5.1 String sorts

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays

String processing

String. Sequence of characters.

Important fundamental abstraction.
- Programming systems (e.g., Java programs).
- Communication systems (e.g., email).
- Information processing.
- Genomic sequences.
- ...

The char data type

C char data type. Typically an 8-bit integer.
- Supports 7-bit ASCII.
- Can represent at most 256 characters.

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

```
Hexadecimal to ASCII conversion table

<table>
<thead>
<tr>
<th>Hex</th>
<th>A</th>
<th>á</th>
<th>Æ</th>
<th>À</th>
<th>Ï</th>
<th>Æ</th>
<th>Ì</th>
<th>Ñ</th>
<th>Ï</th>
<th>Æ</th>
<th>Ì</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</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>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>E</td>
<td>F</td>
<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>6</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>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</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>
</tr>
</tbody>
</table>

some Unicode characters

“A 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
I ♥ Unicode

The String data type

String data type in Java. Immutable sequence of characters.

Length. Number of characters.
Indexing. Get the \textsuperscript{i}th character.
Concatenation. Concatenate one string to the end of another.

\begin{itemize}
\item \texttt{s.length()}
\item \texttt{s.charAt(3)}
\end{itemize}

The String data type: representation

Representation (Java 7). Immutable \texttt{char[]} array + cache of hash.

<table>
<thead>
<tr>
<th>operation</th>
<th>Java</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>\texttt{s.length()}</td>
<td>1</td>
</tr>
<tr>
<td>indexing</td>
<td>\texttt{s.charAt(1)}</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>\texttt{s + t}</td>
<td>\texttt{M + N}</td>
</tr>
<tr>
<td></td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

The String data type: immutability

Q. Why are Java strings immutable?
String performance trap

Q. How to build a long string, one character at a time?

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

A. Use StringBuilder data type (mutable char[] resizing array).

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

Comparing two strings

Q. How many character compares to compare two strings, each of length $W$?

Running time. Proportional to length of longest common prefix.
- Proportional to $W$ in the worst case.
- But, often sublinear in $W$.

Alphabets

Digital key. Sequence of digits over fixed alphabet.
Radix. Number of digits $R$ in alphabet.

<table>
<thead>
<tr>
<th>name</th>
<th>$R$</th>
<th>$\log R$</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>abcdedefghijklmnopqrstuvwxyz defghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>extended 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>

5.1 STRING SORTS

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
**Review: 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>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>1</td>
<td>✓</td>
<td>compareToST</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>✓</td>
<td>compareToST</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>✓</td>
<td>compareToST</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>1</td>
<td></td>
<td>compareToST</td>
</tr>
</tbody>
</table>

* probabilistic

**Lower bound.** $N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?
A. Yes, if we don’t depend on key compares.

**Key-indexed counting demo**

**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 items.
- Copy back into original array.

```java
def count(frequencies):
    for i in range(N):
        a[i] = aux[i];
```

```java
def merge(a[], aux[], N, r, l):
    count = new int[r+1];
    for i in range(l, r):
        count[a[i]]++;+
    for i in range(l, r):
        aux[count[a[i]]] = aux[i];
```

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

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

**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. [stay tuned]

**Remark.** Keys may have associated data $\not\Rightarrow$
can’t just count up number of keys of each value.

**Typical candidate for key-indexed counting**

<table>
<thead>
<tr>
<th>input</th>
<th>sorted result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson 2</td>
<td>Harris 1</td>
</tr>
<tr>
<td>Brown 3</td>
<td>Martin 1</td>
</tr>
<tr>
<td>Davis 3</td>
<td>Moore 1</td>
</tr>
<tr>
<td>Garcia 4</td>
<td>Anderson 2</td>
</tr>
<tr>
<td>Harris 1</td>
<td>Martinez 2</td>
</tr>
<tr>
<td>Jackson 3</td>
<td>Miller 2</td>
</tr>
<tr>
<td>Johnson 4</td>
<td>Robinson 2</td>
</tr>
<tr>
<td>Jones 3</td>
<td>White 2</td>
</tr>
<tr>
<td>Martin 1</td>
<td>Brown 3</td>
</tr>
<tr>
<td>Martinez 2</td>
<td>Davis 3</td>
</tr>
<tr>
<td>Miller 2</td>
<td>Jackson 3</td>
</tr>
<tr>
<td>Moore 1</td>
<td>Jones 3</td>
</tr>
<tr>
<td>Robinson 2</td>
<td>Taylor 3</td>
</tr>
<tr>
<td>Smith 4</td>
<td>Williams 3</td>
</tr>
<tr>
<td>Taylor 3</td>
<td>Garcia 4</td>
</tr>
<tr>
<td>Thomas 4</td>
<td>Johnson 4</td>
</tr>
<tr>
<td>Thompson 4</td>
<td>Smith 4</td>
</tr>
<tr>
<td>White 2</td>
<td>Thomas 4</td>
</tr>
<tr>
<td>Williams 3</td>
<td>Thompson 4</td>
</tr>
<tr>
<td>Wilson 4</td>
<td></td>
</tr>
</tbody>
</table>

**Keys are small integers**
Key-indexed counting demo

**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 items.
- 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]+1]++; 
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

Key-indexed counting demo

**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 items.
- 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]+1]++; 
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

Key-indexed counting demo

**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 items.
- 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]+1]++; 
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

Radix sorting: quiz 1

Which of the following are properties of key-indexed counting?

A. Running time proportional to \( N + R \).
B. Extra space proportional to \( N + R \).
C. Stable.
D. All of the above.
E. *I don't know.*
5.1 STRING SORTS

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays

5.1 STRING SORTS

LSD string sort: correctness proof

**Proposition.** LSD sorts fixed-length strings in ascending order.

**Pf.** [by induction on i]

After pass i, strings are sorted by last i characters.
- If two strings differ on sort key, key-indexed sort puts them in proper relative order.
- If two strings agree on sort key, stability keeps them in proper relative order.

**Proposition.** LSD sort is stable.

**Pf.** Key-indexed counting is stable.

LSD string sort: Java implementation

```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--)
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]] = a[i];
        sort(a, W-d);
    }
}
```
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>$\frac{1}{2}N^2$</td>
<td>$\frac{1}{4}N^2$</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39N \lg N$</td>
<td>$1.39N \lg N$</td>
<td>$c \lg N$</td>
<td></td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort †</td>
<td>$2W(N + R)$</td>
<td>$2W(N + R)$</td>
<td>$N + R$</td>
<td>✔</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

† probabilistic
† fixed-length W keys

Q. What if strings are not all of same length?

Radix sorting: quiz 2

Which sorting method to use to sort 1 million 32-bit integers?

A. Insertion sort.
B. Mergesort.
C. Quicksort.
D. LSD radix sort.
E. I don’t know.

Sort array of 128-bit numbers

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.
How to take a census in 1900s?

1880 Census. Took 1500 people 7 years to manually process data.

Herman Hollerith. Developed a tabulating and sorting machine.
- Use punch cards to record data (e.g., sex, age).
- Machine sorts one column at a time (into one of 12 bins).
- Typical question: how many women of age 20 to 30?

1890 Census. Finished in 1 year (and under budget)!

How to get rich sorting in 1900s?

Punch cards. [1900s to 1950s]
- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

Hollerith’s company later merged with 3 others to form Computing Tabulating Recording Corporation (CTRC); company renamed in 1924.

IBM 80 Series Card Sorter (650 cards per minute)

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
- pick up cards (stable)
- move left one column
- continue until sorted

Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

5.1 STRING SortS
- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
Reverse LSD

- Consider characters from left to right.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).

### MSD string sort: example

**Input**

```
she
sells
seashells
by
the
sea
shore
shells
she
surely
seashells
```

**Output**

```
she
sells
seashells
by
the
sea
shore
shells
she
surely
seashells
```

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

```
  0: a d d
  1: a c e
  2: b a d
  3: c a b
  4: d a b
  5: e b e
  6: f c a b
d b a d
f c a b
g d a b
h d a d
i e b b
j f a d
ba f e e
bb f e d
```

**Sort key**

```
0 1 2 3 4 5 6 7 8 9 10 11
```

**Count**

```
0 1 2 3 4 5 6 7 8 9 10 11
```

**Sort subarrays recursively**

```
0 1 2 3 4 5 6 7 8 9 10 11
```

**Sort key**

```
0 1 2 3 4 5 6 7 8 9 10 11
```

**Variable-length strings**

Treat strings as if they had an extra char at end (smaller than any char).

```
0: s e a -1
1: s e a s h e l l s s l s -1
2: s e a s h e l l s
3: s h e -1
4: s h e -1
5: s h e l l s -1
6: s h e l l s -1
7: s u r e l y -1
```

```
private static int charAt(String s, int d) {
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

C strings. Have extra char \"\0\" at end ⇒ no extra work needed.
**MSD string sort: Java implementation**

```java
public static void sort(String[] a)
{
    aux = new String[a.length];
    sort(a, aux, 0, a.length - 1, 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) + 2]++;
    for (int r = 0; r < R; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]+1] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i-lo];
    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 subarrays.
- Insertion sort, but start at $d^th$ character.

```java
private 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);
}
```

- Implement less() so that it compares starting at $d^th$ character.

```java
private static boolean less(String v, String w, int d)
{
    for (int i = d; i < Math.min(v.length(), w.length()); i++)
    {
        if (v.charAt(i) < w.charAt(i)) return true;
        if (v.charAt(i) > w.charAt(i)) return false;
    }
    return v.length() < w.length();
}
```

**MSD string sort: potential for disastrous performance**

**Observation 1.** Much too slow for small subarrays.
- Each function call needs its own count[] array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

**Observation 2.** Huge number of small subarrays because of recursion.

**MSD string sort: performance**

**Number of characters examined.**
- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!
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>( \frac{1}{2} N^2 )</td>
<td>( \frac{1}{4} N^2 )</td>
<td>1</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
<td>✓</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>✓</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>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort</td>
<td>( 2 W (N + R) )</td>
<td>( 2 W (N + R) )</td>
<td>( N + R )</td>
<td>✓</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD sort</td>
<td>( 2 W (N + R) )</td>
<td>( N \log_a N )</td>
<td>( N + D R )</td>
<td>✓</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

D = function-call stack depth (length of longest prefix match)
† fixed-length \( W \) keys
‡ average-length \( W \) keys

Engineering a radix sort (American flag sort)

Optimization 0. Cutoff to insertion sort.

Optimization 1. Replace recursion with explicit stack.
  - Push subarrays to be sorted onto stack.
  - One count[] array now suffices.

Optimization 2. Do \( R \)-way partitioning in place.
  - Eliminates aux[] array.
  - Sacrifices stability.

American national flag problem

Dutch national flag problem

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.
- Extra space for aux[].
- Extra space for count[].
- Inner loop has a lot of instructions.
- Accesses memory “randomly” (cache inefficient).

Disadvantage of quicksort.
- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

Goal. Combine advantages of MSD and quicksort.

5.1 STRING SORTS

- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the 6th character.
- Less overhead than 4-way partitioning in MSD radix sort.
- Does not re-examine characters equal to the partitioning char.
  (but does re-examine characters not equal to the partitioning char)

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 \(\sim 2N \ln N\) string compares on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.
- Uses \(\sim 2N \ln N\) character compares on average for random strings.
- Avoids re-comparing long common prefixes.
3-way string quicksort vs. MSD string sort

**MSD string sort.**
- Is cache-inefficient.
- Too much memory storing `count[]`.
- Too much overhead reinitializing `count[]` and `aux[]`.

**3-way string quicksort.**
- Is in-place.
- Is cache-friendly.
- Has a short inner loop.
- But not stable.

Bottom line. 3-way string quicksort is 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>½ (N^2)</td>
<td>¼ (N^2)</td>
<td>1</td>
<td>✔️</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>(N)</td>
<td>✔️</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>✔️</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>✔️</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD sort \†</td>
<td>2 (W(N + R))</td>
<td>2 (W(N + R))</td>
<td>(N + R)</td>
<td>✔️</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD sort \‡</td>
<td>2 (W(N + R))</td>
<td>(N \log e N)</td>
<td>(N + D R)</td>
<td>✔️</td>
<td>charAt()</td>
</tr>
<tr>
<td>3–way string quicksort</td>
<td>1.39 (W N \lg R) *</td>
<td>1.39 (N \lg N)</td>
<td>(\log N + W) *</td>
<td>✔️</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
\† fixed-length \(W\) keys
\‡ average-length \(W\) keys

Keyword-in-context search

Given a text of \(N\) characters, preprocess it to enable fast substring search (find all occurrences of query string context).

```bash
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
```

Keyword-in-context search

Given a text of $N$ characters, preprocess it to enable fast substring search (find all occurrences of query string context).

```
% java KWIC tale.txt 15
"search"
"o st giless to search for contraband"
"her unavailing search for your father"
"t provinces in search of impoverishe"
"dispersing in search of other carri"n that bed and search the straw hold
```

Applications. Linguistics, databases, web search, word processing, ...

Keyword-in-context search: suffix-sorting solution

- Preprocess: suffix sort the text.
- Query: binary search for query; scan until mismatch.

KWIC search for "search" in Tale of Two Cities

| 632698 | sealed my letter and |
| 713727 | seamstress is lifted |
| 660598 | seamstress of twenty |
| 67610  | seamstress who was wi |
| 44208  | search for contraband |
| 42705  | search for your father |
| 499797 | search of her husband |
| 182945 | search of impoverishe |
| 143299 | search of other carri |
| 411801 | search the straw hold |
| 158410 | seared marking about |
| 691356 | seab and madame defar |
| 536569 | sease a terrible pass |
| 484763 | sease that had brought |

Suffix sort

```
input string
i t was b e s t i t w a s w
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

form suffixes
0 it was best it was w
1 it was best it was w
2 was best it was w
3 as best it was w
4 s best it was w
5 best it was w
6 est it was w
7 st it was w
8 it was w
9 t it was w
10 t was w
11 w was w
12 a s w
13 s w
14 w
```

sort suffixes to bring query strings together

```
3 as best it was w
12 a s w
5 b est it was w
6 est it was w
0 it was best it was w
1 it was w
2 s best it was w
3 st it was w
7 it was w
8 t it was w
10 t was w
14 w
2 w as best it was w
1 w as w
```

array of suffix indices in sorted order

War story

Q. How to efficiently form (and sort) suffixes?

```
String[] suffixes = new String[N];
for (int i = 0; i < N; i++)
suffixes[i] = s.substring(i, N);
Arrays.sort(suffixes);
```

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>Java 7u5</th>
<th>Java 7u6</th>
</tr>
</thead>
<tbody>
<tr>
<td>amendments.txt</td>
<td>18 thousand</td>
<td>0.25 sec</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>192 thousand</td>
<td>1.0 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>mobyduck.txt</td>
<td>1.2 million</td>
<td>7.6 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>61 sec</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

3rd printing (2012)
The String data type: Java 7u5 implementation

```java
public final class String implements Comparable<String> {
    private char[] value; // characters
    private int offset; // index of first char in array
    private int length; // length of string
    private int hash; // cache of hashCode()
}
```

String s = "Hello, World"

![value[]: H E L L O , W O R L D](image1)

offset = 0

length = 12

String t = s.substring(7, 12);

![value[]: W O R L D](image2)

offset = 7

length = 5

The String data type: Java 7u6 implementation

```java
public final class String implements Comparable<String> {
    private char[] value; // characters
    private int hash; // cache of hashCode()
}
```

String s = "Hello, World"

![value[]: H E L L O , W O R L D](image3)

offset = 0

length = 12

String t = s.substring(7, 12);

![value[]: W O R L D](image4)

offset = 7

length = 5

The String data type: performance

<table>
<thead>
<tr>
<th>operation</th>
<th>Java 7u5</th>
<th>Java 7u6</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>indexing</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring extraction</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>concatenation</td>
<td>M + N</td>
<td>M + N</td>
</tr>
<tr>
<td>immutable?</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>memory</td>
<td>64 + 2N</td>
<td>56 + 2N</td>
</tr>
</tbody>
</table>

A Reddit exchange

I'm the author of the substring() change. As has been suggested in the analysis here there were two motivations for the change:
- Reduce the size of String instances. Strings are typically 20–40% of common apps footprint.
- Avoid memory leakage caused by retained substrings holding the entire character array.

Changing this function, in a bugfix release no less, was totally irresponsible. It broke backwards compatibility for numerous applications with errors that didn't even produce a message, just freezing and timeouts... All pain, no gain. Your work was not just vain, it was thoroughly destructive, even beyond its immediate effect.

http://www.reddit.com/r/programming/comments/1qw73v/til_oracle_changed_the_internal_string
Suffix sort

Q. How to efficiently form (and sort) suffixes in Java 7u6?
A. Define Suffix class ala Java 7u5 String.

```java
public class Suffix implements Comparable<Suffix> {
    private final String text;
    private final int offset;
    public Suffix(String text, int offset) {
        this.text = text;
        this.offset = offset;
    }
    public int length() { return text.length() - offset; }
    public char charAt(int i) { return text.charAt(offset + i); }
    public int compareTo(Suffix that) { /* see textbook */ }
}
```

text[] HELLO, WORLD
0 1 2 3 4 5 6 7 8 9 10 11

Lessons learned

Lesson 1. Put performance guarantees in API.
Lesson 2. If API has no performance guarantees, don’t rely upon any!

Corollary. May want to avoid String data type for huge strings.
• Are you sure charAt() and length() take constant time?
• If lots of calls to charAt(), overhead for function calls is large.
• If lots of small strings, memory overhead of String is large.

Ex. Our optimized algorithm for suffix arrays is 5x faster and uses 32x less memory than our original solution in Java 7u5!

Radix sorting: quiz 3

What is worst-case running time of our suffix array algorithm?
A. Quadratic.
B. Linearithmic.
C. Linear.
D. None of the above.
E. I don’t know.

suffixes
0 a a a a a a a a a a
1 a a a a a a a a a a
2 a a a a a a a a a a
3 a a a a a a a a a a
4 a a a a a a a a a a
5 a a a a a a a a a a
6 a a a a a a a a a a
7 a a a a a a a a a a
8 a a a a a a a a a a
9 a a a a a a a a a a
Suffix arrays: theory

Conjecture (Knuth 1970). No linear-time algorithm.

Proposition. Linear-time algorithms (suffix trees).

" has no practical virtue... but a historic monument in the area of string processing. "

On-line construction of suffix trees ¹

Peter Mäkinen

The Root Corporation, Santa Monica, California²

Abstract

In 1970, Knuth, Prüfer, and Szymanski showed how to do basic suffix sorting in linear time. Several techniques, such as those discussed in [17], have proven to be useful for suffix sorting, but the standard time complexity of the algorithms is quadratic. In this paper we introduce a new data structure called a suffix tree. A linear-time algorithm for constructing a compact version of a suffix tree, associated with a given pattern, is described. Suffix trees can be used to solve several pattern matching problems, including some that are linear time.

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.
- Input size is 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.
- $1.39 N \log N$ chars for random data.

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

Suffix arrays: practice

Applications. Bioinformatics, information retrieval, data compression, ...

Many ingenious algorithms.
- Constants and memory footprint very important.
- State-of-the-art still changing.

<table>
<thead>
<tr>
<th>year</th>
<th>algorithm</th>
<th>worst case</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>Manber–Myers</td>
<td>$N \log N$</td>
<td>$8 N$</td>
</tr>
<tr>
<td>1999</td>
<td>Larsson–Sadakane</td>
<td>$N \log N$</td>
<td>$8 N$</td>
</tr>
<tr>
<td>2003</td>
<td>Kärkkäinen–Sanders</td>
<td>$N$</td>
<td>$13 N$</td>
</tr>
<tr>
<td>2003</td>
<td>Ko–Aluru</td>
<td>$N$</td>
<td>$10 N$</td>
</tr>
<tr>
<td>2008</td>
<td>divsufsort2</td>
<td>$N \log N$</td>
<td>$5 N$</td>
</tr>
<tr>
<td>2010</td>
<td>sais</td>
<td>$N$</td>
<td>$6 N$</td>
</tr>
</tbody>
</table>

¹ (To appear in ALGORITHMICA)

² E-mail: peter.makinen@root.com

³ Tel.: +358-0-7084172, fax: +358-0-7084441

Department of Computer Science, University of Helsinki,
P.O. Box 68 (Tuohimaa), 00014 University of Helsinki, Finland

Tel.: +358-0-726021, fax: +358-0-726022

Abstract

The on-line algorithm is presented for constructing the suffix tree for a given string in linear-time in the length of the string. The new algorithm has the desirable properties of preserving the string stored in the tree and being able to produce the tree in the same order as the input string. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It also allows for fast access to the suffix tree for any given string. A linear-time algorithm for constructing a suffix tree from a given string is presented. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement. The algorithm is based on the ideas of the divide-and-conquer algorithm for constructing the suffix tree. It is simple and easy to implement.