Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
String sorts
- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

Tries
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.
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

**String data type.** Sequence of characters (immutable).

**Underlying implementation.** Immutable char[] array, offset, and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>guarantee</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

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

---

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

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

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

<table>
<thead>
<tr>
<th>operation</th>
<th>String guarantee</th>
<th>String extra space</th>
<th>StringBuilder guarantee</th>
<th>StringBuilder extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
<td>1*</td>
<td>1*</td>
</tr>
</tbody>
</table>

* amortized

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

Q. How to efficiently reverse a string?

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

B. public static String reverse(String s) {
    StringBuilder rev = new StringBuilder();
    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>input string</th>
<th>suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a c a a g t t t a c a a g c</td>
<td>a a c a a g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>a c a a g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>c a a g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>a a g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>a g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>g t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>t t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>t t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>t a c a a g c</td>
</tr>
<tr>
<td></td>
<td>a c a a g c</td>
</tr>
<tr>
<td></td>
<td>c a a g c</td>
</tr>
<tr>
<td></td>
<td>a a g c</td>
</tr>
<tr>
<td></td>
<td>a g c</td>
</tr>
<tr>
<td></td>
<td>g c</td>
</tr>
<tr>
<td></td>
<td>c</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;
}
```

Linear time and linear space vs. quadratic time and quadratic space.
Longest common prefix

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

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

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

**Digital key.** Sequence of digits over fixed alphabet.

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

<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>
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>N² / 2</td>
<td>N² / 4</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 N lg N *</td>
<td>1.39 N lg N</td>
<td>c lg N</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>2 N lg N</td>
<td>2 N lg N</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

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

Q. Can we do better (despite the lower bound)?
A. Yes, if we don't depend on key compares.

* probabilistic
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.
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

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

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
**Goal.** Sort an array `a[]` of `N` integers between 0 and `R - 1`.

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

```
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 $a[]$ 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];
```
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];
```

<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></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>4</td>
<td></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></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td>f</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></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
**Goal.** Sort an array \( a[] \) of \( N \) integers between 0 and \( R - 1 \).

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Goal. Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Goal. Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
**Goal.** Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.

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

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
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>1</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>e</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>12</td>
</tr>
<tr>
<td>-</td>
<td>12</td>
</tr>
</tbody>
</table>
```

move items

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

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Goal. Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.

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

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Key-indexed counting demo

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

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

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
**Goal.** Sort an array \( a[] \) of \( N \) integers between 0 and \( R - 1 \).

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
Goal. Sort an array \( a[] \) of \( N \) integers between 0 and \( R - 1 \).

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

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

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

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

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

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```
**Goal.** Sort an array \( a[\cdot] \) 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];
```

<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>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

Move items
**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: analysis

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

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

Stable? ✔

| a[0] | Anderson | 2 | Harris | 1 | aux[0] |
| a[12] | Robinson | 2 | Taylor | 3 | aux[12] |
| a[16] | Thompson | 4 | Smith  | 4 | aux[16] |
| a[18] | Williams | 3 | Thompson | 4 | aux[18] |
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 b</td>
<td>0 d a b</td>
<td>0 a c e</td>
</tr>
<tr>
<td>1 a d d</td>
<td>1 c a b</td>
<td>1 a d d</td>
</tr>
<tr>
<td>2 c a b</td>
<td>2 e b b</td>
<td>2 f 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 e e</td>
<td>4 f a d</td>
<td>4 d a d</td>
</tr>
<tr>
<td>5 b a d</td>
<td>5 b a d</td>
<td>5 e b b</td>
</tr>
<tr>
<td>6 d a d</td>
<td>6 d a d</td>
<td>6 a c e</td>
</tr>
<tr>
<td>7 b e e</td>
<td>7 f e d</td>
<td>7 a d d</td>
</tr>
<tr>
<td>8 f e d</td>
<td>8 b e d</td>
<td>8 f e d</td>
</tr>
<tr>
<td>9 b e d</td>
<td>9 f e e</td>
<td>9 b e d</td>
</tr>
<tr>
<td>10 e b b</td>
<td>10 b e e</td>
<td>10 f e e</td>
</tr>
<tr>
<td>11 a c e</td>
<td>11 a c e</td>
<td>11 b e e</td>
</tr>
</tbody>
</table>

Sort must be stable
(arrows do not cross)
Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

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

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

<table>
<thead>
<tr>
<th>Pass</th>
<th>Key</th>
<th>Pass</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d a b</td>
<td>0</td>
<td>a c e</td>
</tr>
<tr>
<td>1</td>
<td>c a b</td>
<td>1</td>
<td>a d d</td>
</tr>
<tr>
<td>2</td>
<td>f a d</td>
<td>2</td>
<td>b a d</td>
</tr>
<tr>
<td>3</td>
<td>b a d</td>
<td>3</td>
<td>b e d</td>
</tr>
<tr>
<td>4</td>
<td>d a d</td>
<td>4</td>
<td>b e e</td>
</tr>
<tr>
<td>5</td>
<td>e b b</td>
<td>5</td>
<td>c a b</td>
</tr>
<tr>
<td>6</td>
<td>a c e</td>
<td>6</td>
<td>d a b</td>
</tr>
<tr>
<td>7</td>
<td>a d d</td>
<td>7</td>
<td>d a d</td>
</tr>
<tr>
<td>8</td>
<td>f e d</td>
<td>8</td>
<td>e b b</td>
</tr>
<tr>
<td>9</td>
<td>b e d</td>
<td>9</td>
<td>f a d</td>
</tr>
<tr>
<td>10</td>
<td>f e e</td>
<td>10</td>
<td>f e d</td>
</tr>
<tr>
<td>11</td>
<td>b e e</td>
<td>11</td>
<td>f e e</td>
</tr>
</tbody>
</table>
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]++; // key-indexed counting for each digit from right to left
         for (int r = 0; r < R; r++)
            count[r+1] += count[r]; // radix R
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i]; // do key-indexed counting for each digit from right to left
      }

      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^2 / 2)</td>
<td>(N^2 / 4)</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N)</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 (N \log N) *</td>
<td>1.39 (N \log N)</td>
<td>(c \log N)</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>(2N \log N)</td>
<td>(2N \log N)</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>2 (W N)</td>
<td>2 (W N)</td>
<td>(N + R)</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

\* probabilistic
† fixed-length \(W\) keys

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

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

Which sorting method to use?

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

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

Problem. Sort one million 32-bit integers.
Ex. Google (or presidential) interview.

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.
String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.
Ex. Supercomputer sort, internet router.

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

Divide each word into eight 16-bit “chars”
$2^{16} = 65,536$ counters
LSD sort on leading 32 bits in 2 passes
Finish with insertion sort
Examines only ~25% of the data
How to take a census in 1900s?

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

Herman Hollerith. Developed counting and sorting machine to automate.

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

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

Punch cards. [1900s to 1950s]

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

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

IBM 80 Series Card Sorter (650 cards per minute)
LSD string sort: a moment in history (1960s)

To sort a card deck
- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted

card punch  punched cards  card reader  mainframe  line printer

card sorter
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
**Most-significant-digit-first string sort**

**MSD string (radix) sort.**

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

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

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

**sort key**

**count[]**

```
0
1
2
3
4
5
6
7
8
9
10
11
```

**sort subarrays recursively**
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).

<table>
<thead>
<tr>
<th></th>
<th>sea</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sea</td>
<td>shells -1</td>
</tr>
<tr>
<td>2</td>
<td>ells</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>he</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>he</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>shells</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>shore</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>surely</td>
<td>-1</td>
</tr>
</tbody>
</table>

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, 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);
}

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

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

Number of characters examined.

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

compareTo() based sorts can also be sublinear!

<table>
<thead>
<tr>
<th>Random (sublinear)</th>
<th>Non-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E10402</td>
<td>are</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1HYL490</td>
<td>by</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1R0Z572</td>
<td>sea</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2HXE734</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2IYE230</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2X0R846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>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>

Characters examined by MSD string sort
## Summary of the performance of sorting algorithms

### Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 N \lg N$ *</td>
<td>$1.39 N \lg N$</td>
<td>$c \lg N$</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 N \lg N$</td>
<td>$2 N \lg N$</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>$2 N W$</td>
<td>$2 N W$</td>
<td>$N + R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>$2 N W$</td>
<td>$N \log R N$</td>
<td>$N + D R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

* probabilistic
† fixed-length W keys
‡ average-length W keys
Disadvantages of MSD string sort.
• Accesses memory "randomly" (cache inefficient).
• Inner loop has a lot of instructions.
• Extra space for `count[]`.
• Extra space for `aux[]`.

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

Goal. Combine advantages of MSD and quicksort.
String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the $d^{th}$ character.

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

![Diagram of 3-way string quicksort](image)
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)
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

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

---

**Fast Algorithms for Sorting and Searching Strings**

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort, it is competitive with the best known C sort codes. The searching algorithm blends tries and binary that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.
3-way string quicksort vs. MSD string sort

MSD string sort.
- Is cache-inefficient.
- Too much memory storing $\text{count}[]$.
- Too much overhead reinitializing $\text{count}[]$ and $\text{aux}[]$.

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

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

## Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>1</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 \times N \log N$ *</td>
<td>$1.39 \times N \log N$</td>
<td>$c \log N$</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 \times N \log N$</td>
<td>$2 \times N \log N$</td>
<td>1</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD \†</td>
<td>$2 \times N \times W$</td>
<td>$2 \times N \times W$</td>
<td>$N + R$</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD \‡</td>
<td>$2 \times N \times W$</td>
<td>$N \log_R N$</td>
<td>$N + D \times R$</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>$1.39 \times W \times N \log N$ *</td>
<td>$1.39 \times W \times N \log N$</td>
<td>$\log N + W$</td>
<td>no</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

input string

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

form suffixes

0 a a c a a g t t t t a c a a g c
1 a c a a g t t t t a c a a g c
2 c a a g t t t t a c a a g c
3 a a g t t t t a c a a g c
4 a g t t t t a c a a g c
5 g t t t t a c a a g c
6 t t t a c a a g c
7 t t a c a a g c
8 t a c a a g c
9 a c a a g c
10 c a a g c
11 a a g c
12 a g c
13 g c
14 c

sort suffixes to bring repeated substrings together

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

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

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

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


Mary Had a Little Lamb

Bach's Goldberg Variations
Longest repeated substring

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

Brute-force algorithm.

- Try all indices $i$ and $j$ for start of possible match.
- Compute longest common prefix (LCP) for each pair.

Analysis. Running time $\leq D N^2$, where $D$ is length of longest match.
Longest repeated substring: a sorting solution

input string

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

form suffixes

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

sort suffixes to bring repeated substrings together

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

compute longest prefix between adjacent suffixes

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

create suffixes (linear time and space)

sort suffixes

find LCP between adjacent suffixes in sorted order

% java LRS < mobydict.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
Problem. Five scientists $A$, $B$, $C$, $D$, and $E$ are looking for long repeated substring in a genome with over 1 billion nucleotides.

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

✓ $E$ uses suffix sorting solution with 3-way string quicksort. but only if LRS is not long (!)

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

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>brute</th>
<th>suffix sort</th>
<th>length of LRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRS.java</td>
<td>2,162</td>
<td>0.6 sec</td>
<td>0.14 sec</td>
<td>73</td>
</tr>
<tr>
<td>amendments.txt</td>
<td>18,369</td>
<td>37 sec</td>
<td>0.25 sec</td>
<td>216</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>191,945</td>
<td>1.2 hours</td>
<td>1.0 sec</td>
<td>58</td>
</tr>
<tr>
<td>mobydicke.txt</td>
<td>1.2 million</td>
<td>43 hours †</td>
<td>7.6 sec</td>
<td>79</td>
</tr>
<tr>
<td>chromosomel1.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>0  twins wins</td>
<td>9  ins</td>
</tr>
<tr>
<td>1  wins twins</td>
<td>8  instwins</td>
</tr>
<tr>
<td>2  instwins</td>
<td>7  ns</td>
</tr>
<tr>
<td>3  nstwins</td>
<td>6  nstwins</td>
</tr>
<tr>
<td>4  stwins</td>
<td>5  s</td>
</tr>
<tr>
<td>5  twins</td>
<td>4  stwins</td>
</tr>
<tr>
<td>6  wins</td>
<td>3  twins</td>
</tr>
<tr>
<td>7  ins</td>
<td>2  twinstwins</td>
</tr>
<tr>
<td>8  ns</td>
<td>1  wins</td>
</tr>
<tr>
<td>9  s</td>
<td>0  wins twins</td>
</tr>
</tbody>
</table>

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

**Running time.** Quadratic (or worse) in the length of the longest match.
Problem. Suffix sort an arbitrary string of length $N$.

Q. What is worst-case running time of best algorithm for problem?
• Quadratic.
✓ • Linearithmic.  
✓ • Linear.  
• Nobody knows.

✓ Manber's algorithm
✓ suffix trees (beyond our scope)
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

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

done

key-indexed counting sort (first character)

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

sorted
### Linearithmic suffix sort example: phase 1

#### original suffixes

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

#### index sort (first two characters)

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

sorted
### Linearithmic suffix sort example: phase 2

#### original suffixes

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

#### index sort (first four characters)

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

The sorted index sort (first four characters) is shown in the shaded boxes.
### Linearithmic suffix sort example: phase 3

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

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

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

SO suffixes_{8}[9] ≤ suffixes_{8}[0]
## Suffix sort: experimental results

### time to suffix sort (seconds)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobyduck.txt</th>
<th>aesopaeop.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36.000 †</td>
<td>4000 †</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td>LSD</td>
<td>not fixed length</td>
<td>not fixed length</td>
</tr>
<tr>
<td>MSD</td>
<td>395</td>
<td>out of memory</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>2.8</td>
<td>400</td>
</tr>
<tr>
<td>Manber MSD</td>
<td>17</td>
<td>8.5</td>
</tr>
</tbody>
</table>

† estimated
String sorting summary

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

We can develop sublinear-time sorts.
- Should measure amount of data in keys, not number of keys.
- Not all of the data has to be examined.

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

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

- String sorts
- Tries
  - R-way tries
  - Ternary search tries
  - Character-based operations
Review: summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

<table>
<thead>
<tr>
<th>implementation</th>
<th>typical case</th>
<th>ordered operations</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>hash table</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</td>
</tr>
</tbody>
</table>

• † under uniform hashing assumption

Q. Can we do better?
A. Yes, if we can avoid examining the entire key, as with string sorting.
## String symbol table basic API

String symbol table. Symbol table specialized to string keys.

```java
public class StringST<

    StringST()
        create an empty symbol table

    void put(String key, Value val)
        put key-value pair into the symbol table

    Value get(String key)
        return value paired with given key

    void delete(String key)
        delete key and corresponding value

    ;
```

### Goal.
Faster than hashing, more flexible than BSTs.
## String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c lg^2 N</td>
<td>c lg^2 N</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

### Parameters
- N = number of strings
- L = length of string
- R = radix

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>words</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>moby.txt</td>
<td>1.2 MB</td>
<td>210 K</td>
<td>32 K</td>
</tr>
<tr>
<td>actors.txt</td>
<td>82 MB</td>
<td>11.4 M</td>
<td>900 K</td>
</tr>
</tbody>
</table>

### Challenge
Efficient performance for string keys.
- R-way tries
- Ternary search tries
- Character-based operations
Tries. [from retrieval, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has \( R \) children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

```
key | value
---|---
by  | 4
sea | 6
sells | 1
she | 0
shells | 3
shore | 7
the | 5
```
Search in a trie

Follow links corresponding to each character in the key.

- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach a null link or node where search ends has null value.

get("shells")
Search in a trie

Follow links corresponding to each character in the key.

- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

get("she")

```
  b
 /  
\  /
 a
 /  
\  
 l
 / 
 s
```

Search may terminated at an intermediate node (return 0)
Search in a trie

Follow links corresponding to each character in the key.
- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach a null link or node where search ends has null value.

get("shell")

![Trie diagram]

- b
  - y
  - e
    - a
    - l
      - s
      - l
        - s
          - e

- s
- h
  - e
  - o
    - e

- t
  - h

No value associated with last key character (return null)
Search in a trie

Follow links corresponding to each character in the key.
- **Search hit**: node where search ends has a non-null value.
- **Search miss**: reach a null link or node where search ends has null value.

get("shelter")
Insertion into a trie

Follow links corresponding to each character in the key.
- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

```plaintext
put("shore", 7)
```
Trie construction demo

trie
Trie construction demo

`put("she", 0)`

Key is sequence of characters from root to value.

Value is in node corresponding to last character.
Trie construction demo
Trie construction demo
Trie construction demo

put("sells", 1)
Trie construction demo
Trie construction demo

trie

![Trie Diagram]

The diagram above illustrates a Trie data structure with the following words:

- She
- Sell
- Else
- Elle

The Trie is constructed by inserting these words, with each node representing a character in the words.
Trie construction demo

put("sea", 2)
Trie construction demo
Trie construction demo

put("shells", 3)
Trie construction demo

trie

```
trie
```

```
trie
```

```
trie
```
Trie construction demo

```
put("by", 4)
```
Trie construction demo
Trie construction demo

put("the", 5)
Trie construction demo
put("sea", 6)

Trie construction demo

overwrite old value with new value
Trie construction demo
Trie construction demo

deep copy
Trie construction demo

put("shore", 7)
Trie construction demo
Trie representation: Java implementation

Node. A value, plus references to $R$ nodes.

```java
private static class Node
{
    private Object value;
    private Node[] next = new Node[R];
}
```

- Trie representation: each node has an array of links and a value.
- Characters are implicitly defined by link index.
- Neither keys nor characters are explicitly stored.
- Use `Object` instead of `Value` since no generic array creation in Java.
public class TrieST<Value> {
    private static final int R = 256;
    private Node root;

    private static class Node {
        /* see previous slide */
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put(Node x, String key, Value val, int d) {
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d + 1);
        return x;
    }
    :

}
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{  
    Node x = get(root, key, 0);
    if (x == null) return null;
    return (Value) x.val;  
}

private Node get(Node x, String key, int d)
{  
    if (x == null) return null;
    if (d == key.length()) return x;
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
}
Trie performance

Search hit. Need to examine all $L$ characters for equality.

Search miss.
- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. $R$ null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

Bottom line. Fast search hit and even faster search miss, but wastes space.
To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If that node has all null links, remove that node (and recur).

```
delete("shells")
```
**String symbol table implementations cost summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>moby.txt</td>
</tr>
<tr>
<td></td>
<td>search miss</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td></td>
</tr>
<tr>
<td></td>
<td>space (references)</td>
<td></td>
</tr>
<tr>
<td>red-black BST</td>
<td>$L + c \log^2 N$</td>
<td>1.40</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>$L$</td>
<td>0.76</td>
</tr>
<tr>
<td>R-way trie</td>
<td>$L$</td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td>$\log R N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$L$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(R+1) N$</td>
<td></td>
</tr>
</tbody>
</table>

**R-way trie.**

- Method of choice for small $R$.
- Too much memory for large $R$.

**Challenge.** Use less memory, e.g., 65,536-way trie for Unicode!
Digression: out of memory?

“640 K ought to be enough for anybody.”
— (mis)attributed to Bill Gates, 1981
(commenting on the amount of RAM in personal computers)

“64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance.” — Windows XP manual, 2002

“64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can't think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly.” — Bill Gates, 2003
Digression: out of memory?

A short (approximate) history.

<table>
<thead>
<tr>
<th>machine</th>
<th>year</th>
<th>address bits</th>
<th>addressable memory</th>
<th>typical actual memory</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDP-8</td>
<td>1960s</td>
<td>12</td>
<td>6 KB</td>
<td>6 KB</td>
<td>$16K</td>
</tr>
<tr>
<td>PDP-10</td>
<td>1970s</td>
<td>18</td>
<td>256 KB</td>
<td>256 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>IBM S/360</td>
<td>1970s</td>
<td>24</td>
<td>4 MB</td>
<td>512 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>VAX</td>
<td>1980s</td>
<td>32</td>
<td>4 GB</td>
<td>1 MB</td>
<td>$1M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1990s</td>
<td>32</td>
<td>4 GB</td>
<td>1 GB</td>
<td>$1K</td>
</tr>
<tr>
<td>Xeon</td>
<td>2000s</td>
<td>64</td>
<td>enough</td>
<td>4 GB</td>
<td>$100</td>
</tr>
<tr>
<td>??</td>
<td>future</td>
<td>128+</td>
<td>enough</td>
<td>enough</td>
<td>$1</td>
</tr>
</tbody>
</table>

“512-bit words ought to be enough for anybody.”
— Kevin Wayne, 1995
R-way tries
Ternary search tries
Character-based operations
Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort, it is competitive with the best known C sort codes. The searching algorithm blends tries and binary search trees, it is faster than hashing and other commonly used search methods. The basic ideas behind the algo-

that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,
Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

TST representation of a trie
Search in a TST

Follow links corresponding to each character in the key.
- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

Search hit. Node where search ends has a non-null value.
Search miss. Reach a null link or node where search ends has null value.
Search in a TST

get("sea")

return value associated with last key character
get("shelter")

```java
get("shelter")
```

```
no link to 't' (return null)
```
Ternary search trie insertion demo

ternary search trie
put("she", 0)

key is sequence of characters from root to value using middle links
value is in node corresponding to last character
put("she", 0)
put("sells", 1)
ternary search trie
put("sea", 2)
Ternary search trie insertion demo

ternary search trie
Ternary search trie insertion demo

put("shells", 3)
Ternary search trie insertion demo

ternary search trie
Ternary search trie insertion demo

put("by", 4)
Ternary search trie insertion demo
put("the", 5)
Ternary search trie insertion demo

ternary search trie

```
  s
 /   \\
 b     t
 /   /   \\
 y     h   e
 /   /   /   \\
 a   l   e   h
 /   /   /   /   \\
 2   l   e   e   5
 /   /   /   /   /   \\
 s   l   l   s   1   3
```

149
Ternary search trie insertion demo

put("sea", 6)

overwrite old value with new value
Ternary search trie insertion demo

ternary search trie
Ternary search trie insertion demo

put("shore", 7)
Ternary search trie insertion demo

ternary search trie

Diagram of a ternary search trie with nodes labeled with letters and numbers to represent the insertion process.
Ternary search trie insertion demo

ternary search trie

```
   s
  /   
 b     t
 /     /
 y     h
  |     /
 4   e   h
  |   /   |
  l   e   e
  /   /   5
 s l o r e
 / / / /
 a l l r e
 / / / / 7
 s l l e
```


26-way trie vs. TST

26-way trie. 26 null links in each leaf.

TST. 3 null links in each leaf.
A TST node is five fields:

- A value.
- A character $c$.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```java
private class Node {
    private Value val;
    private char c;
    private Node left, mid, right;
}
```
public class TST<Value> {
    private Node root;

    private class Node {
        /* see previous slide */
    }

    public void put(String key, Value val) {
        root = put(root, key, val, 0);
    }

    private Node put(Node x, String key, Value val, int d) {
        char c = key.charAt(d);
        if (x == null) { x = new Node(); x.c = c; }
        if (c < x.c) x.left = put(x.left, key, val, d);
        else if (c > x.c) x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1) x.mid = put(x.mid, key, val, d+1);
        else x.val = val;
        return x;
    }

    // ...
}
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{  
  Node x = get(root, key, 0);
  if (x == null) return null;
  return x.val;
}

private Node get(Node x, String key, int d)
{  
  if (x == null) return null;
  char c = key.charAt(d);
  if      (c < x.c)              return get(x.left,  key, d);
  else if (c > x.c)              return get(x.right, key, d);
  else if (d < key.length() - 1) return get(x.mid,   key, d+1);
  else                           return x;
}
## String symbol table implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>(L + c \lg^2 N)</td>
<td>(c \lg^2 N)</td>
</tr>
<tr>
<td>hashing (linear probing)</td>
<td>(L)</td>
<td>(L)</td>
</tr>
<tr>
<td>R-way trie</td>
<td>(L)</td>
<td>(\log R N)</td>
</tr>
<tr>
<td>TST</td>
<td>(L + \ln N)</td>
<td>(\ln N)</td>
</tr>
</tbody>
</table>

**Remark.** Can build balanced TSTs via rotations to achieve \(L + \log N\) worst-case guarantees.

**Bottom line.** TST is as fast as hashing (for string keys), space efficient.
TST vs. hashing

Hashing.
• Need to examine entire key.
• Search hits and misses cost about the same.
• Performance relies on hash function.
• Does not support ordered symbol table operations.

TSTs.
• Works only for strings (or digital keys).
• Only examines just enough key characters.
• Search miss may involve only a few characters.
• Supports ordered symbol table operations (plus others!).

Bottom line. TSTs are:
• Faster than hashing (especially for search misses).
  More flexible than red-black BSTs. [stay tuned]
Tries

- R-way tries
- Ternary search tries
- Character-based operations
Character-based operations. The string symbol table API supports several useful character-based operations.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>by</td>
<td>4</td>
</tr>
<tr>
<td>sea</td>
<td>6</td>
</tr>
<tr>
<td>sells</td>
<td>1</td>
</tr>
<tr>
<td>she</td>
<td>0</td>
</tr>
<tr>
<td>shells</td>
<td>3</td>
</tr>
<tr>
<td>shore</td>
<td>7</td>
</tr>
<tr>
<td>the</td>
<td>5</td>
</tr>
</tbody>
</table>

Prefix match. Keys with prefix "sh": "she", "shells", and "shore".

Wildcard match. Keys that match ".he": "she" and "the".

Longest prefix. Key that is the longest prefix of "shellsort": "shells".
String symbol table API

public class StringST<Value>

- StringST()  
  create a symbol table with string keys

- void put(String key, Value val)  
  put key-value pair into the symbol table

- Value get(String key)  
  value paired with key

- void delete(String key)  
  delete key and corresponding value

- Iterable<String> keys()  
  all keys

- Iterable<String> keysWithPrefix(String s)  
  keys having s as a prefix

- Iterable<String> keysThatMatch(String s)  
  keys that match s (where . is a wildcard)

- String longestPrefixOf(String s)  
  longest key that is a prefix of s

Remark. Can also add other ordered ST methods, e.g., floor() and rank().
Warmup: ordered iteration

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```java
keysWithPrefix(""");
```

```
key  q
  b
  by
  s
  se
sea  by sea
sel
sell
sells  by sea sells
  sh
  she  by sea sells she
shell
shells  by sea sells she shells
  sho
shor
shore  by sea sells she shells shore
  t
  th
the  by sea sells she shells shore the
```
To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```java
public Iterable<String> keys()
{
    Queue<String> queue = new Queue<String>();
    collect(root, "", queue);
    return queue;
}

private void collect(Node x, String prefix, Queue<String> q)
{
    if (x == null) return;
    if (x.val != null) q.enqueue(prefix);
    for (char c = 0; c < R; c++)
        collect(x.next[c], prefix + c, q);
}
```
Prefix matches

Find all keys in symbol table starting with a given prefix.

Ex. Autocomplete in a cell phone, search bar, text editor, or shell.
• User types characters one at a time.
• System reports all matching strings.
Find all keys in symbol table starting with a given prefix.

```java
public Iterable<String> keysWithPrefix(String prefix) {
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}
```
## Wildcard matches

Use wildcard `. ` to match any character in alphabet.

<table>
<thead>
<tr>
<th>co....er</th>
<th>.c...c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>coalizer</td>
<td>acresce</td>
</tr>
<tr>
<td>coberger</td>
<td>acroach</td>
</tr>
<tr>
<td>codifier</td>
<td>acuracy</td>
</tr>
<tr>
<td>cofaster</td>
<td>octarch</td>
</tr>
<tr>
<td>cofather</td>
<td>science</td>
</tr>
<tr>
<td>cognizer</td>
<td>scranch</td>
</tr>
<tr>
<td>cohelper</td>
<td>scratch</td>
</tr>
<tr>
<td>colander</td>
<td>scrauch</td>
</tr>
<tr>
<td>coleader</td>
<td>screich</td>
</tr>
<tr>
<td>...</td>
<td>scrinch</td>
</tr>
<tr>
<td>compiler</td>
<td>scritch</td>
</tr>
<tr>
<td>...</td>
<td>scrunch</td>
</tr>
<tr>
<td>composer</td>
<td>scudick</td>
</tr>
<tr>
<td>computer</td>
<td>scutock</td>
</tr>
<tr>
<td>cowkeeper</td>
<td></td>
</tr>
</tbody>
</table>
Search as usual if character is not a period; go down all $R$ branches if query character is a period.

```java
public Iterable<String> keysThatMatch(String pat)
{
    Queue<String> queue = new Queue<String>();
    collect(root, "", 0, pat, queue);
    return queue;
}

private void collect(Node x, String prefix, String pat, Queue<String> q)
{
    if (x == null) return;
    int d = prefix.length();
    if (d == pat.length() && x.val != null) q.enqueue(prefix);
    if (d == pat.length()) return;
    char next = pat.charAt(d);
    for (char c = 0; c < R; c++)
        if (next == '.' || next == c)
            collect(x.next[c], prefix + c, pat, q);
}
```
Longest prefix

Find longest key in symbol table that is a prefix of query string.

**Ex.** To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

```
"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"
```

Note. Not the same as floor: \( \text{floor}("128.112.100.16") = "128.112.055.15" \)
Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.

**Possibilities for longestPrefixOf()**

- `"she"` search ends at end of string value is not null return she

- `"shell"` search ends at end of string value is null return she (last key on path)

- `"shellsort"` search ends at null link return shells (last key on path)
Longest prefix: Java implementation

Find longest key in symbol table that is a prefix of query string.
- Search for query string.
- Keep track of longest key encountered.

```java
public String longestPrefixOf(String query) {
    int length = search(root, query, 0, 0);
    return query.substring(0, length);
}

private int search(Node x, String query, int d, int length) {
    if (x == null) return length;
    if (x.val != null) length = d;
    if (d == query.length()) return length;
    char c = query.charAt(d);
    return search(x.next[c], query, d+1, length);
}
```
**T9 texting**

**Goal.** Type text messages on a phone keypad.

**Multi-tap input.** Enter a letter by repeatedly pressing a key until the desired letter appears.

"a much faster and more fun way to enter text"

**T9 text input.**
- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

**Ex.** hello
- Multi-tap: 4 4 3 3 5 5 5 5 5 5 6 6 6
- T9: 4 3 5 5 6

www.t9.com
To: info@t9support.com  
Date: Tue, 25 Oct 2005 14:27:21 -0400 (EDT)

Dear T9 texting folks,

I enjoyed learning about the T9 text system from your webpage, and used it as an example in my data structures and algorithms class. However, one of my students noticed a bug in your phone keypad:

http://www.t9.com/images/how.gif

Somehow, it is missing the letter s. (!)

Just wanted to bring this information to your attention and thank you for your website.

Regards,

Kevin
Thank you Kevin.

I am glad that you find T9 o valuable for your cli. I had not noticed thi before. Thank for writing in and letting u know.

Take care,

Brooke nyder
OEM Dev upport
AOL/Tegic Communication
1000 Dexter Ave N. uite 300
eattle, WA 98109

ALL INFORMATION CONTAINED IN THIS EMAIL IS CONSIDERED CONFIDENTIAL AND PROPERTY OF AOL/TEGIC COMMUNICATIONS
Patricia trie. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

Applications.

- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

Also known as: crit-bit tree, radix tree.
Suffix tree

- Patricia trie of suffixes of a string.
- Linear-time construction: beyond this course.

suffix tree for BANANAS

Applications.
- Linear-time: longest repeated substring, longest common substring, longest palindromic substring, substring search, tandem repeats, ….
- Computational biology databases (BLAST, FASTA).
String symbol tables summary

A success story in algorithm design and analysis.

Red-black BST.
• Performance guarantee: \( \log N \) key compares.
• Supports ordered symbol table API.

Hash tables.
• Performance guarantee: constant number of probes.
• Requires good hash function for key type.

Tries. R-way, TST.
• Performance guarantee: \( \log N \) characters accessed.
• Supports character-based operations.

Bottom line. You can get at anything by examining 50-100 bits (!!!)