5. Strings

- 5.1 Sorting Strings
- 5.2 String Symbol Tables
- 5.3 Substring Search
- 5.4 Pattern Matching
- 5.5 Data Compression
String processing

String. Sequence of characters.

Important fundamental abstraction.

• Java programs.
• Natural languages.
• Genomic sequences.
• ...

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

**C char data type.** Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Need more bits to represent certain characters.

**Java char data type.** A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Awkwardly supports 21-bit Unicode 3.0.
The String data type

**Character extraction.** Get the \(i^{th}\) character.

**Substring extraction.** Get a contiguous sequence of characters from a string.

**String concatenation.** Append one character to end of another string.

```java
String s = "strings"; // s = "strings"
char c = s.charAt(2); // c = 'r'
String t = s.substring(2, 6); // t = "ring"
String u = t + c; // u = "ringr"
```
Implementing strings in Java

Java strings are immutable ⇒ two strings can share underlying char[] array.

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

    private String(int offset, int count, char[] value)
    {
        this.offset = offset;
        this.count  = count;
        this.value  = value;
    }

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

    public char charAt(int index)
    {  return value[index + offset];  }

    ...  
}
```

java.lang.String

constant time
Implementing strings in Java

```java
public String concat(String that)
{
    char[] buffer = new char[this.length() + that.length());
    for (int i = 0; i < this.length(); i++)
        buffer[i] = this.value[i];
    for (int j = 0; j < that.length(); j++)
        buffer[this.length() + j] = that.value[j];
    return new String(0, this.length() + that.length(), buffer);
}
```

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

use byte[] or char[] instead of String to save space

<table>
<thead>
<tr>
<th>operation</th>
<th>guarantee</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<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>
String VS. StringBuilder

**String.** [immutable] Constant substring, linear concatenation.

**StringBuilder.** [mutable] Linear substring, constant (amortized) append.

Ex.  Reverse a String.

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

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

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

**Challenge.** How to efficiently form array of suffixes?

**input string**

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

**suffixes**

```
  0 a a c a a g t t t a c a a g c
  1 a c a a g t t t a c a a g c
  2 c a a g t t t a c a a g c
  3 a a g t t t a c a a g c
  4 a g t t t a c a a g c
  5 g t t t a c a a g c
  6 t t t a c a a g c
  7 t t a c a a g c
  8 t a c a a g c
  9 a c a a g c
 10 c a a g c
 11 a a g c
 12 a g c
 13 g c
 14 c
```
String challenge: array of suffixes

Challenge. 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 space quadratic time and space!
**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>lgR()</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>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>EXTENDED_ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

Standard alphabets
6.1 Sorting Strings

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays
Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

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

* probabilistic

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

Q. Can we do better (despite the lower bound)?
A. Yes, if we don't depend on compares.
‐ key-indexed counting
‐ LSD string sort
‐ MSD string sort
‐ 3-way radix quicksort
‐ longest repeated substring
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.

**Remark.** Keys may have associated data ⇒ can’t just count up number of keys of each value.
**Goal.** Sort an array \( a[] \) of \( N \) integers between 0 and \( R-1 \).

- Count frequencies of each letter using key as index.
- 
- 

```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>

### Key-indexed counting

offset by 1
[stay tuned]
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.
- 

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

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

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

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

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

6 keys < a, 8 keys < e so d’s go in \( a[6] \) and \( a[7] \)
**Key-indexed counting**

**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 records.

```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 records.

```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

**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 records.

```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></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></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td></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>
<tr>
<td></td>
<td>r</td>
<td>count[r]</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>12</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 records.

```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 records.

```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 records.

```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

**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 records.

```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

**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 records.

```
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

**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 records.

```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 records.

```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

**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 records.

```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>i</th>
<th>$aux[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td></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>f</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>

---
Key-indexed counting

**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 records.

```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 records.

```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

**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 records.

```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

**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 records.
- 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 takes time proportional to \( N + R \) to sort \( N \) records whose keys are integers between 0 and \( R-1 \).

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

**Stable?** Yes!
key-indexed counting
LSD string sort
MSD string sort
3-way string quicksort
suffix arrays
Least-significant-digit-first radix sort

LSD string sort.

- Consider characters from right to left.
- Stably sort using \(d^{th}\) character as the key (using key-indexed counting).

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

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

**Pf.** [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.

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

**Sort key in order by previous passes:**
- 0: a, c, e
- 1: a, d, d
- 2: b, a, d
- 3: b, e, d
- 4: b, e, e
- 5: c, a, b
- 6: d, a, b
- 7: d, a, d
- 8: e, b, b
- 9: f, e, d
- 10: f, e, e
- 11: f, e, e
LSD string sort: Java implementation

```java
public class LSD {
   public static void sort(String[] a, int W) {
      int R = 256
      int N = a.length;
      String[] aux = new String[N];
      for (int d = W-1; d >= 0; d--)
      {
         int[] count = new int[R+1];
         for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < N; i++)
            a[i] = aux[i];
      }
   }
}
```

- Fixed-length W strings
- Radix R
- Do key-indexed counting for each digit from right to left
- Key-indexed counting
LSD string sort: example

<table>
<thead>
<tr>
<th>Input</th>
<th>d = 6</th>
<th>d = 5</th>
<th>d = 4</th>
<th>d = 3</th>
<th>d = 2</th>
<th>d = 1</th>
<th>d = 0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4PGC938</td>
<td>2IYE230</td>
<td>3CIO720</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>2IYE230</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>3CIO720</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>1ICK750</td>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2IYE230</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>10HV845</td>
<td>3CIO720</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>10HV845</td>
</tr>
<tr>
<td>4JZY524</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>3CIO720</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
</tr>
<tr>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2IYE230</td>
<td>3ATW723</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>3CIO720</td>
<td>10HV845</td>
<td>4PGC938</td>
<td>1ICK750</td>
<td>3CIO720</td>
<td>3CIO720</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>3ATW723</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>10HV845</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>4PGC938</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4PGC938</td>
<td>3CIO720</td>
<td>3CIO720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>3ATW723</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>4PGC938</td>
<td>4JZY524</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
</tr>
</tbody>
</table>
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>$2 W N$</td>
<td>$2 W N$</td>
<td>$N + R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length $W$ keys
Sorting challenge 1

Problem. Sort a huge commercial database on a fixed-length key field.
Ex. Account number, date, SS number, ...

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

256 (or 65536) counters:
Fixed-length strings sort in W passes.
Sorting challenge 2a

Problem. Sort 1 million 32-bit integers.
Ex. Google interview or presidential interview.

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

Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)
- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays
Most-significant-digit-first string sort

**MSD string sort.**

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

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

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

- sort key
- sort these independently (recursive)
**MSD string sort: top level trace**

```
use key-indexed counting on first character

<table>
<thead>
<tr>
<th>count frequencies</th>
<th>transform counts to indices</th>
<th>distribute and copy back</th>
<th>indices at completion of distribute phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>she</td>
<td>0</td>
<td>0</td>
<td>sort(a, 0, 0);</td>
</tr>
<tr>
<td>sells</td>
<td>1</td>
<td>1</td>
<td>sort(a, 1, 1);</td>
</tr>
<tr>
<td>seashells</td>
<td>2</td>
<td>2</td>
<td>sort(a, 2, 1);</td>
</tr>
<tr>
<td>by</td>
<td>3</td>
<td>3</td>
<td>sort(a, 3, 1);</td>
</tr>
<tr>
<td>the sea</td>
<td>4</td>
<td>4</td>
<td>sort(a, 4, 1);</td>
</tr>
<tr>
<td>shore</td>
<td>5</td>
<td>5</td>
<td>sort(a, 5, 1);</td>
</tr>
<tr>
<td>the shells</td>
<td>6</td>
<td>6</td>
<td>sort(a, 6, 1);</td>
</tr>
<tr>
<td>she sells</td>
<td>7</td>
<td>7</td>
<td>sort(a, 7, 1);</td>
</tr>
<tr>
<td>sells surely</td>
<td>8</td>
<td>8</td>
<td>sort(a, 8, 1);</td>
</tr>
<tr>
<td>seashells the</td>
<td>9</td>
<td>9</td>
<td>sort(a, 9, 1);</td>
</tr>
<tr>
<td>the</td>
<td>10</td>
<td>10</td>
<td>sort(a, 10, 1);</td>
</tr>
<tr>
<td>the</td>
<td>11</td>
<td>11</td>
<td>sort(a, 11, 1);</td>
</tr>
<tr>
<td>the</td>
<td>12</td>
<td>12</td>
<td>sort(a, 12, 1);</td>
</tr>
<tr>
<td>the</td>
<td>13</td>
<td>13</td>
<td>sort(a, 13, 1);</td>
</tr>
</tbody>
</table>

```

start of s subarray

1 + end of s subarray
**MSD string sort: example**

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>shell</td>
</tr>
<tr>
<td>2</td>
<td>sells</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>she</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>she</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>shells</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>shore</td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td>surely</td>
<td>y</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.
MSD string sort: Java implementation

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

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

key-indexed counting

can recycle aux[] but not count[]

recursively sort subarrays
**MSD string sort: potential for disastrous performance**

**Observation 1.** Much too slow for small subarrays.
- The `count[]` array must be re-initialized.
- ASCii (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 32,000x slower for N = 2.

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

**Solution.** Cutoff to insertion sort for small N.
Cutoff to insertion sort

Solution. Cutoff to insertion sort for small N.
- 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!

<table>
<thead>
<tr>
<th>Random (sublinear)</th>
<th>Non-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1EI0402</td>
<td>are</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1HYL490</td>
<td>by</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1ROZ572</td>
<td>sea</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2HXE734</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2IYE230</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2X0R846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>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>

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

---

* stack depth $D = \text{length of longest prefix match}$
MSD string sort vs. quicksort for strings

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

Disadvantage of quicksort.
• *Linearithmic number of string compares (not linear).*
• *Has to rescan long keys for compares.*

[but stay tuned]
- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays
3-way string quicksort (Bentley and Sedgewick, 1997)

**Overview.** Do 3-way partitioning on the $d^{th}$ character.
- Cheaper than R-way partitioning of MSD string sort.
- Need not examine again characters equal to the partitioning char.

![Diagram of 3-way string quicksort]

Trace of recursive calls for 3-way string quicksort (no cuto
for small subarrays)

Recursively sort subarrays, excluding first character for "equal" subarray
3-way string quicksort: trace of recursive calls

Trace of recursive calls for 3-way string quicksort (no cutoff for small subarrays)
3-way string quicksort: Java implementation

```java
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

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

3-way partitioning, using d\textsuperscript{th} character

sort 3 pieces recursively
3-way string quicksort vs. standard quicksort

Standard quicksort.
• Uses $2N \ln N$ string compares on average.
• Costly for long keys that differ only at the end (and this is a common case!)

3-way string quicksort.
• Uses $2N \ln N$ character compares on average for random strings.
• Avoids recomparing initial parts of the string.
• Adapts to data: uses just "enough" characters to resolve order.
• Sublinear when strings are long.

Proposition. 3-way string quicksort is optimal (to within a constant factor); no sorting algorithm can (asymptotically) examine fewer chars.

Pf. Ties cost to entropy. Beyond scope of 226.
3-way string quicksort vs. MSD string sort

MSD string sort.
• Has a long inner loop.
• Is cache-inefficient.
• Too much overhead reinitializing `count[]` and `aux[]`.

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

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

### Frequency of operations.

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

* probabilistic
† fixed-length $W$ keys
‡ average-length $W$ keys
key-indexed counting
LSD string sort
MSD string sort
3-way radix quicksort
suffix arrays
Warmup: longest common prefix

**LCP.** Given two strings, find the longest substring that is a prefix of both.

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

**Running time.** Linear-time in length of prefix match.

**Space.** Constant extra space.
Longest repeated substring

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

**Ex.**

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

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


Mary Had a Little Lamb

Bach's Goldberg Variations
Longest repeated substring

**LRS.** 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 $\leq M N^2$, where $M$ is length of longest match.
Longest repeated substring: a sorting solution

Input string:

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

Form suffixes:

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

Sort suffixes to bring repeated substrings together:

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

Compute longest prefix between adjacent suffixes:

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

65
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++)
    {
        String x = lcp(suffixes[i], suffixes[i+1]);
        if (x.length() > lrs.length()) lrs = x;
    }
    return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between suffixes that are adjacent after sorting

```bash
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```
Sorting challenge

Problem. Five scientists A, B, C, D, and E are looking for long repeated substring in a genome with over 1 billion nucleotides.
- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- E uses suffix sorting solution with 3-way string quicksort.

✓ 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>mobydict.txt</td>
<td>1.2 million</td>
<td>43 hours†</td>
<td>7.6 sec</td>
<td>79</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>2 months†</td>
<td>61 sec</td>
<td>12,567</td>
</tr>
<tr>
<td>pi.txt</td>
<td>10 million</td>
<td>4 months†</td>
<td>84 sec</td>
<td>14</td>
</tr>
</tbody>
</table>

† estimated
Suffix sorting: worst-case input

Longest repeated substring not long. Hard to beat 3-way string quicksort.

Longest repeated substring very long.
- Radix sorts are quadratic in the length of the longest match.
- Ex: two copies of Aesop’s fables.

% more abcdefgh2.txt
abcdefgh
abcdefgh
bcdefgh
bcdefgh
bcdefghabcdefgh
bcdefgh
bcdefghabcdefgh
cdefgh
cdefghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
h
h
h
h

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydick.txt</th>
<th>aesopaesop.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>
</tbody>
</table>

† estimated
Suffix sorting challenge

**Problem.** Suffix sort an arbitrary string of length $N$.

**Q.** What is worst-case running time of best algorithm for problem?
- Quadratic.
- Linearithmic. ✓ Manber’s algorithm
- Linear. ✓ suffix trees (see COS 423)
- Nobody knows.
Suffix sorting in linearithmic time

Manber's MSD algorithm.

• 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. \(N \log N\).

• Finishes after \(\lg N\) phases.
• Can perform a phase in linear time. (!) [stay tuned]
Linearithmic suffix sort example: phase 0

original suffixes

key-indexed counting sort (first character)

sorted
### Linearithmic suffix sort example: phase 1

#### Original suffixes

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

#### Index sort (first two characters)

- Sorted
Linearithmic suffix sort example: phase 2

original suffixes

index sort (first four characters)

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 a b c b a b a a a a a 0
3  a a a a a b c b a b a a a a a 0
4  a a a b c b a b a a a a a 0
5  a 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

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

sorted
### Linearithmic suffix sort example: phase 3

#### 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></td>
<td>b</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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</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>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</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>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></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>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

#### index sort (first eight characters)

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

---

**FINISHED! (no equal keys)**
Achieve 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>0</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

so suffixes₈[9] ≤ suffixes₈[0]
### Suffix sort: experimental results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydict.txt</th>
<th>aesopaesop.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.
• *Compares not necessary for string keys.*
• *Use digits to index an array.*

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

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

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