5. Strings

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>
</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>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

Hexadecimal to ASCII conversion table

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

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 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.

String.

<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>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>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
</tbody>
</table>

\( s = \text{ATTACK AT DAWN} \)
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.

```java
public final class StringBuilder
{
    private char[] val; // characters
    private int length; // length of string
    private int capacity; // current capacity

    public StringBuilder(String s)
    {
        this.length = s.length();
        this.capacity = 2; // minimum capacity
        this.val = new char[this.capacity];
        this.val = s; // copy characters
    }
```

The 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;
}
```

quadratic time

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();
}
```

linear time

Remark. StringBuffer data type is similar, but thread safe (and slower).
**String challenge: array of suffixes**

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

```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;
}
```

**Strings 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);
    for (int i = 0; i < N; i++)
        suffixes[i] = sb.substring(i, N);
    return suffixes;
}
```

**Alphabets**

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

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

<table>
<thead>
<tr>
<th>name</th>
<th>R0</th>
<th>R1</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>abcddefghijklmnopqrstuvwxyz</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.

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.

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 $i = 0$; $i < N$; $i++$) $a[i] = aux[i]$;

Typical candidate for key-indexed counting

input name section sorted result
Andersen 2 Harris 1 a
Brown 3 Martin 1 b
Garza 4 Anderson 2 c
Harris 1 Martinez 2 d
Jackson 3 Miller 2 e
Johnson 4 Robinson 2 f
Jones 3 White 2 g
Marti 1 Brown 3 h
Martinez 2 Davis 3 i
Miller 2 Jackson 3 j
Moore 1 Jones 3 k
Robinson 2 Taylor 3 l
Smith 4 Williams 3 m
Wilson 4 Thompson 4 n

offset by 1 [stay tuned]
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]]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
for (int i = 0; i < N; i++)
    aux[aux[count[a[i]]]++] = a[i];
```

### Proposition
- Key-indexed counting uses $8N + 3R$ array accesses to sort $N$ records whose keys are integers between 0 and $R - 1$.
- Key-indexed counting uses extra space proportional to $N + R$.

**Stable?** Yes!
**In-place?** No.
Least-significant-digit-first string sort

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

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

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.

LSD string sort: correctness proof

LSD string sort: example
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?

String 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.

✿ 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.

Google CEO Eric Schmidt interviews Barack Obama

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)

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

Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

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

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

use key-indexed counting on first character
- sort
- transform
- distribute
- and copy back
- indices at completion of distribute phase
- recursively sort subarrays

sorted key

start of subarray
1 + end of subarray
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)]++;
    for (int r = 0; r < R; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[count[charAt(a[i], d)]--];
    for (int r = 0; r < R; r++)
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
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;
}
```

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>$N \log N$</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>$N \log N$</td>
<td>$N + D R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

```
stack depth $D = \text{length of longest prefix match}$  
† probabilistic  
‡ fixed-length $W$ keys  
average-length $W$ keys
```
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.

```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: trace of recursive calls

Partitioning element

3-way string quicksort: Java implementation

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

<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 $N$</td>
<td>2 $N$</td>
<td>$N + R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD †</td>
<td>2 $N$</td>
<td>$N \log_2 N$</td>
<td>$N + D$</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

library call numbers

<table>
<thead>
<tr>
<th>code</th>
<th>call numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>WUS</td>
<td>10706--------7---10</td>
</tr>
<tr>
<td>WUS</td>
<td>12692--------4---27</td>
</tr>
<tr>
<td>WLSOC</td>
<td>2542----30</td>
</tr>
<tr>
<td>LTK</td>
<td>6015-P-63-1988</td>
</tr>
<tr>
<td>LDS</td>
<td>361-H-4</td>
</tr>
</tbody>
</table>

...
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

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

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring: a musical application


```
Mary Had a Little Lamb
Bach's Goldberg Variations
```

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 a

\( i \) \( j \)
```

Analysis. Running time \( MN^2 \), where \( M \) is length of longest match.
Longest repeated substring: a sorting solution

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.

Which one is more likely to lead to a cure cancer?

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);
    int lrsLength = 0;
    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;
}
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.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>subbydick.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</td>
<td>not fixed</td>
</tr>
<tr>
<td>MSD</td>
<td>195</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>

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. $\rightarrow$ Manber’s algorithm
- Linear.
- Nobody knows.

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

Suffix sorting in linearithmic time
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>0  babaabcbabaaa 0</td>
<td>0  aaaaaa0</td>
</tr>
<tr>
<td>1  abababcbabaaa 0</td>
<td>1  aaaaaa0</td>
</tr>
<tr>
<td>2  baaaabcbabaaa 0</td>
<td>2  aaaaaa0</td>
</tr>
<tr>
<td>3  aaaaaaaaaaa 0</td>
<td>3  aaaaaa0</td>
</tr>
<tr>
<td>4  babaabcbabaaa 0</td>
<td>4  aaaaaa0</td>
</tr>
<tr>
<td>5  babaabcbabaaa 0</td>
<td>5  aaaaaa0</td>
</tr>
<tr>
<td>6  babaabcbabaaa 0</td>
<td>6  aaaaaa0</td>
</tr>
<tr>
<td>7  babaabcbabaaa 0</td>
<td>7  aaaaaa0</td>
</tr>
<tr>
<td>8  babaabcbabaaa 0</td>
<td>8  aaaaaa0</td>
</tr>
<tr>
<td>9  abababcbabaaa 0</td>
<td>9  aaaaaa0</td>
</tr>
<tr>
<td>10 abababcbabaaa 0</td>
<td>10 abababcbabaaa 0</td>
</tr>
<tr>
<td>11 abababcbabaaa 0</td>
<td>11 abababcbabaaa 0</td>
</tr>
<tr>
<td>12 abababcbabaaa 0</td>
<td>12 abababcbabaaa 0</td>
</tr>
<tr>
<td>13 abababcbabaaa 0</td>
<td>13 abababcbabaaa 0</td>
</tr>
<tr>
<td>14 abababcbabaaa 0</td>
<td>14 abababcbabaaa 0</td>
</tr>
<tr>
<td>15 abababcbabaaa 0</td>
<td>15 abababcbabaaa 0</td>
</tr>
<tr>
<td>16 abababcbabaaa 0</td>
<td>16 abababcbabaaa 0</td>
</tr>
<tr>
<td>17 abababcbabaaa 0</td>
<td>17 abababcbabaaa 0</td>
</tr>
</tbody>
</table>

sorted

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  babaabcbabaaa 0</td>
<td>0  a000</td>
<td>14</td>
</tr>
<tr>
<td>1  abababcbabaaa 0</td>
<td>1  a000</td>
<td>19</td>
</tr>
<tr>
<td>2  baaaabcbabaaa 0</td>
<td>2  a000</td>
<td>12</td>
</tr>
<tr>
<td>3  aaaaaaaaaaa 0</td>
<td>3  a000</td>
<td>3</td>
</tr>
<tr>
<td>4  babaabcbabaaa 0</td>
<td>4  a000</td>
<td>4</td>
</tr>
<tr>
<td>5  babaabcbabaaa 0</td>
<td>5  a000</td>
<td>7</td>
</tr>
<tr>
<td>6  babaabcbabaaa 0</td>
<td>6  a000</td>
<td>8</td>
</tr>
<tr>
<td>7  babaabcbabaaa 0</td>
<td>7  a000</td>
<td>4</td>
</tr>
<tr>
<td>8  babaabcbabaaa 0</td>
<td>8  a000</td>
<td>11</td>
</tr>
<tr>
<td>9  abababcbabaaa 0</td>
<td>9  a000</td>
<td>16</td>
</tr>
<tr>
<td>10 abababcbabaaa 0</td>
<td>10 abababcbabaaa 0</td>
<td>10</td>
</tr>
<tr>
<td>11 abababcbabaaa 0</td>
<td>11 abababcbabaaa 0</td>
<td>15</td>
</tr>
<tr>
<td>12 abababcbabaaa 0</td>
<td>12 abababcbabaaa 0</td>
<td>2</td>
</tr>
<tr>
<td>13 abababcbabaaa 0</td>
<td>13 abababcbabaaa 0</td>
<td>13</td>
</tr>
<tr>
<td>14 abababcbabaaa 0</td>
<td>14 abababcbabaaa 0</td>
<td>5</td>
</tr>
<tr>
<td>15 abababcbabaaa 0</td>
<td>15 abababcbabaaa 0</td>
<td>1</td>
</tr>
<tr>
<td>16 abababcbabaaa 0</td>
<td>16 abababcbabaaa 0</td>
<td>17</td>
</tr>
<tr>
<td>17 abababcbabaaa 0</td>
<td>17 abababcbabaaa 0</td>
<td>0</td>
</tr>
</tbody>
</table>

sorted

\(\text{suffixes}[13] \leq \text{suffixes}[4]\) (because \(\text{inverse}[13] < \text{inverse}[4]\))

In \(\text{suffixes}[9] \leq \text{suffixes}[0]\)
### 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.
- Key compares not necessary for string keys.
- Use characters as index in an array.

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

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

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