6. Strings

Using Strings in Java

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

Substring. Extract a contiguous list of characters from a 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);
    }
}
```

String Implementation in Java

Memory. 28 + 2N bytes for virgin string!

Could use byte array instead of String to save space.

"The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string G’s, A’s, T’s and C’s. This string is the root data structure of an organism’s biology." - M. V. Olson
String vs. StringBuilder

**String.** [immutable] Fast substring, slow concatenation.
**StringBuilder.** [mutable] Slow substring, fast append.

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

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

Radix Sorting

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

**Applications.**
- Sorting strings.
- Full text indexing.
- Plagiarism detection.
- Burrows-Wheeler transform. [see data compression]
- Computational molecular biology.

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: computational molecular biology.

**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.

```
\begin{tabular}{|c|c|c|}
\hline
\code{a a c a a g t t t a c a a g c} \\
\hline
\end{tabular}
```
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: computational molecular biology.

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.
- Form N suffixes of original string.
- Sort to bring longest repeated substrings together.

String Sorting

Notation.
- String = variable length 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
- Number of strings: N = a.length
- The i-th string: a[i]
- The d-th character of the i-th string: a[i].charAt(d)
- Strings to be sorted: a[0], ..., a[N-1]

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);
        System.out.println(lcp(suffixes));
    }
}
```

Suffix Sorting: Java Implementation

A Sorting Solution

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: computational molecular biology.

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.
- Form N suffixes of original string.
- Sort to bring longest repeated substrings together.
### 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 \uparrow )</td>
</tr>
</tbody>
</table>

\( W = \text{max length of string.} \)
\( N = \text{number of strings.} \)
\( \uparrow \text{probabilistic guarantee} \)

### Key Indexed Counting

#### Key Indexed Counting

- Count frequencies of each letter. [0\(^{th}\) character]
- Compute cumulative frequencies.

#### Key Indexed Counting

- Use cumulative frequencies to rearrange strings.

#### Key Indexed Counting

```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]++;
}
```

#### Key Indexed Counting

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

#### Key Indexed Counting

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

<table>
<thead>
<tr>
<th>a</th>
<th>count</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>ace</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>bad</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>bbe</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>bad</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>cab</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>dca</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>dca</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>ebb</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>fad</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>fee</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>fed</td>
<td>11</td>
</tr>
</tbody>
</table>

LSD Radix Sort

Least significant digit radix sort. Ancient method used for card-sorting.

Lysergic Acid Diethylamide, Circa 1960
Card Sorter, Circa 1960

Least significant digit radix sort.
- Consider digits from right to left:
  - Use key-indexed counting to stable sort by character

```
public static void lsd(String[] a) {
    int W = a[0].length();
    for (int d = W-1; d >= 0; d--) {
        // do key-indexed counting sort on digit d ...
    }
}
```

Assumes fixed length strings (length = W)
LSD Radix Sort: Correctness

**Pf 1. [left-to-right]**
- If two strings differ on first character, key-indexed sort puts them in proper relative order.
- If two strings agree on first character, stability keeps them in proper relative order.

**Pf 2. [right-to-left]**
- If the characters not yet examined differ, it doesn’t matter what we do now.
- If the characters not yet examined agree, later pass won’t affect order.

LSD Radix Sort Correctness

**Running time.** \(\Theta(W(N + R))\).

**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 Implementation

```java
public static void msd(String[] a) {
    int N = a.length;
    msd(a, 0, N-1, 0);
    for (int i = 0; i < 255; i++)
        msd(a, l + count[i], l + count[i+1] - 1, d+1);
}
```

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 \(d^{th}\) character?
A. Use key-indexed counting.
String Sorting Performance

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix (sec)</th>
<th>Worst Case</th>
<th>Moby Dick</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute</td>
<td>W N^2</td>
<td>36,000</td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td>W N log N</td>
<td>9.5</td>
<td>36,000</td>
</tr>
<tr>
<td>LSD *</td>
<td>W(N + R)</td>
<td>395</td>
<td></td>
</tr>
<tr>
<td>MSD</td>
<td>W(N + R)</td>
<td>395</td>
<td></td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>W(N + R)</td>
<td>6.8</td>
<td></td>
</tr>
</tbody>
</table>

R = radix.
W = max length of string.
N = number of strings.
\( \hat{N} \) = 1.2 million for Moby Dick

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 of MSD Radix Sort

Trie structure. Describe recursive calls in MSD radix sort.

Problem: Algorithm touches lots of empty nodes ala R-way tries.
- Tree can be as much as 256 times bigger than it appears!
Correspondence With Sorting Algorithms

Correspondence between trees and sorting algorithms.

- BSTs correspond to quicksort recursive partitioning structure.
- R-way tries corresponds to MSD radix sort.
- What corresponds to ternary search tries?

Recursive Structure of MSD Radix Sort vs. 3-Way Quicksort

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

3-way radix quicksort recursion tree
(1035 null links, not shown)

3-way radix quicksort recursion tree
(155 null links)

Correspondence between trees and sorting algorithms.

Idea 1. Use d-th character to "sort" into 3 pieces instead of 256, and sort each piece recursively.

Idea 2. Keep all duplicates together in partitioning step.

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].charAt(d) < v) if (i == hi) break;
        if (i > j) break;
        if (a[i].charAt(d) == v) if (++i == hi) break;
        if (a[i].charAt(d) == v) if (i == hi) break;
        while (v < a[--j].charAt(d)) if (j == lo) break;
        if (j > i) break;
        if (a[j].charAt(d) == v) if (--j == lo) break;
        if (a[j].charAt(d) == v) if (j == lo) break;
    }
    if (p == q) {
        // special case for all equal characters
        if (v == '\0') quicksortX(a, lo, hi, d+1);
        return;
    }
    if (a[i].charAt(d) < v) i++;
    for (int k = lo; k < p; k++) if (a[i].charAt(d) == v) if (k == lo) break;
    for (int k = hi; k > q; k--) if (a[i].charAt(d) == v) if (k == hi) break;
    quicksortX(a, lo, j, d);
    quicksortX(a, i+1, j-1, d);
    quicksortX(a, i, hi, d);
}

3-way Radix Quick Sort
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?  Θ(N log N) ?  Θ(N) ?

Observation. Must find longest repeated substring while suffix sorting to beat N².

Input: "abcdefghiacdefgh"

Suffix Sorting:  Worst Case Input
Input: "abcdeghiabcdefghi"
abcdefghi abcdefghiabcdefghi bcdefghi ... we do better?  !(N log N) ?  !(N) ?
Observation. Must find longest repeated substring while suffix sorting to beat N².

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.

Quicksort vs. 3-Way Radix Quicksort
Quicksort.!2N ln N string comparisons on average.!Long keys are costly to compare ... is of size NW.
Theorem. Quicksort with 3-way partitioning is OPTIMAL.
Pf. Ties cost to entropy. Beyond scope of 226.

Suffix Sorting in Linearithmic Time: Key Idea

Input: "babaaaabcbabaaaaa"
**Suffix Sorting in Subquadratic Time**

*Manber's MSD algorithm.*
- Phase 0: sort on first character using key-indexed sorting.
- Phase $i$: given list of suffixes sorted on first $2^i$ characters, create list of suffixes sorted on first $2^{i-1}$ characters.
- Finishes after $\lg N$ phases.

*Manber's LSD algorithm.*
- Same idea but go from right to left.
- $O(N \log N)$ guaranteed running time.
- $O(N)$ extra space (but need several auxiliary arrays).

**Best in theory.** $O(N)$ but more complicated to implement.

---

**String Sorting Performance**

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix Sort (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Worst Case</td>
</tr>
<tr>
<td>Brute</td>
<td>$W N^2$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$W N \log N$ $^\dagger$</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>
<tr>
<td>3-way radix quicksort</td>
<td>$W N \log N$ $^\dagger$</td>
</tr>
<tr>
<td>Manber $^\dagger$</td>
<td>$N \log N$</td>
</tr>
</tbody>
</table>

$R = \text{radix}$

$W = \text{max length of string}$

$N = \text{number of strings}$

$^\dagger$ fixed length strings only

$^\star$ probabilistic guarantee

$^\dagger$ suffix sorting only

$^\dagger$ estimate

1.2 million for Moby Dick

191 thousand for Aesop's Fables