String processing

String. Sequence of characters.

Important fundamental abstraction.
• Information processing,
• Genomic sequences.
• 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.
• Need more bits to represent certain characters.

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

The String data type

String data type. Sequence of characters (immutable).

Length. Number of characters.
Indexing. Get the $i$th character.
Substring extraction. Get a contiguous sequence of characters.
String concatenation. Append one character to end of another string.

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

The String data type: Java implementation

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

  public int length()
  {  return length;  }

  public char charAt(int i)
  {  return value[i + offset];  }

  private String(int offset, int length, char[] val)
  {
    this.offset = offset;
    this.length = length;
    this.val    = val;
  }

  public String substring(int from, int to)
  {  return new String(offset + from, to - from, val);  }
}
```

The String data type: performance

```
<table>
<thead>
<tr>
<th>String operation</th>
<th>guaranteed space</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
```

Memory. $40 + 2N$ bytes for a virgin string of length $N$. Can use byte[], char[], or short[] instead of String to save space (but lose convenience of String data type).
The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).
Underlying implementation. Resizing char[] array and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>StringBuilder</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>charAt()</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>substring()</td>
<td>I</td>
<td>N</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Remark. StringBuffer data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q. How to efficiently reverse a string?

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

B.
```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();
}
```

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.
```java
public static String[] suffixes(String s)
{
    int N = s.length();
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);
    return suffixes;
}
```

B.
```java
public static String[] suffixes(String s)
{
    int N = s.length();
    StringBuilder sb = new StringBuilder(s);
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = sb.substring(i, N);
    return suffixes;
}
```
Longest common prefix

Q. How long to compute length of longest common prefix?

Running time. Proportional to length \( D \) of longest common prefix.

Remark. Also can compute `compareTo()` in sublinear time.

```
public static int lcp(String s, String t) {
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return i;
    return N;
}
```

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>( \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>abcdeEfgHi jkTemnopqrstuvwxyz</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>ABCDEFGHIJKLMNOPQRSTUVWXYZ</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>

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 = key compares.

<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>( N^2/2 )</td>
<td>( N^2/4 )</td>
<td>1</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
<td>yes</td>
<td><code>compareTo()</code></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>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>2 ( N \lg N )</td>
<td>2 ( N \lg N )</td>
<td>1</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
</tbody>
</table>

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 ⇒ can't just count up number of keys of each value.

Typical candidate for key-indexed counting

Goal. Sort an array $a[1]$ 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.

Goal. Sort an array $a[1]$ 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.

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

### Key-indexed counting demo

**Goal.** Sort an array $a[i]$ 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]]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    a[i] = count[a[i]]++;
for (int i = 0; i < N; i++)
    aux[i] = a[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];
```
Key-indexed counting demo

Goal. Sort an array $a[i]$ 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++)
    count[a[i]+1]++;
for (int i = 0; i < N; i++)
a[i] = aux[i];
```

Key-indexed counting demo

```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++)
    count[a[i]+1]++;
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++)
    if (count[a[i]+1] > 0)
        aux[count[a[i]+1]] = 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++)
    if (count[a[i]+1] > 0)
        aux[count[a[i]+1]] = 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]]++;;
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]]++;;
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];
```

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];
```

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];
```
### String Sorts

- Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays

### 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.

### 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--)
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
        for (int i = 0; i < N; i++)
            a[i] = aux[i];
    }
}
```

### Least-significant-digit-first string sort

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

**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.

```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--)
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
        for (int i = 0; i < N; i++)
            a[i] = aux[i];
    }
}
```
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>$N^2/2$</td>
<td>$N^2/4$</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N$</td>
<td>yes</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>no</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>$2 W N$</td>
<td>$2 W N$</td>
<td>$N + R$</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

Q. What if strings do not have same length?

String sorting challenge 1

Problem. Sort a huge commercial database on a fixed-length key.
Ex. Account number, date, Social Security number, ...

Which sorting method to use?
- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
✓ LSD string sort.

256 (or 65,536) counters;
Fixed-length strings sort in $W$ passes.

String sorting challenge 2a

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.

String sorting challenge 2b

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.
String sorting challenge 2b

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.

How to take a census in 1900s?

1880 Census. Took 1,500 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?

1890 Census. Finished months early and under budget!

String sorting challenge 2b

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 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); the company was renamed in 1924.
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).

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

**MSD string sort: example**

Key-indexed counting
- LSD radix sort
- MSD radix sort
- 3-way radix quicksort
- Suffix arrays
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 ⇒ 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, 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];
    for (int r = 0; r < R; r++)
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

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

```java
public 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);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```
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!

<table>
<thead>
<tr>
<th>Random (sublinear)</th>
<th>Non-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E10492</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2H4590</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1R2772</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2H734</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2H30230</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2H46666</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3C0773</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CV7720</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3H11319</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3H4182</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TV979</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4G03781</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4QG12384</td>
<td>1DNB377</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4Y02225</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>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>N^2 / 2</td>
<td>N^2 / 4</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N</td>
<td>yes</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>no</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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD ‡</td>
<td>2 N W</td>
<td>2 N W</td>
<td>N + R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>2 N W</td>
<td>N log_e N</td>
<td>N + D R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

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

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.

String Sorts

- 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 $d^{th}$ character.

- Less overhead than $R$-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).

```
3-way string quicksort

Overview. Do 3-way partitioning on the $d^{th}$ character.
- Less overhead than $R$-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).
```

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: trace of recursive calls

```
3-way string quicksort: trace of recursive calls
```

3-way string quicksort vs. 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.**
- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

**Bottom line.** 3-way string quicksort is the 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>(N^2/2)</td>
<td>(N^2/4)</td>
<td>1</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>mergesort</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>(N)</td>
<td>yes</td>
<td><code>compareTo()</code></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>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>heapsort</td>
<td>2 (N \lg N)</td>
<td>2 (N \lg N)</td>
<td>1</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD †</td>
<td>2 (N W)</td>
<td>2 (N W)</td>
<td>(N + R)</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD ‡</td>
<td>2 (N W)</td>
<td>(N \log R)</td>
<td>(N + D R)</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>1.39 (W N \lg N)</td>
<td>1.39 (N \lg N)</td>
<td>(\log N + W)</td>
<td>no</td>
<td><code>charAt()</code></td>
</tr>
</tbody>
</table>

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

---

### String Sorts

- 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
java KWIC tale.txt 15
```

Characters of surrounding context:

- better thing
- t is a far far better thing that i do than
- some sense of better things else forgette
- was capable of better things

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

Input string
```
a a c a g t t t a c a a g c
```

Sort suffixes to bring repeated substrings together
```
0 0
1 11
2 3
3 9
4 1
5 12
6 4
7 10
8 2
9 14
10 1
11 13
12 1
13 1
14 c
c
```

Keyword-in-context search: suffix-sorting solution

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

Longest repeated substring

**Given a string of** $N$ **characters, find the longest repeated substring.**

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

Applications. Bioinformatics, cryptanalysis, data compression, ...
Longest repeated substring

Given a string of \( N \) characters, find the longest repeated substring.

**Brute-force algorithm.**
- Try all indices \( i \) and \( j \) for start of possible match.
- Compute longest common prefix (LCP) for each pair.

![Longest repeated substring: a sorting solution](image)

**Analysis.** Running time \( \leq D N^2 \), where \( D \) is length of longest match.

Longest repeated substring: Java implementation

```java
public String lrs(String s)
{
    int N = s.length();
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
    {
        suffixes[i] = s.substring(i, N);
    }
    Arrays.sort(suffixes);
    String lrs = "";
    for (int i = 0; i < N-1; i++)
    {
        int len = lcp(suffixes[i], suffixes[i+1]);
        if (len > lrs.length())
        {
            lrs = suffixes[i].substring(0, len);
        }
    }
    return lrs;
}
```

% java LRS < moby dick.txt
"Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th

Sorting challenge

**Problem.** Five scientists \( A \), \( B \), \( C \), \( D \), and \( E \) are looking for long repeated substring in a genome with over 1 billion nucleotides.

- \( A \) has a grad student do it by hand.
- \( B \) uses brute force (check all pairs).
- \( C \) uses suffix sorting solution with insertion sort.
- \( D \) uses suffix sorting solution with LSD string sort.
- \( E \) uses suffix sorting solution with 3-way string quicksort.

✓ \( E \) uses suffix sorting solution with 3-way string quicksort.

Q. Which one is more likely to lead to a cure cancer?
Longest repeated substring: empirical analysis

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>brute</th>
<th>suffix sort</th>
<th>length of LRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRS.java</td>
<td>2.162</td>
<td>0.6 sec</td>
<td>0.14 sec</td>
<td>73</td>
</tr>
<tr>
<td>amendments.txt</td>
<td>18.369</td>
<td>37 sec</td>
<td>0.25 sec</td>
<td>216</td>
</tr>
<tr>
<td>asscp.txt</td>
<td>191.945</td>
<td>1.2 hours</td>
<td>1.0 sec</td>
<td>58</td>
</tr>
<tr>
<td>mobydick.txt</td>
<td>1.2 million</td>
<td>43 hours</td>
<td>7.6 sec</td>
<td>79</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>2 months</td>
<td>61 sec</td>
<td>12.567</td>
</tr>
<tr>
<td>pi.txt</td>
<td>10 million</td>
<td>4 months</td>
<td>84 sec</td>
<td>14</td>
</tr>
<tr>
<td>pipi.txt</td>
<td>20 million</td>
<td>forever</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

† estimated

Suffix sorting: worst-case input

Bad input: longest repeated substring very long.
- Ex: same letter repeated $N$ times.
- Ex: two copies of the same Java codebase.

LRS needs at least $1 + 2 + 3 + ... + D$ character compares,
where $D = \text{length of longest match}$

Running time. Quadratic (or worse) in the length of the longest match.

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length $N$.

Q. What is worst-case running time of best algorithm for problem?
- Quadratic.
- ✓ Linearithmic.
- Linear.
- ✓ Nobody knows.

Suffix sorting in linearithmic time

Manber’s MSD algorithm overview.
- Phase 0: sort on first character using key-indexed counting sort.
- Phase $i$: given array of suffixes sorted on first $2^i - 1$ characters,
  create array of suffixes sorted on first $2^i$ characters.

Worst-case running time. $N \lg N$.
- Finishes after $\lg N$ phases.
- Can perform a phase in linear time. (!) [ahead]
Linearithmic suffix sort example: phase 0

Original suffixes:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Key-indexed counting sort (first character):

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Sorted:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Linearithmic suffix sort example: phase 1

Original suffixes:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Index sort (first two characters):

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Sorted:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Linearithmic suffix sort example: phase 2

Original suffixes:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Index sort (first four characters):

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Sorted:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Linearithmic suffix sort example: phase 3

Original suffixes:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Index sort (first eight characters):

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
</tr>
</tbody>
</table>

Finished (no equal keys):

<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>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</td>
<td>ba</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.
- Should measure 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 \text{ chars for random data.}

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

Suffix sort: experimental results

time to suffix sort (seconds)

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydict.txt</th>
<th>aesopaeop.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36.000</td>
<td>4000</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td>LSD</td>
<td>not fixed length</td>
<td>not fixed length</td>
</tr>
<tr>
<td>MSD</td>
<td>395</td>
<td>out of memory</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>2.8</td>
<td>400</td>
</tr>
</tbody>
</table>

Manber MSD: 17 8.5

† estimated