)$</td>
<td>$2 W (n + R)$</td>
<td>$n + R$</td>
<td>✔️</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length $W$ keys

1 call to compareTo() can involve as many as $W$ calls to charAt()

Q. What if strings are not all of same length $W$?
Which sorting method to use to sort 1 million 32-bit integers?

A. Insertion sort.
B. Mergesort.
C. Quicksort.
D. Heapsort.
E. LSD radix sort.
Sort Array of 128-Bit Numbers

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 radix 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 \(d\)th character as the key (using key-indexed counting).

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

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

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

not sorted!
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).
Variable-length strings

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>-1</th>
</tr>
</thead>
<tbody>
<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>l</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>
<tr>
<td>7</td>
<td>s</td>
<td>u</td>
<td>r</td>
<td>e</td>
</tr>
</tbody>
</table>

why smaller?
she before shells

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.
public static void sort(String[] a) {
    aux = new String[a.length];
    sort(a, aux, 0, a.length - 1, 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);
}
### 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!

<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>1EIO402</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>2XOR846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CDB573</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>

*CompareTo() based sorts can also be sublinear!*
## Summary of the performance of sorting algorithms

### Frequency of operations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>( \frac{1}{2} n^2 )</td>
<td>( \frac{1}{4} n^2 )</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>( n \lg n )</td>
<td>( n \lg n )</td>
<td>( n )</td>
<td>✔</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></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></td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort †</td>
<td>2 ( W(n + R) )</td>
<td>2 ( W(n + R) )</td>
<td>( n + R )</td>
<td>✔</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD sort ‡</td>
<td>2 ( W(n + R) )</td>
<td>( n \log_k n )</td>
<td>( n + D R )</td>
<td>✔</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
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
5.1 **String Sorts**

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

Overview. Do 3-way partitioning on the $d^{th}$ character.
- Less overhead than $R$-way partitioning in MSD radix 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

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 v = charAt(a[lo], d);

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

- Is in-place.
- Is cache-friendly.
- Has a short inner loop.
- But not stable.

Bottom line. 3-way string quicksort is 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>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{4} n^2$</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
<td>✔</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>✔</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>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort †</td>
<td>$2 W (n + R)$</td>
<td>$2 W (n + R)$</td>
<td>$n + R$</td>
<td>✔</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD sort ‡</td>
<td>$2 W (n + R)$</td>
<td>$n \log_R n$</td>
<td>$n + D R$</td>
<td>✔</td>
<td>charAt()</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>$1.39 W n \log R^*$</td>
<td>$1.39 n \log n$</td>
<td>$\log n + W^*$</td>
<td>✔</td>
<td>charAt()</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 context).

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

...
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
ost giless to search for contraband
her unavailing search for your fathe
tle 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**

```

it was best it was w
```

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

**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 best it was w
6 est it was w
7 st it was w
8 t it was w
9 it was w
10 twas w
11 was w
12 as w
13 sw
14 w
```

**sort suffixes to bring query strings together**

```

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 best it was w
6 est it was w
7 st it was w
8 t it was w
9 it was w
10 as w
11 twas best it was w
12 as w
13 sw
14 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

<table>
<thead>
<tr>
<th>Location</th>
<th>Word Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>632698</td>
<td>sealed my letter and...</td>
</tr>
<tr>
<td>713727</td>
<td>seamstress is lifted...</td>
</tr>
<tr>
<td>660598</td>
<td>seamstress of twenty...</td>
</tr>
<tr>
<td>67610</td>
<td>seamstress who was with...</td>
</tr>
<tr>
<td>4430</td>
<td>search for contraband...</td>
</tr>
<tr>
<td>42705</td>
<td>search for your father...</td>
</tr>
<tr>
<td>499797</td>
<td>search of her husband...</td>
</tr>
<tr>
<td>182045</td>
<td>search of impoverished...</td>
</tr>
<tr>
<td>143399</td>
<td>search of other carriage...</td>
</tr>
<tr>
<td>411801</td>
<td>search the straw hold...</td>
</tr>
<tr>
<td>158410</td>
<td>seared marking about...</td>
</tr>
<tr>
<td>691536</td>
<td>seas and madame deafar...</td>
</tr>
<tr>
<td>536569</td>
<td>sease a terrible pass...</td>
</tr>
<tr>
<td>484763</td>
<td>sease that had brought...</td>
</tr>
</tbody>
</table>
War story

Q. How to efficiently form (and sort) 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 thousand</td>
<td>0.25 sec</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>192 thousand</td>
<td>1.0 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>mobyduck.txt</td>
<td>1.2 million</td>
<td>7.6 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>61 sec</td>
<td>out of memory</td>
</tr>
</tbody>
</table>
Radix sorting: quiz 3

How much memory as a function of n?

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

A. 1
B. n
C. n log n
D. n^2
The String data type: Java 7u5 implementation

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

String s = "Hello, World";

String t = s.substring(7, 12);
(constant extra memory)

(length = 12)

value[]

<table>
<thead>
<tr>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>,</th>
<th>W</th>
<th>O</th>
<th>R</th>
<th>L</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<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>10</td>
</tr>
</tbody>
</table>

offset = 0

(length = 5)

value[]

| H | E | L | L | O | , | W | O | R | L | D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

offset = 7
The String data type: Java 7u6 implementation

```java
public final class String implements Comparable<String> {
    private char[] value;  // characters
    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);`

(Linear extra memory)

```
value[]    W  O  R  L  D
          0 1 2 3 4
```
The String data type: performance

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>substring extraction</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>m + n</td>
<td>m + 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 ala Java 7u5 String.

```java
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 */ }
}
```

<table>
<thead>
<tr>
<th>text[]</th>
<th>HELLO, WORLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
</tbody>
</table>
Q. How to efficiently form (and sort) suffixes in Java 7u6?
A. Define Suffix class ala Java 7u5 String.

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

4th printing (2013)
Lessons learned

Lesson 1. Put performance guarantees in API.
Lesson 2. If API has no performance guarantees, can’t rely upon them!

Corollary. May want to avoid String data type when performance matters.
  • Are you sure charAt() and length() take constant time?
  • If lots of calls to charAt(), overhead for function calls is large.
  • If lots of small strings, memory overhead of String is large.

Ex. Our optimized algorithm for suffix arrays is 5× faster than our original Java 7u5 solution (and reduces memory footprint by 32×).
  • 3-way radix quicksort.
  • Avoid Suffix objects.
Suffix arrays: theory


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

A Space-Economical Suffix Tree Construction Algorithm

EDWARD M. McCREIGHT
Xerox Palo Alto Research Center, Palo Alto, California

ABSTRACT. A new algorithm is presented for constructing auxiliary digital search trees to aid in exact-match substring searching. This algorithm has the same asymptotic running time bound as previously published algorithms, but is more economical in space. Some implementation considerations are discussed, and new work on the modification of these search trees in response to incremental changes in the strings they index (the update problem) is presented.

On-line construction of suffix trees

Esko Ukkonen

Department of Computer Science, University of Helsinki,
P. O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
Tel.: +358-0-7084172, fax: +358-0-7084441
Email: ukkonen@cs.helsinki.fi
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>
String sorting summary

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

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

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