**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.

![Hexadecimal to ASCII conversion table](image)

- **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.

```
s.length()
```

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

```
s.substring(7, 11)
```

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

String data type. Sequence of characters (immutable).

Design Choice. Immutable, cache or share the backing array.
Underlying implementation. Immutable char[] array, offset, and length.

<table>
<thead>
<tr>
<th>String operation</th>
<th>String guarantee</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[] or char[] instead of String to save space (but lose convenience of String data type).
The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).
Design Choice. Easier to update, can’t cache or share array.
Underlying implementation. Resizing char[] array and length.

<table>
<thead>
<tr>
<th>Operation</th>
<th>StringBuilder</th>
<th>Guarantee</th>
<th>Extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>N</td>
<td>N</td>
<td>*</td>
</tr>
<tr>
<td>charAt()</td>
<td>N</td>
<td>N</td>
<td>*</td>
</tr>
<tr>
<td>substring()</td>
<td>N</td>
<td>N</td>
<td>*</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
<td>*</td>
</tr>
</tbody>
</table>

Actually as of Java 1.7 this is O(n) for String as well. Before 1.7 the initial String and substring shared the backing array (no need to copy)

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

String vs. StringBuilder

Q. How to efficiently reverse a string?

A.  
B.  

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

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

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.  
B.  

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;  
}

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?

```
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;
}
```

Running time. Proportional to length $D$ of longest common prefix.

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

Alphabets

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits $R$ in alphabet.

Complexity of some algorithms will depend on this

<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>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDFGHIJKLMNOPQRSTUVWXYZ</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>ABCDFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>Unicode 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
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 phone numbers by area code.
- Sort class roster by section.
- 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.

```
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 (Count Sort)

**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.

```
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.

#### Code

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
```

#### Description

- Copy back into original array.
- Access cumulates using key as index to move items.
- Compute frequency cumulates which specify destinations.
- Count frequencies of each letter using key as index.

#### Table

<table>
<thead>
<tr>
<th>( i )</th>
<th>( a[i] )</th>
<th>( i )</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( i )</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
</tr>
</tbody>
</table>

---

### 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.

#### Code

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
```

#### Description

- Copy back into original array.
- Access cumulates using key as index to move items.
- Compute frequency cumulates which specify destinations.
- Count frequencies of each letter using key as index.

#### Table

<table>
<thead>
<tr>
<th>( i )</th>
<th>( a[i] )</th>
<th>( i )</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( i )</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
</tr>
</tbody>
</table>
**Key-indexed counting demo**

**Goal.** Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

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

**Key-indexed counting demo**

**Goal.** Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int i = 0; i < N; i++)
    aux[count[a[i]]] = a[i];
for (int i = 0; i < N; i++)
a[i] = aux[i];
```
Key-indexed counting demo

Goal. Sort an array $a[]$ of $N$ integers between 0 and $R - 1$.
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
count[r+1] += count[r];

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

**Goal.** Count frequencies of each letter using key as index.

```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 \( a[1] \) 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.

```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 r = 0; r < R; r++)
    if (count[r] > 0)
        System.out.println(a[r] + "\t" + count[r]);
```

Key-indexed counting demo

**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.

```java
int N = a.length;
int[] count = new int[R+1];
for (int i = 0; i < N; i++)
    count[a[i]+1]++;
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
for (int r = 0; r < R; r++)
    if (count[r] > 0)
        System.out.println(a[r] + "\t" + count[r]);
```

Key-indexed counting demo

**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.

```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 r = 0; r < R; r++)
    if (count[r] > 0)
        System.out.println(a[r] + "\t" + count[r]);
```

Key-indexed counting demo

**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.

```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 r = 0; r < R; r++)
    if (count[r] > 0)
        System.out.println(a[r] + "\t" + count[r]);
```
**STRING Sorts**

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

**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.

  - [Thinking about the future]
    - If the characters not yet examined differ, it doesn't matter what we do now.
    - If the characters not yet examined agree, stability ensures later pass won't affect 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--)
            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];
    }
}
```

fixed length W strings

radix R

do key-indexed counting for each digit from right to left

key-indexed counting (count sort)
### 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 \log N )</td>
<td>( N \log N )</td>
<td>( N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( 1.39 N \log N )</td>
<td>( 1.39 N \log N )</td>
<td>( c \log N )</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Heapsort</td>
<td>( 2 N \log N )</td>
<td>( 2 N \log 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>

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

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.

✓ 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. Obama answered “Bubble Sort is not the way to go”

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

Google CEO Eric Schmidt interviews Barack Obama

### String sorting challenge 2a

**Problem.** Sort one million 32-bit integers.

Can view 32-bit integers as:
- Strings of length \( W=1 \) over alphabet of size \( R=2^{32} \)
- Strings of length \( W=2 \) over alphabet of size \( R=2^{16} \)
- Strings of length \( W=3 \) over alphabet of size \( R=2^{8} \)
  ...

- Each LSD sort out of \( W \) takes \( N+R \)
- If \( R=2^{16} \) then we can ignore \( R \), and reduce to \( O(N) \)
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"
$2^{16} = 65,536$ 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 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!
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.

IBM 80 Series Card Sorter (650 cards per minute)

String Sorts

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

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

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).
MSD string sort: example

```java
public static void sort(String[] a) {
    int[] count = new int[R+2];
    for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 1]++;
    aux = new String[a.length];
    sort(a, aux, lo, hi, 0);
}
```

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

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 \( \text{less()} \) 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;  }
```

In Java, forming and comparing substrings is faster than directly comparing chars with \( \text{charAt()} \).

**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

**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 \log N )</td>
<td>( N \log N )</td>
<td>( N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 ( N \log N )</td>
<td>1.39 ( N \log N )</td>
<td>( c \log N )</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>2 ( N \log N )</td>
<td>2 ( N \log N )</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD ( \dagger )</td>
<td>2 ( N \log W )</td>
<td>2 ( N \log W )</td>
<td>( N = R )</td>
<td>yes</td>
<td>\text{charAt()}</td>
</tr>
<tr>
<td>MSD ( \ddagger )</td>
<td>2 ( N \log N )</td>
<td>( N \log W )</td>
<td>( N = D \log K )</td>
<td>yes</td>
<td>\text{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 \( \text{count[]} \).
- Extra space for \( \text{aux[]} \).

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: trace of recursive calls

- Use first character to partition into "less", "equal", and "greater" subarrays.
- Recursively sort subarrays, excluding first character for middle subarray.

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 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 partitioning (using $d^{th}$ character) to handle variable-length strings

3-way string quicksort: Java implementation
3-way string quicksort vs. standard quicksort

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

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

### 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><code>compareTo()</code></td>
</tr>
<tr>
<td>Merge sort</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N)</td>
<td>yes</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>Quicksort</td>
<td>(1.39N \log N)</td>
<td>(1.39N \log N)</td>
<td>(c \log N)</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>Heapsort</td>
<td>(2N \log N)</td>
<td>(2N \log N)</td>
<td>(1)</td>
<td>no</td>
<td><code>compareTo()</code></td>
</tr>
<tr>
<td>LSD</td>
<td>(2N W)</td>
<td>(2N W)</td>
<td>(N + R)</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>MSD</td>
<td>(2N W)</td>
<td>(N \log x N)</td>
<td>(N + D R)</td>
<td>yes</td>
<td><code>charAt()</code></td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>(1.39W N \log N)</td>
<td>(1.39N \log 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

**Summary of the performance of sorting algorithms**

**MSD string sort.**
- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

**3-way string quicksort vs. MSD string sort**

**Bottom line.** 3-way string quicksort is the method of choice for sorting strings.

---

**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 KWIC tale.txt 15
characters of
surrounding context
It was the age of foolishness and the age of wisdom.
It was the best of times and the worst of times.
It was the season of darkness and the season of hope.
```

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

Keyword-in-context search: suffix-sorting solution

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

```
KWIC search for "search" in Tale of Two Cities
```

Applications. Bioinformatics, cryptanalysis, data compression, …

Longest repeated substring

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

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

Applications. Bioinformatics, cryptanalysis, data compression, …
Longest repeated substring: a musical application


Mary Had a Little Lamb

Bach's Goldberg Variations

Longest repeated substring: a musical application

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.

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

Longest repeated substring: a sorting solution

input string

form suffixes

sort suffixes to bring repeated substrings together

compute longest prefix between adjacent suffixes

Longest repeated substring: Java implementation

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 < mobydick.txt
"" Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th

create suffixes (linear time and space)

sort suffixes

find LCP between adjacent suffixes in sorted order
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>aesop.tst</td>
<td>191.945</td>
<td>1.2 hours</td>
<td>1.0 sec</td>
<td>58</td>
</tr>
<tr>
<td>mobydict.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>10 million</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.

```
<table>
<thead>
<tr>
<th>form suffixes</th>
<th>sorted suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>twins twins</td>
<td>twins twins</td>
</tr>
<tr>
<td>twins</td>
<td>twins</td>
</tr>
<tr>
<td>wins</td>
<td>wins</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
</tr>
</tbody>
</table>
```

LRS needs at least \( 1 + 2 + 3 + \ldots + D \) 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. \( \text{✓} \) Manber’s algorithm
- Linearithmic. \( \text{✓} \) Liner.
- Linear. \( \text{✓} \) suffix trees (beyond our scope)
- Nobody knows.
Suffix sorting in linearithmic time

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

Worst-case running time. $N \log N$.
- Finishes after $\log N$ phases.
- Can perform a phase in linear time. (!) [ahead]

Linearithmic suffix sort example: phase 0

original suffixes
1. babaasabcbbabaaaaa0
2. babaabcbbabaaaaa0
3. aabcbabaaaaa0
4. abcbbabaaaaa0
5. abcbabaaaaa0
6. bcbabaaaaa0
7. cbabaaaaa0
8. baaaaa0
9. aaaa0
10. a0
11. 0
12. a0
13. a0
14. a0
15. 0
16. 0
17. 0

key-indexed counting sort (first character)
17. 0
16. a0
15. a0
14. a0
13. a0
12. a0
11. a0
10. a0
9. a0
8. baaaaa0
7. baaaaa0
6. baaaaa0
5. baaaaa0
4. baaaaa0
3. baaaaa0
2. baaaaa0
1. baaaaa0
0. baaaaa0

squared

Linearithmic suffix sort example: phase 1

original suffixes
1. babaasabcbbabaaaaa0
2. babaabcbbabaaaaa0
3. aabcbabaaaaa0
4. abcbbabaaaaa0
5. abcbabaaaaa0
6. bcbabaaaaa0
7. cbabaaaaa0
8. baaaaa0
9. aaaa0
10. a0
11. 0
12. a0
13. a0
14. a0
15. 0
16. 0
17. 0

index sort (first two characters)
17. 0
16. a0
15. a0
14. a0
13. a0
12. a0
11. a0
10. a0
9. a0
8. baaaaa0
7. baaaaa0
6. baaaaa0
5. baaaaa0
4. baaaaa0
3. baaaaa0
2. baaaaa0
1. baaaaa0
0. baaaaa0

squared

Linearithmic suffix sort example: phase 2

original suffixes
1. babaasabcbbabaaaaa0
2. babaabcbbabaaaaa0
3. aabcbabaaaaa0
4. abcbbabaaaaa0
5. abcbabaaaaa0
6. bcbabaaaaa0
7. cbabaaaaa0
8. baaaaa0
9. aaaa0
10. a0
11. 0
12. a0
13. a0
14. a0
15. 0
16. 0
17. 0

index sort (first four characters)
17. 0
16. a0
15. a0
14. a0
13. a0
12. a0
11. a0
10. a0
9. a0
8. baaaaa0
7. baaaaa0
6. baaaaa0
5. baaaaa0
4. baaaaa0
3. baaaaa0
2. baaaaa0
1. baaaaa0
0. baaaaa0

squared
**Linearithmic suffix sort example: phase 3**

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first eight characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a b a a a b c b a b a a a 0</td>
<td>17 0</td>
</tr>
<tr>
<td>a b a a a b c b a b a a a 0</td>
<td>14 0</td>
</tr>
<tr>
<td>a b a a b c b a b a a a 0</td>
<td>13 0</td>
</tr>
<tr>
<td>b c b a b a a a a 0</td>
<td>12 0</td>
</tr>
<tr>
<td>c b a b a a a a 0</td>
<td>11 0</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>10 0</td>
</tr>
<tr>
<td>a a a 0</td>
<td>9 0</td>
</tr>
<tr>
<td>a 0</td>
<td>8 0</td>
</tr>
</tbody>
</table>

Finished (no equal keys)

**Suffix sort: experimental results**

<table>
<thead>
<tr>
<th>time to suffix sort (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>brute-force</td>
</tr>
<tr>
<td>quicksort</td>
</tr>
<tr>
<td>LSD</td>
</tr>
<tr>
<td>MSD</td>
</tr>
<tr>
<td>MSD with cutoff</td>
</tr>
<tr>
<td>3-way string quicksort</td>
</tr>
<tr>
<td>Manber MSD</td>
</tr>
</tbody>
</table>

† estimated

**Constant-time string compare by indexing into inverse**

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first four characters)</th>
<th>inverse frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a b a a a b c b a b a a a 0</td>
<td>17 0</td>
<td>0 14</td>
</tr>
<tr>
<td>a b a a a b c b a b a a a 0</td>
<td>14 0</td>
<td>19</td>
</tr>
<tr>
<td>a b a a b c b a b a a a 0</td>
<td>13 0</td>
<td>2 12</td>
</tr>
<tr>
<td>a b a a b c b a b a a a 0</td>
<td>12 0</td>
<td>3 4</td>
</tr>
<tr>
<td>a b a a b c b a b a a a 0</td>
<td>11 0</td>
<td>6 7</td>
</tr>
<tr>
<td>a b a a b c b a b a a a 0</td>
<td>10 0</td>
<td>5 8</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>9 0</td>
<td>4 11</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>8 0</td>
<td>9 17</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>7 0</td>
<td>8 15</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>6 0</td>
<td>10 10</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>5 0</td>
<td>11 13</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>4 0</td>
<td>12 5</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>3 0</td>
<td>13 6</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>2 0</td>
<td>14 3</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>1 0</td>
<td>15 2</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>0 0</td>
<td>16 1</td>
</tr>
<tr>
<td>a b a a a a 0</td>
<td>0 0</td>
<td>17 0</td>
</tr>
</tbody>
</table>

To do this, inverse-index should be computed for the previous phase. May use for only the last phase

**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.
- $1.39 N \log N$ chars for random data.

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