5. Strings

- 5.1 Strings Sorts
- 5.2 Tries
- 5.3 Substring Search
- 5.4 Regular Expressions
- 5.5 Data Compression
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).

### Hexadecimal to ASCII conversion table

```
Hexadecimal to ASCII conversion table
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
</tr>
<tr>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
</tr>
<tr>
<td>SP</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
</tr>
<tr>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td>}</td>
<td>~</td>
<td>DEL</td>
</tr>
</tbody>
</table>

- Data Compression
- ASCII encoding.
- When you HexDump a bit-stream that contains ASCII-encoded characters, the table at right is useful for reference.
- Given a 2-digit hex number, use the first hex digit as a row index and the second hex digit as a column reference to find the character that it encodes. For example, `31` encodes the digit `1`, `4A` encodes the letter `J`, and so forth.
- This table is for 7-bit ASCII, so the first hex digit must be `7` or less. Hex numbers starting with `0` and `1` (and the numbers `20` and `7F`) correspond to non-printing control characters. Many of the control characters are left over from the days when physical devices like typewriters were controlled by ASCII input; the table highlights a few that you might see in dumps. For example, `SP` is the space character, `NUL` is the null character, `LF` is line-feed, and `CR` is carriage-return.

- Working with data compression requires us to reorient our thinking about standard input and standard output to include binary encoding of data.
- BinaryStdIn and BinaryStdOut provide the methods that we need. They provide a way for you to make a clear distinction in your client programs between writing out information intended for file storage and data transmission (that will be read by programs) and printing information (that is likely to be read by humans).
The String data type

String data type. Sequence of characters (immutable).

Indexing. 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.
The String data type: Java implementation

```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];
    }

    public String concat(String that) {
        char[] val = new char[this.length() + that.length()];
        ...
        return new String(0, this.length() + that.length(), 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>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 $N$.

use byte[] or char[] instead of String to save space
The StringBuilder data type

**StringBuilder data type.** Sequence of characters (mutable).  
**Underlying implementation.** Doubling char[] array and length.

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

* amortized

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

**Challenge.** How to reverse a string?

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

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

Challenge. How to form array of suffixes?

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
```

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 vs. StringBuilder

Challenge. How to 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;
}
```
## Digital key
Sequence of digits over fixed alphabet.

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

<table>
<thead>
<tr>
<th>name</th>
<th>$R()$</th>
<th>$\lg R()$</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>0123456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0123456789ABCDEF</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>2</td>
<td>ACTG</td>
</tr>
<tr>
<td>LOWERCASE</td>
<td>26</td>
<td>5</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>BASE64</td>
<td>64</td>
<td>6</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>EXTENDED_ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

**Standard alphabets**
5.1 String Sorts

- 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>$1.39 \times N \lg N$</td>
<td>$1.39 \times N \lg N$</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>no</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Lower bound. $\sim N \lg N$ 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 string 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 $\Rightarrow$ can't just count up number of keys of each value.
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.
  
```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

offset by 1
[stay tuned]
Goal. Sort an array $a[]$ of $N$ 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];
```
**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 uses $8N + 3R$ array accesses 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!

**In-place?** No.
key-indexed counting
LSD string sort
MSD string sort
3-way string quicksort
suffix arrays
**Least-significant-digit-first string sort**

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

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

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.
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];
        }
    }
    
}"
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

Q. What if strings do not have same length?
**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 65,536) counters;
Fixed-length strings sort in W passes.
String 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.
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.
LSD string sort: a moment in history (1960s)

- card punch
- punched cards
- card reader
- mainframe
- line printer

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)

not related to sorting
- 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).
**MSD string sort: top-level trace**

- **Use key-indexed counting on first character**
  - Count frequencies
  - Transform counts to indices
  - Distribute and copy back
  - Indexes at completion of distribute phase
  - Recursively sort subarrays

### Example
```
she
sells
seashells
by
the
sea
shore
the
shells
she
sells
are
surely
seashells
0 0 0
1 a 0
2 b 1
3 c 1
4 d 0
5 e 0
6 f 0
7 g 0
8 h 0
9 i 0
10 j 0
11 k 0
12 l 0
13 m 0
14 n 0
15 o 0
16 p 0
17 q 0
18 r 0
19 s 2
20 t 10
21 u 2
22 v 0
23 w 0
24 x 0
25 y 0
26 z 0
27 0
```

- **Start of s subarray:** 10
- **End of s subarray:** 14

```
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).

### 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 $N$.
- 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>2XOR846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CDB573</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CVP720</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3IGJ319</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3KNA382</td>
<td>shells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TAV879</td>
<td>shore</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4CQP781</td>
<td>surely</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4QGI284</td>
<td>the</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4YHV229</td>
<td>the</td>
<td>1DNB377</td>
</tr>
</tbody>
</table>

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>

* stack depth $D = \text{length of longest prefix match}$

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

* probabilistic
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.

Goal. Combine advantages of MSD and quicksort.
- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays
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.
3-way string quicksort: trace of recursive calls

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

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

private static void sort(String[] a, int lo, int hi, int d)
{
  if (hi <= lo) return;
  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 $2N \ln N$ 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>$N \log N$</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>$2 \ N \log N$</td>
<td>$2 \ N \log N$</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>
<tr>
<td>3-way string quicksort</td>
<td>$1.39 \ W \ N \log N$ *</td>
<td>$1.39 \ W \ N \log N$</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 string quicksort
- suffix arrays
Warmup: longest common prefix

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

<table>
<thead>
<tr>
<th>p</th>
<th>r</th>
<th>e</th>
<th>f</th>
<th>e</th>
<th>t</th>
<th>c</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

| p | r | e | f | i | x |

```java
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

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

Ex.

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.

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

Analysis. Running time $\leq MN^2$, 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
```

```
AACGTTTACACAGC
```

**form 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
```

**sort suffixes to bring repeated substrings together**

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

```
A A C A A G T T T A C A A G C
```
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 < mobydict.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.

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>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.
• String sorts are quadratic in the length of the longest match.
• Ex: two copies of Aesop’s fables.

% more abcdefg2.txt
abcdefg
abcdefgabcabcdefg
bcdefg
bcdefghabcabcdefg
cdefg
cdefgabcabcdefg
defg
defgabcabcdefg
efgabcdefg
efg
efgabcdefg
efgabcdefg
efgabcdefg
efgabcdefg
efgabcdefg

time to suffix sort (seconds)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydict.txt</th>
<th>aesop2.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 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**

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>key-indexed counting sort (first character)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  b a b a a a a b c b a b a a a a 0</td>
<td>17  0</td>
</tr>
<tr>
<td>0  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>0  b a a a a a b c b a b a a a a 0</td>
<td>15  a a 0</td>
</tr>
<tr>
<td>0  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>0  a a a b c b a b a a a a 0</td>
<td>13  a a a 0</td>
</tr>
<tr>
<td>0  a a b c b a b a a a a 0</td>
<td>12  a a a a 0</td>
</tr>
<tr>
<td>0  b c b a b a a a a 0</td>
<td>11  a b a a a a a 0</td>
</tr>
<tr>
<td>0  c b a b a a a a 0</td>
<td>10  a b a a a a a 0</td>
</tr>
<tr>
<td>0  b a b a a a a a a 0</td>
<td>9  a b a a a a a a a 0</td>
</tr>
<tr>
<td>0  a a a a a 0</td>
<td>8  b a b a a a a a a a 0</td>
</tr>
<tr>
<td>0  a a a a 0</td>
<td>7  b a b a a a a a a 0</td>
</tr>
<tr>
<td>0  a a 0</td>
<td>6  b c b a b a a a a a 0</td>
</tr>
<tr>
<td>0  a 0</td>
<td>5  b a a a a a 0</td>
</tr>
<tr>
<td>0  0</td>
<td>4  a b a a a a 0</td>
</tr>
<tr>
<td>0  sorted</td>
<td></td>
</tr>
</tbody>
</table>

---

**key-indexed counting sort (first character)**

Sorted original suffixes.
**Linearithmic suffix sort example: phase 1**

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

*sorted*
Linearithmic suffix sort example: phase 2

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

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


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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobyduck.txt</th>
<th>aesopaeosop.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.
• $1.39 N \lg N$ chars for random data.

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