Sorting Strings and Suffixes

Juha Kärkkäinen, Peter Sanders

MPI für Informatik
Overview

- String sorting (from a mini course at MPII by Juha)
- Skew: Simple scalable suffix sorting
  (also at ICALP 2003 (July) Eindhoven, Netherlands)
- More
String sorting problem

Sort a set $R = \{s_1, s_2, \ldots, s_n\}$ of $n$ (non-empty) strings into the lexicographic order.

Size of input

- $N =$ total length of strings
- $D =$ total length of distinguishing prefixes

Some Notation:

- $s = s[0] \ldots s[|s| - 1]$
- $\forall c \in \Sigma : s[|s|] > c$ (special sentinel character)
Distinguishing prefix

The *distinguishing prefix* of string $s$ in $R$ is

- shortest prefix of $s$ that is not a prefix of another string (or $s$ if $s$ is a prefix of another string)
- shortest prefix of $s$ that determines the rank of $s$ in $R$
The **distinguishing prefix** of string $s$ in $R$ is

- shortest prefix of $s$ that is not a prefix of another string (or $s$ if $s$ is a prefix of another string)
- shortest prefix of $s$ that determines the rank of $s$ in $R$

A sorting algorithm needs to access

- every character in the distinguishing prefixes
- no character outside the distinguishing prefixes
Alphabet model

Ordered alphabet
  ► only comparisons of characters allowed

Constant alphabet
  ► ordered alphabet of constant size
  ► multiset of characters can be sorted in linear time

Integer alphabet
  ► alphabet is \( \{1, \ldots, \sigma\} \) for integer \( \sigma \geq 2 \)
  ► multiset of \( k \) characters can be sorted in \( O(k + \sigma) \) time
### Lower bounds

<table>
<thead>
<tr>
<th>alphabet</th>
<th>lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>$\Omega(D + n \log n)$</td>
</tr>
<tr>
<td>constant</td>
<td>$\Omega(D)$</td>
</tr>
<tr>
<td>integer</td>
<td>$\Omega(D)$</td>
</tr>
</tbody>
</table>
Standard sorting algorithm

- \( \Theta(n \log n) \) string comparisons

Let \( s_i = \alpha \beta_i \), where \( |\alpha| = |\beta_i| = \log n \)

- \( D = \Theta(n \log n) \)
- lower bound:
  \[ \Omega(D + n \log n) = \Omega(n \log n) \]

- standard sorting:
  \[ \Theta(n \log n) \cdot \Theta(\log n) = \Theta(n \log^2 n) \]
Multikey quicksort

- Ternary partition
- On one character at a time

\[
\begin{array}{l}
\text{alphabet} \\
\text{alignment} \\
\text{algorithm} \\
\text{alternative}
\end{array}
\quad \rightarrow
\begin{array}{l}
\text{alignment} \\
\text{algorithm} \\
\text{alternative}
\end{array}
\]
Multikey quicksort

Multikey- quicksort\( (R, \ell) \) \hspace{1em} // \hspace{1em} R = \text{set of strings with common prefix of length } \ell

1 \hspace{1em} \text{if } |R| \leq 1 \text{ then return } R

2 \hspace{1em} \text{choose pivot } p \in R

3 \hspace{1em} R_\prec := \{ s \in R \mid s[\ell + 1] < p[\ell + 1] \} \\
   R_\equiv := \{ s \in R \mid s[\ell + 1] = p[\ell + 1] \} \\
   R_\succ := \{ s \in R \mid s[\ell + 1] > p[\ell + 1] \}

4 \hspace{1em} \text{Multikey- quicksort}(R_\prec, \ell)

5 \hspace{1em} \text{Multikey- quicksort}(R_\equiv, \ell + 1)

6 \hspace{1em} \text{Multikey- quicksort}(R_\succ, \ell)

7 \hspace{1em} \text{return } R_\prec R_\equiv R_\succ
Multikey quicksort: Analysis

- comparisons in partitioning step dominate runtime

1. if $|R| \leq 1$ then return $R$
2. choose pivot $p \in R$
3. $R_\prec := \{ s \in R \mid s[\ell + 1] < p[\ell + 1] \}$
   $R_\prec := \{ s \in R \mid s[\ell + 1] = p[\ell + 1] \}$
   $R_\succ := \{ s \in R \mid s[\ell + 1] > p[\ell + 1] \}$
4. Multikey-quicksort$(R_\prec, \ell)$
5. Multikey-quicksort$(R_\prec, \ell + 1)$
6. Multikey-quicksort$(R_\succ, \ell)$
7. return $R_\prec R_\prec R_\succ$
Multikey quicksort: Analysis

- If $s[\ell + 1] \neq p[\ell + 1]$, charge the comparison on $s$
  - assume perfect choice of pivot
  - size of the set containing $s$ is halved
  - total charge on $s$ is $\leq \log n$
  - total number of $\neq$-comparisons is $\leq n \log n$

\[ R_{<} := \{ s \in R \mid s[\ell + 1] < p[\ell + 1] \} \]
\[ R_{=} := \{ s \in R \mid s[\ell + 1] = p[\ell + 1] \} \]
\[ R_{>} := \{ s \in R \mid s[\ell + 1] > p[\ell + 1] \} \]
Multikey quicksort: Analysis

- If \( s[\ell + 1] = p[\ell + 1] \), charge the comparison on \( s[\ell + 1] \)
  - \( s[\ell + 1] \) becomes part of common prefix
  - total charge on \( s[\ell + 1] \) is \( \leq 1 \)
  - total number of \( =\)-comparisons is \( \leq D \)

3 \[ R_\prec := \{ s \in R \mid s[\ell + 1] < p[\ell + 1] \} \]
4 \( \text{Multikey-quicksort}(R_\prec, \ell) \)
5 \[ R_\succ := \{ s \in R \mid s[\ell + 1] > p[\ell + 1] \} \]
6 \( \text{Multikey-quicksort}(R_\succ, \ell + 1) \)
Multikey quicksort: Analysis

- comparisons in partitioning step dominate runtime

- If $s[\ell + 1] \neq p[\ell + 1]$, charge the comparison on $s$
  - assume perfect choice of pivot
  - size of the set containing $s$ is halved
  - total charge on $s$ is $\leq \log n$
  - total number of $\neq$-comparisons is $\leq n \log n$

- If $s[\ell + 1] = p[\ell + 1]$, charge the comparison on $s[\ell + 1]$
  - $s[\ell + 1]$ becomes part of common prefix
  - total charge on $s[\ell + 1]$ is $\leq 1$
  - total number of $=\ $-comparisons is $\leq D$

- $O(D + n \log n)$ time
Multikey quicksort: Analysis for Random Pivot

The analysis from standard sorting can be adapted to show that the expected number of $\neq$ comparisons is

$$2n \ln n$$
## Multikey quicksort

<table>
<thead>
<tr>
<th>alphabet</th>
<th>lower bound</th>
<th>upper bound</th>
<th>algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>$\Omega(D + n \log n)$</td>
<td>$O(D + n \log n)$</td>
<td>multikey quicksort</td>
</tr>
<tr>
<td>constant</td>
<td>$\Omega(D)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>$\Omega(D)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Radix sort

LSD-first radix sort

- starts from the end of the strings (Least Significant Digit first)
- accesses all characters: $\Omega(N)$ time
- poor when $D \ll N$

MSD-first radix sort

- starts from the beginning of the strings (Most Significant Digit first)
- accesses only distinguishing prefixes
(MSD-first) Radix sort

- recursive σ-way partitioning using counting sort

```
| a | p | habet |
| a | i | gnment |
| a | l | ocate |
| a | g | orithm |
| a | t | ernative |
| a | i | as |
| a | t | ernate |
| a | l |
| z | 0 |
```

```
| a | 0 |
| g | 1 |
| i | 2 |
| l | 2 |

String and suffix sorting – p.14
Radix sort: Analysis

- partitioning a group of $k$ string takes $O(k + \sigma)$ time
- total size of the partitioned groups is $D$
- $O(D)$ total time on constant alphabets
Radix sort: Analysis

- partitioning a group of $k$ string takes $O(k + \sigma)$ time
- total size of the partitioned groups is $D$
- $o(D)$ total time on constant alphabets
- do trivial partitioning (all characters are the same) in $o(k)$ time
- total number of non-trivial partitionings is $< n$
- $o(D + n\sigma)$ total time on integer alphabets
Radix sort: Analysis

- partitioning a group of $k$ string takes $O(k + \sigma)$ time
- total size of the partitioned groups is $D$
- $O(D)$ total time on constant alphabets
- do trivial partitioning (all characters are the same) in $O(k)$ time
- total number of non-trivial partitionings is $<n$
- $O(D + n\sigma)$ total time on integer alphabets
- switch to multikey quicksort when $k < \sigma$: $O(D + n\log \sigma)$ total time
### Radix sort

<table>
<thead>
<tr>
<th>alphabet</th>
<th>lower bound</th>
<th>upper bound</th>
<th>algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>$\Omega(D + n \log n)$</td>
<td>$o(D + n \log n)$</td>
<td>multikey quicksort</td>
</tr>
<tr>
<td>constant</td>
<td>$\Omega(D)$</td>
<td>$o(D)$</td>
<td>radix sort</td>
</tr>
<tr>
<td>integer</td>
<td>$\Omega(D)$</td>
<td>$o(D + n \log \sigma)$</td>
<td>radix sort + multikey quicksort</td>
</tr>
</tbody>
</table>
More radix sorts

<table>
<thead>
<tr>
<th>alphabet</th>
<th>lower bound</th>
<th>upper bound</th>
<th>algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>$\Omega(D + n \log n)$</td>
<td>$o(D + n \log n)$</td>
<td>mk quicksort</td>
</tr>
<tr>
<td>constant</td>
<td>$\Omega(D)$</td>
<td>$o(D)$</td>
<td>radix sort</td>
</tr>
<tr>
<td>integer</td>
<td>$\Omega(D)$</td>
<td>$o(D + n \log \sigma)$</td>
<td>radix sort + mk quicksort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o(D + \sigma \log \sigma)$</td>
<td>breadth-first radix sort + mk quicksort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$o(D + \sigma)$</td>
<td>two-pass radix sort</td>
</tr>
</tbody>
</table>
Suffix sorting problem

Sort the set \( \{S_0, S_1, \ldots, S_{n-1}\} \) of the suffixes of a string \( S \) of length \( n \) (alphabet \([1, n] = \{1, \ldots, n\}\)) into the lexicographic order.

- suffix \( S_i = S[i, n] \) for \( i \in [0 : n - 1] \)

\[
S = \text{banana}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>banana</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>anana</td>
</tr>
<tr>
<td>2</td>
<td>nana</td>
</tr>
<tr>
<td>3</td>
<td>ana</td>
</tr>
<tr>
<td>4</td>
<td>na</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c}
5 & a & \Rightarrow \\
3 & ana & \\
1 & anana & \\
0 & banana & \\
4 & na & \\
2 & nana & \\
\end{array}
\]
**Suffix sorting problem**

Sort the set \( \{S_0, S_1, \ldots, S_{n-1}\} \) of the suffixes of a string \( S \) of length \( n \) (alphabet \( [1, n] = \{1, \ldots, n\} \)) into the lexicographic order.

- suffix \( S_i = S[i, n] \) for \( i \in [0 : n - 1] \)

**Applications**

- full text **indexing** (binary search)
- Burrows-Wheeler transform (\texttt{bzip2} compressor)
- replacement for more complex **suffix tree**
**Suffix tree**

- compact trie of the suffixes
- $O(n)$ time [Farach 97] for integer alphabets
- Most potent tool of stringology?
- Space consuming
- Efficient construction is complicated

$S = \text{banana0}$
A First Divide-and-Conquer Approach

1. \( SA^1 = \text{sort} \{S_i : i \text{ is odd}\} \) (recursion)
2. \( SA^0 = \text{sort} \{S_i : i \text{ is even}\} \) (easy using \( SA^1 \))
3. merge \( SA^1 \) and \( SA^2 \) (very difficult)

Problem: it's hard to compare odd and even suffixes. [Farach 97] developed a linear time suffix tree construction algorithm based on that idea. Very complicated.

Was only known linear time algorithm for suffix arrays
**Skewed Divide-and-Conquer**

1. $SA^{12} = \text{sort } \{S_i : i \mod 3 \neq 0\}$ (recursion)
2. $SA^0 = \text{sort } \{S_i : i \mod 3 = 0\}$ (easy using $SA^{12}$)
3. merge $SA^{12}$ and $SA^0$ (**easy!**)

$$S = \text{banana}$$

<table>
<thead>
<tr>
<th>5</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>anana</td>
</tr>
<tr>
<td>4</td>
<td>na</td>
</tr>
<tr>
<td>2</td>
<td>nana</td>
</tr>
</tbody>
</table>

+ 

<table>
<thead>
<tr>
<th>3</th>
<th>ana</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>banana</td>
</tr>
</tbody>
</table>

$\Rightarrow$

<table>
<thead>
<tr>
<th>5</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>ana</td>
</tr>
<tr>
<td>1</td>
<td>anana</td>
</tr>
<tr>
<td>0</td>
<td>banana</td>
</tr>
<tr>
<td>4</td>
<td>na</td>
</tr>
<tr>
<td>2</td>
<td>nana</td>
</tr>
</tbody>
</table>
Recursion Example

012345678

S anananas.

nananas.0
3 2 5

anananas.00
2 4 1

sort

.00 an an an an an n a s .0
1 2 2 3 4 5

1 2 2 3 4 5

lexicographic triple names

S^12
325241

recursive call

531042
suffix array

436251
lex. names (ranks) among 23 suffixes

a 4 n 2 a n 3 a 5 n a 6 s 1.
Recursion

- sort triples $S[i : i + 2]$ for $i \mod 3 \neq 0$ (LSD-first radix sort)
- $S^{12} = [S'[i] : i \mod 3 = 1] \circ [S'[i] : i \mod 3 = 2]$, suffix $S^{12}_i$ of $S^{12}$ represents $S_{3i+1}$
  suffix $S^{12}_{n/3+i}$ of $S^{12}$ represents $S_{3i+2}$
- recurseOn($S^{12}$) (alphabet size $\leq 2n/3$)
- Annotate the 23-suffixes with their position in rec. sol.
Sorting mod 0 Suffixes

<table>
<thead>
<tr>
<th>0</th>
<th>c 3 (h 4 i h 6 u 2 a h 5 u 1 a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>h 6 (u 2 a h 5 u 1 a)</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>h 5 (u 1 a)</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Use radix sort (LSD-order already known)
**Merge** $SA^{12}$ **and** $SA^0$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 &lt; 1$</td>
<td>$c^n &lt; c^n$</td>
<td>$4$: (6)u 2 (ahua)</td>
</tr>
<tr>
<td>$0 &lt; 2$</td>
<td>$cc^n &lt; cc^n$</td>
<td>$7$: (5)u 1 (a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2$: (4)i h 6 (uahua)</td>
</tr>
<tr>
<td>$3$: h 6u 2 (ahua)</td>
<td>$1$: (3)h 4 (ihuahua)</td>
<td></td>
</tr>
<tr>
<td>$6$: h 5u 1 (a)</td>
<td>$5$: (2)a h 5 (ua)</td>
<td></td>
</tr>
<tr>
<td>$0$: c 3h 4 (ihuahua)</td>
<td>$8$: (1)a 0 0 0 (0)</td>
<td></td>
</tr>
</tbody>
</table>

8: a
5: ahua
0: chihuahua
1: hihuahua
6: hua
3: huahua
2: ihuahua
7: ua
4: uahua
**Analysis**

1. Recursion: \( T(2n/3) \) plus
   
   **Extract triples:** \( O(n) \) (forall \( i, i \mod 3 \neq 0 \) do . . . )
   
   **Sort triples:** \( O(n) \)
   
   (e.g., LSD-first radix sort — 3 passes)
   
   **Lexicographic naming:** \( O(n) \) (scan)
   
   **Build recursive instance:** \( O(n) \) (forall names do . . . )

2. \( SA^0 = \text{sort } \{ S_i : i \mod 3 = 0 \} : O(n) \) (1 radix sort pass)

3. merge \( SA^{12} \) and \( SA^0 \): \( O(n) \) (ordinary merging with strange comparison function)

All in all: \( T(n) \leq cn + T(2n/3) \)

\( \Rightarrow T(n) \leq 3cn = O(n) \)
inline bool leq(int a1, int a2, int b1, int b2) {
    return(a1 < b1 || a1 == b1 && a2 <= b2);
}
inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {
    return(a1 < b1 || a1 == b1 && leq(a2, a3, b2, b3));
}
Implementation: Radix Sorting

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{
    // count occurrences
    int* c = new int[K + 1]; // counter array
    for (int i = 0; i <= K; i++) c[i] = 0; // reset counters
    for (int i = 0; i < n; i++) c[r[a[i]]]++; // count occurrences
    for (int i = 0, sum = 0; i <= K; i++) { // exclusive prefix sums
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i]; // sort
    delete [] c;
}
```
void suffixArray(int* s, int* SA, int n, int K) {
    int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
    int* s12 = new int[n02 + 3]; s12[n02]= s12[n02+1]= s12[n02+2]=0;
    int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
    int* s0 = new int[n0];
    int* SA0 = new int[n0];

    // generate positions of mod 1 and mod 2 suffixes
    // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
    for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) s12[j++]= i;

    // lsb radix sort the mod 1 and mod 2 triples
    radixPass(s12 , SA12, s+2, n02, K);
    radixPass(SA12, s12 , s+1, n02, K);
    radixPass(s12 , SA12, s , n02, K);
}
Implementation: Lexicographic Naming

// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0; i < n02; i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
        name++; c0 = s[SA12[i]]; c1 = s[SA12[i]+1]; c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3] = name; } // left half
    else { s12[SA12[i]/3 + n0] = name; } // right half
Implementation: Recursion

// recurse if names are not yet unique
if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
    for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;
for (int i=0, j=0; i < n02; i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
radixPass(s0, SA0, s, n0, K);
Implementation: Merging

```c
for (int p=0, t=n0-n1, k=0; k < n; k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0 suffix
    if (SA12[t] < n0 ?
        leq(s[i], s12[SA12[t] + n0], s[j], s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    { // suffix from SA12 is smaller
        SA[k] = i; t++;
        if (t == n02) { // done --- only SA0 suffixes left
            for (k++; p < n0; p++, k++) SA[k] = SA0[p];
        }
    } else {
        SA[k] = j; p++;
        if (p == n0) { // done --- only SA12 suffixes left
            for (k++; t < n02; t++, k++) SA[k] = GetI();
        }
    }
}
delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
```

String and suffix sorting – p.33
Tuning

- Eliminate $\text{mod, div}$
- MSD-first radix sort
- Use partial sorting of triples embracing recursion level
- Various locality improvements

Bottom line: Beats previous algorithms for difficult inputs. (but still $\approx 2 \times$ slower for easy inputs.)
**External Memory Implementation**

**Recursion:** \( T(2n/3) \)

- **Extract triples:** scan input
- **Sort triples:** sort (once)
- **Lexicographic naming:**
  - scan sorted triples
- **Build recursive instance:** sort

**Annotate input:** sort

**Sort the rest:** sort\((\text{once})\)

**All in all:** \( O(T_{\text{sort}}(n)) \) I/Os
Parallel Implementation

**sorting**: parallel integer sorting

**lexicographic naming**: prefix sums $O(n/P + \log P)$

Integer alphabet: $O(n^\varepsilon)$ time, $O(n)$ work

Comparison based: $O(\log^2 n)$ time, $O(n \log n)$ work

---

![Diagram of parallel implementation](image)
More Linear Time Algorithms

- [Kim Sim Park Park CPM 03]
  A direct implementation of Farach’s idea. Complicated.

- [Ko Aluru CPM 03]
  a different recursion. Still somewhat complicated.

- [Kärkkäinen Burkhardt CPM 03]
  Cycle covers allow generalization to smaller recursive subproblems. Extra space $O(\epsilon n)$. Linear time with additional ideas from here.

- [Hong Sadakane Sung FOCS 04]
  extra space $O(n)$ bits. Farach’s idea again.

None looks easy to parallelize/externalize.
**Suffix Array Construction: Conclusion**

- simple, direct, linear time suffix array construction
- easy to adapt to advanced models of computation
- generalization to cycle covers yields space efficient implementation

Future/Ongoing Work

- Implementation (internal/external/parallel)
- Large scale applications