Radix Sorting

LSD radix sort
MSD radix sort
3-way radix quicksort
Suffix sorting

Applications.
- Sorting strings.
- Full text indexing.
- Plagiarism detection.
- Burrows-Wheeler transform. stay tuned
- Computational molecular biology.

An Application: Redundancy Detector

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

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.
A Sorting Solution

Suffix sort.
• Form N suffixes of original string.
• Sort to bring longest repeated substrings together.

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

Suffix Sorting: Java Implementation

Java implementation.
• We use Java String library functions to simplify code.
• Could use byte array to store ASCII string, and array of pointers into the byte array to save memory.

```
public class SuffixSorter {
  public static void main(String[] args) {
    In stdin = new In();
    String s = stdin.readAll();
    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);
    findLongestMatch(suffixes);
  }
}
```

Diversion: String Implementation in Java

Java implementation of String.
• Immutability: use as Key in symbol table, fast substring.
• Memory for virgin string: 28 + 2N bytes (!)

```
public final class String implements Comparable {
  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 Sorting Performance

```
<table>
<thead>
<tr>
<th></th>
<th>String Sort</th>
<th>Suffix (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case</td>
<td>W N²</td>
<td>36,000 §</td>
</tr>
<tr>
<td>Brute</td>
<td>W N²</td>
<td>36,000 §</td>
</tr>
<tr>
<td>Quicksort†</td>
<td>W N log N †</td>
<td>9.5</td>
</tr>
</tbody>
</table>
```

N = number of strings.
§ estimate
1.2 million for Moby Dick.
191 thousand for Aesop’s Fables.
† probabilistic guarantee.
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 UNICODE).

Java syntax.
- Array of strings: `String[] a;`
- The \( i \)th string: `a[i]`
- The \( d \)th character of the \( i \)th string: `a[i].charAt(d)`
- Strings to be sorted: `a[lo], ..., a[hi]`

Key Indexed Counting

Key indexed counting.
- Count frequencies of each letter. (0th character)
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.
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 = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back

Key Indexed Counting

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

```
for (int i = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back

Key Indexed Counting

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

```
for (int i = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back

Key Indexed Counting

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

```
for (int i = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back

LSD Radix Sort

Least significant digit radix sort.
- Ancient method used for card-sorting.
- Consider digits from right to left:
  - use key-indexed counting to STABLE sort by character

```
for (int i = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back

Key Indexed Counting

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

```
for (int i = L; i <= R; i++) {
    char c = a[i].charAt(d);
    temp[count[c]++] = a[i];
}
```

count temp

```java
da d b  a 1 0
1 1 2
2  c  c  1 2
3  f  d  1 3
4  e  e  1 4
5  d  d  1 5
6  d  d  1 6
d  e  e  1 7
d  f  f  1 8
d  e  e  1 9
d  f  f  1 10
d  e  e  1 11
```

d = 0;

```
for (int i = L; i <= R; i++) {
    a[i] = temp[i - L];
}
```

copy back
LSD Radix Sort

Least significant digit radix sort.
- Ancient method used for card-sorting.
- Consider digits from right to left:
  - use key-indexed counting to STABLE sort by character

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

Fixed length strings (length = W)

LSD Radix Sort: Correctness

Proof 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.

Proof 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. \( \Theta(W(N + R)) \).
- why doesn’t it violate \( N \log N \) lower bound?

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

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.

How to sort on \( d \)th character?
- Use key-indexed counting.
### MSD Radix Sort Implementation

```java
public static void msd(String[] a, int lo, int hi) {
    msd(a, lo, hi, 0);
}

private static void msd(String[] a, int lo, int hi, int d) {
    if (hi <= lo) return;

    // do key-indexed counting sort on digit d
    int[] count = new int[256+1];
    ...

    // recursively sort 255 subfiles - assumes '\0' terminated
    for (int i = 0; i < 255; i++)
        msd(a, L + count[i], L + count[i+1]- 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 N^2</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>

$^*$ estimate

### MSD Radix Sort Analysis

**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.
- Competitive with quicksort for string keys.

### Recursive Structure of MSD Radix Sort

Trie structure to 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?

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 Partitioning

3-way partitioning.
- Natural way to deal with equal keys.
- Partition elements into 3 parts:
  - elements between i and j equal to partition element v
  - no larger elements to left of i
  - no smaller elements to right of j

Dutch national flag problem.
- Not easy to implement efficiently. (Try it!)
- Not done in practical sorts before mid-1990s.
- Incorporated into Java system sort, C qsort.
3-Way Partitioning

Elegant solution to Dutch national flag problem.
- Partition elements into 4 parts:
  - no larger elements to left of m
  - no smaller elements to right of m
  - equal elements to left of p
  - equal elements to right of q
- Afterwards, swap equal keys into center.

All the right properties.
- Not much code.
- In-place.
- Linear if keys are all equal.
- Small overhead if no equal keys.

Significance of 3-Way Partitioning

Equal keys omnipresent in applications when purpose of sort is to bring records with equal keys together.
- Finding collinear points.
- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical application.
- Huge file.
- Small number of key values.
- Randomized 3-way quicksort is LINEAR time. (Try it!)

Theorem. Quicksort with 3-way partitioning is OPTIMAL.
Proof. Ties cost to entropy. Beyond scope of 226.

Quicksort vs. 3-Way Radix Quicksort

Quicksort.
- $2N \log N$ 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.
- $2N \log N$ character comparisons on average for random strings.
- Sub-linear sort for large $W$ since input is of size $NW$.

```java
private static void quicksortX(String a[], int lo, int hi, int d) {
    if (hi - lo <= 0) return;
    int i = lo-1, j = hi, p = lo-1, q = hi;
    char v = a[hi].charAt(d);
    while (i < j) {
        while (a[++i].charAt(d)< v);
        while (v < a[--j].charAt(d))
            if (j == lo) break;
        if (i > j) break;
        exch(a, i, j);
        if (a[i].charAt(d)= = v) {p ++; exch(a, p, i); } swap equal chars
        if (a[j].charAt(d) == v) {q --; exch(a, j, q); } to left or right
    }
    if (p == q) {
        if (v != '\0') quicksortX(a, lo, hi, d+1);
        return;
    }
    if (a[1].charAt(d) < v) i+;
    for (int k = hi; k >= q; k--) exch(a, k, j); swap equal ones
    for (int k = lo; k <= p; k++, j--) exch(a, k, i); back to middle
    quicksortX(a, lo, j, d);
    if ((i == hi) && (a[i].charAt(d) == v)) i+;
    if (v != '\0') quicksortX(a, j+1, i-1, d+1); sort 3 pieces
    quicksortX(a, i, hi, d);
}
```
String Sorting Performance

<table>
<thead>
<tr>
<th>String Sort</th>
<th>Suffix Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worst Case</strong></td>
<td><strong>Moby Dick</strong></td>
</tr>
</tbody>
</table>

- **Brute**: $W N^2$, 36,000
- **QuickSort**: $W N \log N$, 9.5
- **LSD**: $W(N + R)$, -
- **MSD**: $W(N + R)$, 395
- **MSD with cutoff**: $W(N + R)$, 6.8
- **3-Way Radix Quicksort**: $W N \log N$, 2.8

- $R =$ radix,
- $W =$ max length of string,
- $N =$ number of strings.

\[ \text{estimate} \]
\[ \text{fixed length strings only} \]
\[ \text{probabilistic guarantee} \]

Suffix Sorting: Worst Case Input

- Length of longest match small.
  - 3-way radix quicksort rules!
- Length of longest match very long.
  - 3-way radix quicksort is quadratic.
  - Two copies of Moby Dick.

Can we do better?

- $O(N \log N)$?
- $O(N)$?

Observation. Must find longest repeated substring WHILE suffix sorting to beat $N^2$.

Suffix Sorting in $N \log N$ Time: Key Idea

```
0 babaaaabcbabaaaaa0
1 babaaaabcbabaaaa0b
2 babaaaabcbabaaaaa0ba
3 babaaaabcbabaaaaa0bab
4 babaaaabcbabaaaaa0babaa
5 babaaaabcbabaaaaa0babaaa
6 babaaaabcbabaaaaa0babaaaa
7 babaaaabcbabaaaaa0babaaaaa
8 babaaaabcbabaaaaa0babaaaaab
9 babaaaabcbabaaaaa0babaaaaabba
10 babaaaabcbabaaaaa0babaaaaabbaa
11 babaaaabcbabaaaaa0babaaaaabbaaa
12 babaaaabcbabaaaaa0babaaaaabbaaaa
13 babaaaabcbabaaaaa0babaaaaabbaaaa0
14 babaaaabcbabaaaaa0babaaaaabbaaaa00
15 babaaaabcbabaaaaa0babaaaaabbaaaa000
16 babaaaabcbabaaaaa0babaaaaabbaaaa0000
17 babaaaabcbabaaaaa0babaaaaabbaaaa00000
```

Input: "babaaaabcbabaaaaa"

Suffix Sorting in $N \log N$ Time

- Manber's MSD algorithm.
  - Phase 0: sort on first character using key-indexed sorting.
  - Phase $n$: given list of suffixes sorted on first $n$ characters, create list of suffixes sorted on first $2n$ characters.
  - Finishes after $\log N$ phases.

- Manber's LSD algorithm.
  - Same idea but go from right to left.
  - $O(N \log N)$ guaranteed running time.
  - $O(N)$ extra space.
## 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 $\dagger$</td>
<td>$W N \log N$ $\ddagger$</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 $\ddagger$</td>
<td>$W N \log N$ $\ddagger$</td>
</tr>
<tr>
<td>Manber $\ddagger$</td>
<td>$N \log N$</td>
</tr>
</tbody>
</table>

$R =$ radix.

$W =$ max length of string.

$N =$ number of strings.

$\dagger =$ fixed length strings only

$\ddagger =$ probabilistic guarantee

$\ddagger =$ suffix sorting only

$|$ estimate