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.

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

The String data type

String data type. 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)
```

The String data type: Java implementation

```java
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

```
<table>
<thead>
<tr>
<th>String operation</th>
<th>guarantee extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
</tr>
<tr>
<td>concat()</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).
Underlying implementation. Resizing char[] array and length.

### Operation Guarantee

<table>
<thead>
<tr>
<th>Operation</th>
<th>String</th>
<th>StringBuilder</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>N</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>1 *</td>
</tr>
</tbody>
</table>

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

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;
}
```

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();
}
```

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

<table>
<thead>
<tr>
<th>suffixes</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>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td>acaagtttacaagc</td>
<td></td>
</tr>
</tbody>
</table>

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```java
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.

```java
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;
}
```
Longest common prefix

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

- Linear time (worst case)
- Sublinear time (typical case)

```
public static int lcp(String s, String t) {
    int N = Math.min(s.length(), t.length);
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return i;
    return N;
}
```

Running time. Proportional to length $D$ of longest common prefix.

Remark. Also can compute `compareTo()` in sublinear time.

Alphabets

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

<table>
<thead>
<tr>
<th>name</th>
<th>$R$</th>
<th>$\lg R$</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>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>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>EXTENDED_ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

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 $\approx$ 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>$N$</td>
<td>$N$</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>
</tbody>
</table>

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

Q. Can we do better (despite the lower bound)?
A. Yes, if we don’t depend on key compares.
### Key-indexed counting: assumptions about keys

**Assumption.** Keys are integers between 0 and \( R - 1 \).

**Implication.** Can use key as an array index.

**Applications.**
- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

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

### Key-indexed counting demo

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

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

```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];
```
Key-indexed counting demo

**Goal.** Sort an array $a[i]$ 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]++; // Count frequencies of each letter using key as index.
for (int r = 0; r < R; r++)
    count[r+1] += count[r]; // Compute frequency cumulates which specify destinations.
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i]; // Access cumulates using key as index to move items.
a[i] = aux[i]; // Copy back into original array.
```

---

Key-indexed counting demo

**Goal.** Sort an array $a[i]$ 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]++; // Count frequencies of each letter using key as index.
for (int r = 0; r < R; r++)
    count[r+1] += count[r]; // Compute frequency cumulates which specify destinations.
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i]; // Access cumulates using key as index to move items.
a[i] = aux[i]; // Copy back into original array.
```
Key-indexed counting demo

Goal. Sort an array $a[i]$ 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.

Sort an array $a[i]$ 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 i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
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 r = 0; r < R; r++)
    count[r+1]++; // 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 r = 0; r < R; r++)
    count[r+1]++; // 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]]++;    // count frequencies of each letter using key as index.

int[] aux = new int[R+1];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];    // Access cumulates using key as index to move items.

for (int r = 0; r < R; r++)
    count[r+1] += count[r];    // Compute frequency cumulates which specify destinations.

for (int i = 0; i < N; i++)
    a[i] = aux[count[a[i]]++];    // Copy back into original array.
```

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]]++;    // count frequencies of each letter using key as index.

int[] aux = new int[R+1];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];    // Access cumulates using key as index to move items.

r = 0;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];    // Compute frequency cumulates which specify destinations.

for (int i = 0; i < N; i++)
    a[i] = aux[count[a[i]]++];    // Copy back into original array.
```
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.

```
for (int i = 0; i < N; i++)
    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.

```
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
```

```
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.

```
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
```

```
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.

```
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
```

```
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 $\sim 11 N + 4 R$ 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? ✔
**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>sort key (d=2)</th>
<th>sort key (d=1)</th>
<th>sort key (d=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 d a h</td>
<td>1 d a h</td>
<td>0 a c a</td>
</tr>
<tr>
<td>1 a d d</td>
<td>1 c a h</td>
<td>1 a d d</td>
</tr>
<tr>
<td>2 c a h</td>
<td>2 f a d</td>
<td>2 b a d</td>
</tr>
<tr>
<td>3 f a d</td>
<td>3 a d d</td>
<td>3 b a d</td>
</tr>
<tr>
<td>4 f a d</td>
<td>4 f a d</td>
<td>4 b a d</td>
</tr>
<tr>
<td>5 b a d</td>
<td>5 a b h</td>
<td>5 c a b</td>
</tr>
<tr>
<td>6 d a h</td>
<td>6 a b h</td>
<td>6 d a b</td>
</tr>
<tr>
<td>7 a b h</td>
<td>7 f a d</td>
<td>7 a d d</td>
</tr>
<tr>
<td>8 f a d</td>
<td>8 a b h</td>
<td>8 f a d</td>
</tr>
<tr>
<td>9 b a d</td>
<td>9 f a d</td>
<td>9 b a d</td>
</tr>
<tr>
<td>10 a b h</td>
<td>10 b a d</td>
<td>10 f a d</td>
</tr>
<tr>
<td>11 a b h</td>
<td>11 f a d</td>
<td>11 f a d</td>
</tr>
</tbody>
</table>

Sort must be 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 sort puts them in proper relative order.
  - If two strings agree on sort key, stability keeps them in proper relative order.

---

**LSD string sort: Java implementation**

```java
public class LSD {
    public static void sort(String[] a, int W) {
        int R = 256;
        int N = a.length;
        String[] aux = new String[N];
        int[] count = new int[R+1];
        for (int d = W-1; d >= 0; d--)
            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>$N$</td>
<td>$N$</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>$2 W N$</td>
<td>$2 W N$</td>
<td>$N + R$</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length $W$ keys

Q. What if strings do not have same length?

### String sorting challenge 1

#### 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.

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

### 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.
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.
Sort in 8 passes.

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

**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).

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

**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).

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

---

MSD string sort: Java implementation

```java
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);
}
```

---

MSD string sort: potential for disastrous performance

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

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

---

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.
- Insertion sort, but start at \( d \)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);
}
```

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

Summary of the performance of sorting algorithms

Frequency of operations.

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 $k$-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char (but does re-examine characters not equal to the partitioning char).

```java
private static void sort(String[] a)
{ if (v >= 0) sort(a, lt, gt, d+1);
  else            i++;
}
```

3-way string quicksort: trace of recursive calls

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

3-way string quicksort vs. standard quicksort

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

**3-way string (radix) quicksort.**
- Uses $2N \ln N$ character compares on average for random strings.
- Avoids re-comparing 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.
• 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

<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</td>
<td>N</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</td>
<td>N + D R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>1.39 W N lg N</td>
<td>1.39 N lg N</td>
<td>log N + W</td>
<td>no</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

Keyword-in-context search

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

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

Longest repeated substring

Longest repeated substring: a musical application

Keyword-in-context search: suffix-sorting solution
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.

\[
\text{a a c a a g t t t a c a a g c}
\]

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

Longest repeated substring: Java implementation

```java
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 < mobydick.txt
```

Sorting challenge

Problem. Five scientists \( A, B, C, D, \) and \( E \) are looking for long repeated substring in a genome with over 1 billion nucleotides.

- \( A \) has a grad student do it by hand.
- \( B \) uses brute force (check all pairs).
- \( C \) uses suffix sorting solution with insertion sort.
- \( D \) uses suffix sorting solution with LSD string sort.
- \( E \) uses suffix sorting solution with 3-way string quicksort.

\( \checkmark \) Uses suffix sorting solution with 3-way string quicksort.

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>mobydic.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

## Suffix sorting: worst-case input

**Bad input:** longest repeated substring very long.
- Ex: same letter repeated $N$ times.
- Ex: two copies of the same Java codebase.

<table>
<thead>
<tr>
<th>form suffixes</th>
<th>sorted suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>t w i n s t w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>w i n s t w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>n s t w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>s t w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>t w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>w i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>i n s</td>
<td>i n s t w i n s</td>
</tr>
<tr>
<td>s</td>
<td>i n s t w i n s</td>
</tr>
</tbody>
</table>

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

Original suffixes:
1. baba
2. abaa
3. abab
4. babb
5. baba
6. abaa
7. abab
8. babb
9. baaa
10. aaab
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

Key-indexed counting sort (first character):
1. aaaa
2. aaaa
3. aaaa
4. aaaa
5. aaaa
6. aaaa
7. aaaa
8. aaaa
9. abba
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

Sorted:
1. abaaa
2. abaaa
3. abaaa
4. abaaa
5. abaaa
6. abaaa
7. abaaa
8. abaaa
9. abba
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

Linearithmic suffix sort example: phase 1

Original suffixes:
0. baba
1. abaa
2. abab
3. babb
4. baba
5. abaa
6. abab
7. babb
8. baaa
9. aaab
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

Index sort (first two characters):
0. a
1. a
2. a
3. a
4. a
5. a
6. a
7. a
8. a
9. a
10. a
11. a
12. a
13. a
14. a
15. a
16. a
17. a

Linearithmic suffix sort example: phase 2

Original suffixes:
0. baba
1. abaa
2. abab
3. babb
4. baba
5. abaa
6. abab
7. babb
8. baaa
9. aaab
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

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

Sorted:
1. abaaa
2. abaaa
3. abaaa
4. abaaa
5. abaaa
6. abaaa
7. abaaa
8. abaaa
9. abba
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

Linearithmic suffix sort example: phase 3

Original suffixes:
0. baba
1. abaa
2. abab
3. babb
4. baba
5. abaa
6. abab
7. babb
8. baaa
9. aaab
10. abba
11. abba
12. abba
13. abba
14. abba
15. abba
16. abba
17. abba

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

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

original suffixes

0  babaabaabcabaaaaa0
1  abaaaaabcabaaaaa0
2  baaaaabcabaaaaa0
3  aabcbabaaaaa0
4  abcabaaaaa0
5  cbabaaaaa0
6  cbabaaaaa0
7  babaabaabcabaaaaa0
8  abaaaaabcabaaaaa0
9  baaaaabcabaaaaa0
10 aabcbabaaaaa0
11 abcabaaaaa0
12 cbabaaaaa0
13 cbabaaaaa0
14 babaabaabcabaaaaa0
15 abaaaaabcabaaaaa0
16 baaaaabcabaaaaa0
17 aabcbabaaaaa0

50 suffixes[9] < suffixes[0]

index sort (first four characters)

0  0
1  0
2  0
3  0
4  0
5  a
6  a
7  a
8  a
9  a
10 a
11 a
12 a
13 a
14 0
15 0
16 0
17 0

inverse

0  14
1  9
2  12
3  4
4  7
5  8
6  11
7  16
8  17
9  15
10 10
11 13
12 5
13 3
14 3
15 2
16 1
17 0

Long strings are rarely random in practice.

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

3-way string quicksort is asymptotically optimal.

- $1.39 N \log N$ chars for random data.

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.

Suffix sort: experimental results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydict.txt</th>
<th>aesopaeasop.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>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>† estimated</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.
- 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.
- $1.39 N \log N$ chars for random data.

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