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

5.1 String Sorts

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).
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.

```java
s.length()
```

```
  0  1  2  3  4  5  6  7  8  9 10 11 12
  -------------------------------
  s = A T T A C K A T D A W N

  s.charAt(3)
  s.charAt(11)
```

**The String data type: immutability**

Q. Why are Java strings immutable?

A. All the usual reasons.
   - Provides security.
   - Ensures consistent state.
   - Can use as keys in symbol table.
   - Removes need to defensively copy.
   - Supports concurrency / thread safety.
   - Simplifies tracing and debugging code.
   - Enables compiler to perform certain optimizations.
   - ...

Immutable strings. Java, C#, Python, Scala, ...

Mutable strings. C, C++, Matlab, Ruby, ...

---

**String performance trap**

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

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

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

---

**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(i)</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>s + t</td>
<td>M + N</td>
</tr>
</tbody>
</table>

Q. Could concatenation be O(1)?

A. Yes, but charAt would no longer be.

---

**Comparing two strings**

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

```java
s.compareTo(t)
```

A. Compare with $O(W)$ runtime.

---

**Running time**. Proportional to length of longest common prefix.

- Proportional to $W$ in the worst case.
- But, often sublinear in $W$. 
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()
    ...
}
```

```java
String s = "Hello, World"
```

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

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()
    ...
}
```

```java
String s = "Hello, World"
```

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

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>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
Alphabets

**Digital key.** Sequence of digits over fixed alphabet.

**Radix.** Number of digits it in alphabet.

<table>
<thead>
<tr>
<th>name</th>
<th>R</th>
<th>lg(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>abcdefghijklmnopqrstuvwxyz2</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ2</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>ABCDEFGHIJKLMNOPQRSTUVWXYZ2abcdefgijklmnopqrstuvwxyz2</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>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

Program optimization

"The first rule of program optimization: don’t do it.
The second rule of program optimization (for experts only!): don’t do it yet." — Michael A. Jackson

5.1 String Sorts

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

### 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>$N\log N$</td>
<td>$N\log N$</td>
<td>$N$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$1.39N\log N$</td>
<td>$1.39N\log N$</td>
<td>$e\log N$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$2N\log N$</td>
<td>$2N\log N$</td>
<td>1</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Lower bound. ~ $N\log N$ 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. use array accesses to make 8-way decisions (instead of binary decisions)
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 ⇒ can’t just count up number of keys of each value.

Stably sorting a set of cards by suit

We want clubs, then diamonds, hearts, spades

Stability: within each suit, same order as original

Count the number of cards in each suit

Use this to calculate starting position of each suit in sorted order

Move cards into final position one by one
Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Suit Count Start pos

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>

Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Suit Count Start pos

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>

Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Suit Count Start pos

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>4</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>
### Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>4</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>

### Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>3</td>
</tr>
<tr>
<td>Hearts</td>
<td>4</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>

### Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>3</td>
</tr>
<tr>
<td>Hearts</td>
<td>4</td>
</tr>
<tr>
<td>Spades</td>
<td>6</td>
</tr>
</tbody>
</table>
Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>3</td>
</tr>
<tr>
<td>Hearts</td>
<td>5</td>
</tr>
<tr>
<td>Spades</td>
<td>7</td>
</tr>
</tbody>
</table>

Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>3</td>
</tr>
<tr>
<td>Hearts</td>
<td>5</td>
</tr>
<tr>
<td>Spades</td>
<td>7</td>
</tr>
</tbody>
</table>
Stably sorting a set of cards by suit

<table>
<thead>
<tr>
<th>Suit</th>
<th>Count</th>
<th>Start pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hearts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Spades</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suit</th>
<th>Next pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>1</td>
</tr>
<tr>
<td>Diamonds</td>
<td>3</td>
</tr>
<tr>
<td>Hearts</td>
<td>6</td>
</tr>
<tr>
<td>Spades</td>
<td>9</td>
</tr>
</tbody>
</table>

---

Key-indexed counting

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

Q. Modify this code to sort an array `a[]` of Objects, assuming a `key()` method that returns `int` between `0` and `R-1`.

---

Key-indexed counting

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

Q. Modify this code to sort an array `a[]` of Objects, assuming a `key()` method that returns `int` between `0` and `R-1`.  

---
Radix sorting: quiz 1

Recall from midterm

We can sort an array by date by first sorting it by day, then by month, then by year, but only if we use a stable sorting algorithm.

Least-significant-digit-first string sort

LSD string (radix) sort.
- Consider characters from right to left.
- Stably sort using $d^{th}$ character as the key (using key-indexed counting).
**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}{2} N )</td>
<td>1</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Merge sort</td>
<td>( N \log N )</td>
<td>( N \log N )</td>
<td>( N )</td>
<td>✓</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Quick sort</td>
<td>( 1.39 N \log N )</td>
<td>( 1.39 N \log N )</td>
<td>( c \log N )</td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td>Heap sort</td>
<td>( 2 N \log N )</td>
<td>( 2 N \log 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>
</tbody>
</table>

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

---

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

Divide each word into eight 16-bit “chars”
- $2^8 = 65,536$ counters.
- Sort in 8 passes.

———

**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.

Divide each word into eight 16-bit “chars”
- $2^8 = 65,536$ counters.
- LSD sort on leading 32 bits in 2 passes.
- Finish with insertion sort.
- Examines only $-25\%$ of the data.
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?

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 (CCTR); company renamed in 1924.

IBM 80 Series Card Sorter (650 cards per minute)

LSD string sort: a moment in history (1960s)

card punch
punched cards
card reader
mainframe
line printer
card sorter

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

not directly related to sorting

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 key (using key-indexed counting).

<table>
<thead>
<tr>
<th>$d$</th>
<th>Sort Key (d = 0)</th>
<th>Sort Key (d = 1)</th>
<th>Sort Key (d = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c, b</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>1</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>2</td>
<td>a, d</td>
<td>a, d</td>
<td>a, d</td>
</tr>
<tr>
<td>3</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>4</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>5</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>6</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>7</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>8</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>9</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>10</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
<tr>
<td>11</td>
<td>a, d</td>
<td>b, d</td>
<td>b, c</td>
</tr>
</tbody>
</table>

**Most-significant-digit-first string sort**

**MSD string (radix) 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).

**Variable-length strings**

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

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]++;
    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 < R; i++)
        aux[i] = aux[i + lo];
    for (int r = 0; r < R; r++)
        sort(a, aux, lo + count[r], hi + count[r+1] - 1, d+1);
}
```

**Cutoff to insertion sort**

**Solution.** Cutoff to insertion sort for small subarrays.
- Insertion sort, but start at $d^0$ 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]); j--)
              exchange(a, j, j-1);
  }
  ```
- Implement `less()` so that it compares starting at $d^0$ 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!

---

**Characters examined by MSD string sort**

<table>
<thead>
<tr>
<th>Random (including)</th>
<th>Non-random with duplicate (linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E0462 a.w</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1H1490 b/</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1K2572 sea</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2W1734 seashells</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2T2230 seashells</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2X8846 sells</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3C5733 sells</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CV720 she</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3I2319 she</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3KA382 shells</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TV879 shes</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4C7881 surely</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4Q1284 the</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4Y0229 the</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
</tbody>
</table>

Characters examined by MSD string sort
Summary of the performance of sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stabile?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>( \frac{1}{2} N^2 )</td>
<td>( \frac{1}{2} N^2 )</td>
<td>1</td>
<td>✔</td>
<td>comparesTo()</td>
</tr>
<tr>
<td>MergeSort</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
<td>✔</td>
<td>comparesTo()</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>comparesTo()</td>
</tr>
<tr>
<td>Heapsort</td>
<td>( 2 N \lg N )</td>
<td>( 2 N \lg N )</td>
<td>1</td>
<td>✔</td>
<td>comparesTo()</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 R )</td>
<td>( N + D R )</td>
<td>✔</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

\( D \) = function-call stack depth
\( \) = length of longest prefix match

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 rescans 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.
- One count[] array now suffices.

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

Engineering a radix sort (American flag sort)

American national flag problem
Dutch national flag problem

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 $d^h$ character.
- Less overhead than $R$-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

```java
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 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);
  sort(a, gt+1, hi, d); }
```

3-way string quicksort vs. standard quicksort

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

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

Fast Algorithms for Sorting and Searching Strings

JiE. Bentley*  Robert Sedgewick*

Abstract

We present improved algorithms for sorting and searching, based on multikey fast string searching algorithms. The sorting algorithm blends quicksort and
radix sort, is $2N \ln N$ string compares on average, and runs in $O(N)$ time.

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481, or email permissions@acm.org.

Publication rights reserved by ACM 1-58113-627-1/03/0007. ©2003 ACM 07803-8972-9/03/0007...

```
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

**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_2 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).

```plaintext
% 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
```

---

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 |
| characters of surrounding context |

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

```
630588 sealed many letter and...
713726 seamstress is lifted...
999556 seamstress of twenty...
671060 seamstress who was wi...
645529 search for contraband...
472057 search of her husband...
499787 search of her husband...
180495 search of impoverished...
143199 search of other carriage...
415081 search the straw hold...
158410 seared marking about...
002368 seas and madame defar...
538593 sea a terrible pass...
468761 sea that had brought...
```

Suffix sort

input string

```
it was best it was w
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 it was best it was w
3 it was best it was w
4 it was best it was w
5 it was best it was w
6 it was best it was w
7 it was best it was w
8 it was best it was w
9 it was best it was w
10 it was best it was w
11 it was best it was w
12 it was best it was w
13 it was best it was w
14 it was best it was w
```

sort suffixes to bring query strings together

```
1 asbestos it was w
2 as w
3 best it was w
4 best it was w
5 best it was w
6 best it was w
7 best it was w
8 best it was w
9 best it was w
10 best it was w
11 best it was w
12 best it was w
13 best it was w
```

suffixes to bring query strings together

```
array of suffix indices in sorted order
```

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 */ }
}
```

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

offset
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.

Hint: this is a worst-case input

```
0  a  a  a  a  a  a  a  a  a  a  a
1  a  a  a  a  a  a  a  a  a  a  a
2  a  a  a  a  a  a  a  a  a  a  a
3  a  a  a  a  a  a  a  a  a  a  a
4  a  a  a  a  a  a  a  a  a  a  a
5  a  a  a  a  a  a  a  a  a  a  a
6  a  a  a  a  a  a  a  a  a  a  a
7  a  a  a  a  a  a  a  a  a  a  a
8  a  a  a  a  a  a  a  a  a  a  a
9  a
```

Suffix arrays: theory

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

Proposition. Linear-time algorithms (suffix trees).

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-Alur</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>

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.