6. Strings

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

Substring. Extract a contiguous list of characters from a string.

```
strings
0 1 2 3 4 5 6
```

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

Implementing Strings in Java

Memory. 40 + 2N bytes for a virgin string!

```
public final class String implements Comparable<String> {
    private char[] value;  // characters
    private int offset;  // index of first char into 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);
    }
    ...
}
```

Strings

Sequence of characters.

Ex. Natural languages, Java programs, 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
Radix Sorting


Longest Common Prefix

Longest common prefix. Given two strings, find the common prefix that is as long as possible.

Radix Sorting

Radix sorting.
- Specialized sorting solution for strings.
- Same ideas for bits, digits, etc.

Applications.
- Bioinformatics.
- Sorting strings.
- Full text indexing.
- Plagiarism detection.
- Burrows-Wheeler transform. [see data compression]
An Application: Redundancy Detector

Longest repeated substring.
- Given a string of N characters, find the longest repeated substring.
- Ex: `a c a a g t t t a c a a g c`
- Application: bioinformatics.

Dumb brute force.
- Try all indices `i` and `j`, and all match lengths `k`, and check.
- `O(W N^3)` time, where `W` is length of longest match.

Suffix sort solution.

Suffix Sorting

Longest repeated substring.
- Given a string of N characters, find the longest repeated substring.
- Ex: `a c a a g t t t a c a a g c`
- Application: bioinformatics.

Brute force.
- Try all indices `i` and `j` for start of possible match, and check.
- `O(W N^2)` time, where `W` is length of longest match.

Suffix sort solution.
- Form `N` suffixes of original string.
- Sort to bring longest repeated substrings together.
- `O(W N \log N)` time, where `W` is length of longest match.
Suffix Sorting: Java Implementation

```java
public class LRS {
    public static void main(String[] args) {
        String s = StdIn.readString();
        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;
        }
        System.out.println(lrs);
    }
}
```

% java LRS < mobydick.txt
"Such a funny, sporty, gany, jesty, joky, hoky-poky lad, is the Ocean, oh! Th

LSD Radix Sort

String Sorting Performance

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>W N^2</td>
</tr>
<tr>
<td>Quicksort</td>
<td>W N log N</td>
</tr>
</tbody>
</table>

\[ W = \text{max length of string} \]
\[ N = \# \text{number of strings} \]
\[ \approx \text{estimate} \]
\[ \pm \text{probabilistic guarantee} \]

1.2 million for Moby Dick

String Sorting Notation

**Notation.**
- String = sequence of characters.
- W = max # characters per string.
- N = # input strings.
- R = radix.

- 256 for extended ASCII, 65,536 for original Unicode

**Java syntax.**
- Array of strings:
  - `String[] a`
  - `N = a.length`
  - `a[i]`
  - `a[i].charAt(d)`
  - `a[0], ..., a[N-1]`
Key Indexed Counting

Key indexed counting.
- Count frequencies of each letter. [0th character]
- Compute cumulative frequencies.

```java
int N = a.length;
int[] count = new int[256+1];
for (int i = 0; i < N; i++) {
    char c = a[i].charAt(d);
    count[c+1]++;
}  // d = 0

frequencies
```

Key Indexed Counting

Key indexed counting.
- Count frequencies of each letter. [0th character]
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```java
for (int i = 0; i < N; i++)
    a[i] = temp[i]; copy back
```

Key Indexed Counting

Key indexed counting.
- Count frequencies of each letter. [0th character]
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```java
for (int i = 1; i < 256; i++)
    count[i] += count[i-1];
cumulative counts
```

Key Indexed Counting

Key indexed counting.
- Count frequencies of each letter. [0th character]
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```java
for (int i = 0; i < N; i++)
    a[i] = temp[i];
copy back
```
**LSD.** Consider digits $d$ from right to left: stably sort using $d$th character as the key via key-indexed counting.

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

**Running time.** $O(W(N + R))$.

**Advantage.** Fastest sorting method for random fixed length strings.

**Disadvantages.**
- Accesses memory "randomly."
- Inner loop has a lot of instructions.
- Wastes time on low-order characters.
- Doesn’t work for variable-length strings.
- Not much semblance of order until very last pass.

**Goal.** Find fast algorithm for variable length strings.
**MSD Radix Sort**

**MSD Radix Sort Implementation**

```java
public static void msd(String[] a) {
    msd(a, 0, a.length, 0);
}

private static void msd(String[] a, int l, int r, int d) {
    // key-indexed counting sort on digit d of a[l] to a[r]
    int[] count = new int[256];
    ...
    // recursively sort 255 subfiles - assumes '\0' terminated
    for (int i = 0; i < 255; i++)
        msd(a, l + count[i], l + count[i+1], d+1);
}
```

**String Sorting Performance**

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>W N²</td>
</tr>
<tr>
<td>Quicksort</td>
<td>W N log N ¹</td>
</tr>
<tr>
<td>LSD *</td>
<td>W(N + R)</td>
</tr>
<tr>
<td>MSD</td>
<td>W(N + R)</td>
</tr>
</tbody>
</table>

- **R** = radix
- **W** = max length of string
- **N** = number of strings

² 1.2 million for Moby Dick

---

**MSD Radix Sort**

- Most significant digit radix sort.
  - Partition file into 256 pieces according to first character.
  - Recursively sort all strings that start with the same character, etc.

**Q.** How to sort on d<sup>th</sup> character?

**A.** Key-indexed counting.
MSD Radix Sort: Small Files

Disadvantages.
- Too slow for small files.
  - ASCII: 100x slower than insertion sort for \( N = 2 \)
  - Unicode: 30,000x slower for \( N = 2 \)
- Huge number of recursive calls on small files.

Solution. Cutoff to insertion sort for small \( N \).

Consequence. Competitive with quicksort for string keys.

Recursive Structure: MSD

Recursive structure. R-way branching leads to lots of empty calls.

String Sorting Performance

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case</td>
<td>Moby Dick</td>
</tr>
<tr>
<td>Brute</td>
<td>( W N^3 )</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( W N \log N ) $^†$</td>
</tr>
<tr>
<td>LSD *</td>
<td>( W(N + R) )</td>
</tr>
<tr>
<td>MSD</td>
<td>( W(N + R) )</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>( W(N + R) )</td>
</tr>
</tbody>
</table>

\( R = \text{radix} \)
\( W = \text{max length of string} \)
\( N = \text{number of strings} \)

$^6$ estimate
$^†$ fixed length strings only
$^†$ probabilistic guarantee

3-Way Radix Quicksort
private static void quicksortX(String a[], int lo, int hi, int d) {
    if (hi - lo <= 0) return;
    int i = lo - 1, j = hi;
    int p = lo - 1, q = hi;
    char v = a[hi].charAt(d);

    while (i < j) {
        while (a[i + 1].charAt(d) < v) if (i == hi) break; // find i on left and
        while (v < a[j - 1].charAt(d)) if (j == lo) break; // j on right to swap
        if (i > j) break;
        exch(a, i, j);
        if (a[i].charAt(d) == v) exch(a, ++p, i); // swap equal char to left or right
        if (a[j].charAt(d) == v) exch(a, --q, j);
    }

    if (p == q) {
        if (v == '\0') quicksortX(a, lo, hi, d + 1); // special case for
        return;
    }

    if (a[i].charAt(d) < v) i++;
    for (int k = lo; k <= p; k++) exch(a, k, p); // swap equal ones
    for (int k = hi; k >= q; k--) exch(a, k, i + k - p); // back to middle
    quicksortX(a, lo, j, d);
    quicksortX(a, i + 1, hi, d + 1);
    if (v == '\0') quicksortX(a, j + 1, i - 1, d + 1);
    quicksortX(a, i, hi, d);
}

3-Way Radix Quicksort

Idea 1. Use dth character to "sort" into 3 pieces instead of 256, and
sort each piece recursively.
Idea 2. Keep all duplicates together in partitioning step.

3-way partition

<table>
<thead>
<tr>
<th>antinian</th>
<th>openbite</th>
<th>antinian</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeffrey</td>
<td>jonedad</td>
<td>bractel</td>
</tr>
<tr>
<td>commond</td>
<td>antinian</td>
<td>openbite</td>
</tr>
<tr>
<td>comnled</td>
<td>bractel</td>
<td>comnled</td>
</tr>
<tr>
<td>secures</td>
<td>secures</td>
<td>cain</td>
</tr>
<tr>
<td>cumin</td>
<td>dilately</td>
<td>diriness</td>
</tr>
<tr>
<td>chariness</td>
<td>inbloc</td>
<td>centesimal</td>
</tr>
<tr>
<td>bractel</td>
<td>jeffrey</td>
<td>dunkersu</td>
</tr>
<tr>
<td>display</td>
<td>display</td>
<td>dimulflex</td>
</tr>
<tr>
<td>mllweight</td>
<td>mllweight</td>
<td>millwright</td>
</tr>
<tr>
<td>repastive</td>
<td>repastive</td>
<td>repastive</td>
</tr>
<tr>
<td>dourness</td>
<td>dourness</td>
<td>dourness</td>
</tr>
<tr>
<td>centesimal</td>
<td>southeast</td>
<td>southeast</td>
</tr>
<tr>
<td>fondier</td>
<td>fondier</td>
<td>fondier</td>
</tr>
<tr>
<td>interval</td>
<td>interval</td>
<td>interval</td>
</tr>
<tr>
<td>reverzional</td>
<td>reverzional</td>
<td>reverzional</td>
</tr>
<tr>
<td>dilitately</td>
<td>cain</td>
<td>secures</td>
</tr>
<tr>
<td>inbloc</td>
<td>diriness</td>
<td>dilately</td>
</tr>
<tr>
<td>southeast</td>
<td>centesimal</td>
<td>inbloc</td>
</tr>
<tr>
<td>dunkersu</td>
<td>dunkersu</td>
<td>jeffrey</td>
</tr>
<tr>
<td>dimulflex</td>
<td>dimulflex</td>
<td>dimulflex</td>
</tr>
</tbody>
</table>

3-way radix quicksort

Recursive Structure: MSD vs. 3-Way Quicksort

3-way radix quicksort collapses empty links in MSD tree.

Quicksort vs. 3-Way Radix Quicksort

Quicksort.
- 2N \ln N string comparisons on average.
- Long keys are costly to compare if they differ only at the end, and
  this is common case!
- absolutism, absolut, absolutely, absolute.

3-way radix quicksort.
- Avoids re-comparing initial parts of the string.
- Uses just "enough" characters to resolve order.
- 2 N \ln N character comparisons on average for random strings.
- Sub-linear sort for large W since input is of size NW.

Theorem. Quicksort with 3-way partitioning is OPTIMAL.
Pf. Ties cost to entropy. Beyond scope of 226.
Suffix Sorting: Worst Case Input

- Length of longest match small.
  - Hard to beat 3-way radix quicksort.

- Length of longest match very long.
  - 3-way radix quicksort is quadratic.
  - Ex: two copies of Moby Dick.

Can we do better? \( \Theta(N \log N) \) ? \( \Theta(N) \)?

**Observation.** Must find longest repeated substring while suffix sorting to beat \( N^2 \).

Suffix Sorting: Worst Case Input

**Input:** “abcdeghiabcdefghi”

### String Sorting Performance

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Worst Case</th>
<th>Moby Dick</th>
<th>AesopFables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brute</strong></td>
<td>$W N^2$</td>
<td>36,000 s</td>
<td>3,990 s</td>
</tr>
<tr>
<td><strong>Quicksort</strong></td>
<td>$W N \log N$ †</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td><strong>LSD</strong> *</td>
<td>$W(N + R)$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>MSD</strong></td>
<td>$W(N + R)$</td>
<td>395</td>
<td>memory</td>
</tr>
<tr>
<td><strong>MSD with cutoff</strong></td>
<td>$W(N + R)$</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td><strong>3-way radix quicksort</strong></td>
<td>$W N \log N$ †</td>
<td>2.8</td>
<td>400</td>
</tr>
<tr>
<td><strong>Merge</strong> ‡</td>
<td>$N \log N$</td>
<td>17</td>
<td>8.5</td>
</tr>
</tbody>
</table>

- $W = \text{max length of string}$
- $N = \text{number of strings}$

* estimate
† fixed length strings only
‡ probabilistic guarantee
§ suffix sorting only

1.2 million for Moby Dick
191 thousand for Aesop’s Fables