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.**

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

"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

**The char data type**

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

```
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

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

String data type in Java. Immutable sequence of characters.

Length. Number of characters.
Indexing. Get the \( i \)-th character.
Concatenation. Concatenate one string to the end of another.

The String data type: immutability

Q. Why immutable?

A. All the usual reasons.
   • Can use as keys in symbol table.
   • Don’t need to defensively copy.
   • Ensures consistent state.
   • Supports concurrency.
   • Improves security.

public class FileInputstream
{
    private String filename;
    public FileInputstream(String filename) {
        if (!allowedToReadFile(filename))
            throw new SecurityException();
        this.filename = filename;
    }
    ...
}

attacker could bypass security if string type were mutable

The String data type: representation

Representation (Java 7). Immutable 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>s.length()</td>
<td>1</td>
</tr>
<tr>
<td>indexing</td>
<td>s.charAt()</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>s + t</td>
<td>( M + N )</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
String performance trap

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

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

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

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

Comparing two strings

Q. How many character compares to compare two strings of length \( W \) ?

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

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>0123456789ABCDDEF</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>abcdEFGHJIKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>uppercase</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHJIKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>protein</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHJIKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>base64</td>
<td>64</td>
<td>6</td>
<td>ABCDEFGHJIKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>extended ascii</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>unicode 16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

5.1 String sorts

- 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>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^r)</td>
<td>(1.39 N \lg N^r)</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>
</tbody>
</table>

* probabilistic

Lower bound. \(\sim N \lg N\) 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: 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 \(\Rightarrow\) can’t just count up number of keys of each value.

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.

```c
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.

```c
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++)
    a[i] = aux[i];
```

Key-indexed counting: analysis

**Proposition.** Key-indexed takes time proportional to $N + R$.

**Proposition.** Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✓

```c
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++)
    a[i] = aux[i];
```
5.1 STRING SORTS

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

5.1.1 LSD string sort

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.
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>$\lg N$</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>charAt()</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>
</tbody>
</table>

* probabilistic
† fixed-length W keys

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

---

String sorting interview question

Problem. Sort one million 32-bit integers.
Ex. Google (or presidential) interview.

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

Hollerith tabulating machine and sorter

punch card (12 holes per column)

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.

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

Reverse LSD
- Consider characters from left to right.
- Stably sort using \( d \)th character as the key (using key-indexed counting).

5.1 String Sorts
- strings in Java
- key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- suffix arrays
Most-significant-digit-first string sort

- Partition array into $R$ pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).

```
0 d a b
da c a
2 e b d
c a b
d b a
e b f
d d f
f e e
e d a
f e e
d e f
```

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 1 s 1
2 s e l l s 1
3 s h e 1
4 s h e 1
5 s h e l l s 1
6 s h o r e 1
7 s u r e l y 1
```

**C strings.** Have extra char `\0` at end $\rightarrow$ 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+1; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];
    sort R subarrays recursively.
}
```
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.

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();
}
```

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!

compareTo() based sorts can also be sublinear!

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>½ $N^2$</td>
<td>½ $N^2$</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>$N \log_2 N$</td>
<td>$N + D R$</td>
<td>✔</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

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.
  - Now, one count[] array suffices.
- Optimization 2. Do k-way partitioning in place.
  - Eliminates aux[] array.
  - Sacrifices stability.

3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the \(d^\text{th}\) character.
- Less overhead than \(k\)-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.
  (but does re-examine characters not equal to the partitioning char)

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: trace of recursive calls

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 cache-friendly.
- Is in-place.
- Has a short inner loop.

Bottom line. 3-way string quicksort is method of choice for sorting strings.
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>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>charAt()</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>$N \log N$</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).

% 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

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

5.1 STRING Sorts

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

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 for contraband
for your father
le and gone in
searches of her husband
searches of other carriers
that bed and
search the straw hold

better thing

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

input string

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td>s</td>
<td>w</td>
</tr>
</tbody>
</table>

form suffixes

sort suffixes to bring query strings together

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
</tr>
<tr>
<td>t</td>
<td>w</td>
<td>a</td>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
</tr>
<tr>
<td>w</td>
<td>a</td>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>i</td>
<td>t</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>t</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

array of suffix indices in sorted order

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

| 612696 | sealed _my_ letter _and_ _ |
| 713727 | seawater _is_lifted_ _ |
| 660598 | seawater_of_twenty_ _ |
| 67600 | search_for_contraband_ |
| 44430 | search_for_your_Father_ |
| 43795 | search_of_her_husband_ |
| 499708 | search_of_impovertiser_ |
| 182045 | search_of_other_carrier_ |
| 143399 | search_the_straw_hold_ |
| 411801 | seared_marking_about_ _ |
| 158410 | seas_and_madame_defar_ |
| 691536 | sea_a_ _a_ _a_ _a_ _a_ _a_ |
| 536590 | sease_a_ _a_ _a_ _a_ _a_ |
| 484761 | sease_that_had_brought_ |

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

offset = 0

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

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

offset = 7

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);

The String data type: Java 7u5 implementation

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>Java 7u4</th>
<th>Java 7u5</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>mobydict.txt</td>
<td>1.2 million</td>
<td>7.6 sec</td>
<td>out of memory</td>
</tr>
<tr>
<td>chromosomell.txt</td>
<td>7.1 million</td>
<td>61 sec</td>
<td>out of memory</td>
</tr>
</tbody>
</table>
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"

```java
value[] H E L L O , W O R L D
0 1 2 3 4 5 6 7 8 9 10 11
```

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

```java
value[] W O R L D
0 1 2 3 4
```

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

The String data type: performance

String data type (in Java). Sequence of characters (immutable).

Java 7u5. Immutable char[] array, offset, length, hash cache.

Java 7u6. Immutable char[] array, hash cache.

<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>M + N</td>
<td>M + 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>

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 s, int offset) {
        this.text = s;
        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[] H E L L O , W O R L D
0 1 2 3 4 5 6 7 8 9 10 11
```
Suffix sort

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

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

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 7u6!

Suffix Arrays: theory

Q. What is worst-case running time of our suffix arrays algorithm?
- Quadratic.
- Logarithmic.
- Linear.
- None of the above.

N^2 log N

Lessons learned

Q. What is complexity of suffix arrays?
- Quadratic.
- Linear.
- Nobody knows.

Suffix Arrays: theory

Q. What is complexity of suffix arrays?
- Quadratic.
- Linear.

suffix trees (beyond our scope)

Suffix arrays:
A new method for on-line string searches

Li and Myers

The Rand Corporation, Santa Monica, California

Abstract
In 1975, Knuth, Pratt, and Morris [1] showed how to do basic pattern matching in linear time. Before that, linear time had not been achieved by efficient linear algorithms. In this paper, we investigate an interesting data structure called a Suffix Array. A Suffix Array allows us to efficiently store (almost) all suffixes in a string in a manner similar to the Suffix Tree. This structure is used in many algorithms, including some that have been recently improved.

Suffix arrays are a very natural and convenient data structure to store all suffixes of a string. For on-line string searches, the Suffix Array is faster than the Suffix Tree. Our implementation of suffix array is based on the algorithm described in [2]. The algorithm is very simple and has a linear running time. The Suffix Array is an excellent alternative to the Suffix Tree for many applications.
Suffix Arrays: practice

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

Many ingenious algorithms.
- 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>1990</td>
<td>Manber-Myers</td>
<td>$N \log N$</td>
<td>$8N$</td>
</tr>
<tr>
<td>1999</td>
<td>Larsson-Sadakane</td>
<td>$N \log N$</td>
<td>$8N$</td>
</tr>
<tr>
<td>2003</td>
<td>Kärkkäinen-Sanders</td>
<td>$N$</td>
<td>$13N$</td>
</tr>
<tr>
<td>2003</td>
<td>Ko-Aluru</td>
<td>$N$</td>
<td>$10N$</td>
</tr>
<tr>
<td>2008</td>
<td>divsufsort2</td>
<td>$N \log N$</td>
<td>$5N$</td>
</tr>
<tr>
<td>2010</td>
<td>sais</td>
<td>$N$</td>
<td>$6N$</td>
</tr>
</tbody>
</table>

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.
- $1.39 N \lg N$ chars for random data.

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

good choices
(Yuta Mori)