5. Strings

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

The char data type

C char data type. Typically an 8-bit integer.
• Supports 7-bit ASCII.
• Need more bits to represent certain characters.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>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>
</tr>
<tr>
<td>4</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>5</td>
<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>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</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>7</td>
<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>
</tr>
</tbody>
</table>

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.
• Supports original 16-bit Unicode.
• Awkwardly supports 21-bit Unicode 3.0.

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 String data type

Character extraction. Get the i<sup>th</sup> character.
Substring extraction. Get a contiguous sequence of characters from a string.
String concatenation. Append one character to end of another string.

```
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.

```
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];
    }
    ...
}
```

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

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

operation | guarantee | extra space
--- | --- | ---
charAt() | 1 | 1
substring() | 1 | 1
concat() | N | N

String vs. StringBuilder

String. [immutable] Constant substring, linear concatenation.
StringBuilder. [mutable] Linear substring, constant (amortized) append.

Ex. Reverse a String.

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

```
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 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 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
```
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;
}

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;
}

6.1 Sorting Strings

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays

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

Frequency of operations = key compares.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>N² / 2</td>
<td>N² / 4</td>
<td>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 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>no</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

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: 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.

---

**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.

```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 \( R-1 \).
- Compute frequency cumulates which specify destinations.
Key-indexed counting

Goal. Sort an array $a[]$ of $n$ integers between 0 and $n-1$.
- 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 $n-1$.
- 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 \(N-1\).
- 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];
```

```
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 $n-1$.
- 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 $n-1$.
- 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 $n-1$.
- 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];
    a[i] = aux[i];
}
```
LSD string sort
- Consider characters from right to left.
- Stably sort using \( d \)th character as the key (using key-indexed counting).

```
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--)
            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];
    }
}
```

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

\textbf{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--)
            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];
    }
}
```
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.

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>
</tbody>
</table>

LSD $^\dagger$ | $2 W N$ | $2 W N$ | $N + R$ | yes | charAt() |

$^*$ probabilistic
$^\dagger$ fixed-length W keys

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

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).

use key-indexed counting on first character

<table>
<thead>
<tr>
<th>count</th>
<th>transform counts toindices at completionof distribute and copy back</th>
<th>recursively sort subarrays</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>
</tbody>
</table>

sort key

LSD string sort: top level trace

Lysergic Acid Diethylamide
(Lazy in the Sky with Diamonds)
### 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)]++] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];
    sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}

private static int charAt(String s, int d) {
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

### Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).

<table>
<thead>
<tr>
<th>le</th>
<th>el</th>
<th>se</th>
<th>sh</th>
<th>s</th>
<th>e</th>
<th>s</th>
<th>h</th>
<th>e</th>
<th>a</th>
<th>sh</th>
<th>s</th>
<th>e</th>
<th>s</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>6</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>s</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>sh</td>
<td>s</td>
<td>e</td>
<td>s</td>
<td>h</td>
</tr>
</tbody>
</table>

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

### 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;
}
```

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 (linear)</th>
<th>Note-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E0462</td>
<td>aro</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3H01490</td>
<td>by</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1R0572</td>
<td>sea</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2H05734</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2T0E230</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2X0R846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3DC0573</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3C0P720</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3I03319</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3K04382</td>
<td>shells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TAV879</td>
<td>shoe</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4CP0781</td>
<td>surely</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4Q2E284</td>
<td>the</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4V0V229</td>
<td>the</td>
<td>1DNB377</td>
</tr>
</tbody>
</table>

Characters examined by MSD string sort

Summary of the performance of sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2/2$</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>$N \times R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>$2NW$</td>
<td>$N \log_4 N$</td>
<td>$N \times D \times R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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]
3-way string quicksort: trace of recursive calls

- LSD string sort
- MSĐ string sort
- Key-indexed counting

3-way string quicksort

Overview: Do 3-way partitioning on the \(d\)-th character.
- Cheaper than \(R\)-way partitioning of MSĐ string sort.
- Need not examine again characters equal to the partitioning char.

```
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 i = lo;
    if (v >= 0) sort(a, lt, gt, d+1);
    if (v < 0) sort(a, lt, gt, d+1);
    if (v = 0) sort(a, lt, gt, d+1);
    if (v <= 0) sort(a, lt, gt, d+1);
    else {
        sort(a, lo, i-1, d);
        i++;
        sort(a, i, 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><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 $N \lg N^*$</td>
<td>1.39 $N \lg N$</td>
<td>$c \lg N$</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>1</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD †</td>
<td>$2N W$</td>
<td>$2N W$</td>
<td>$N + R$</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD ‡</td>
<td>$2N W$</td>
<td>$N \log N$</td>
<td>$N + D R$</td>
<td>yes</td>
<td><code>charAt()</code></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><code>charAt()</code></td>
</tr>
</tbody>
</table>

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

---

**Additional references:**

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

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

\[ \text{time to suffix sort (seconds)} \]

<table>
<thead>
<tr>
<th>algorithm</th>
<th>subdick.txt</th>
<th>aesop-aesop.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

Linearithmic suffix sort example: phase 0

Manber’s MSD algorithm.
- Phase 0: sort on first character using key-indexed counting sort.
- Phase \( i \): given array of suffixes sorted on first \( 2^i \) 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 1

Problem. Suffix sort an arbitrary string of length \( N \).

Q. What is worst-case running time of best algorithm for problem?
- Quadratic.
- Linear. \( \rightsquigarrow \) radix sorts (see COS 423).
- Nobody knows.

\[ \text{time to suffix sort (seconds)} \]

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydic.txt</th>
<th>aesop_aesop.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

Original suffix:

key-indexed counting sort (first character):

<table>
<thead>
<tr>
<th>( i )</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>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</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>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>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<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>
<td>a</td>
</tr>
<tr>
<td>e</td>
<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>
<td>a</td>
</tr>
<tr>
<td>f</td>
<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>
<td>a</td>
</tr>
<tr>
<td>g</td>
<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>
<td>a</td>
</tr>
<tr>
<td>h</td>
<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>
<td>a</td>
</tr>
</tbody>
</table>

Original suffix:

key-indexed counting sort (first character):

<table>
<thead>
<tr>
<th>( i )</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>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</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>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>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<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>
<td>a</td>
</tr>
<tr>
<td>e</td>
<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>
<td>a</td>
</tr>
<tr>
<td>f</td>
<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>
<td>a</td>
</tr>
<tr>
<td>g</td>
<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>
<td>a</td>
</tr>
<tr>
<td>h</td>
<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>
<td>a</td>
</tr>
</tbody>
</table>
### Linearithmic suffix sort example: phase 1

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

#### Linearithmic suffix sort example: phase 2

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

#### Linearithmic suffix sort example: phase 3

<table>
<thead>
<tr>
<th>original suffix</th>
<th>index sort (first eight characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>1 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>2 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>3 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>4 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>5 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>6 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>7 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>8 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>9 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>10 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>11 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>12 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>13 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>14 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>15 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>16 babaaabcbaba 0</td>
<td>0</td>
</tr>
<tr>
<td>17 babaaabcbaba 0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Achieve constant-time string compare by indexing into inverse

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


suffixes[9] ≤ suffixes[0]
### Suffix sort: experimental results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>mobydick.txt</th>
<th>aesop.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.