Intro
Objectives:

- To design and implement a program for a small yet reasonably complicated problem
- To introduce and review a variety of implementation languages and to have students review the pros and cons of different implementation choices and languages.

*Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowcharts; they’ll be obvious.*

— Frederick P. Brooks, *The Mythical Man Month*
Markov
Case Study

Problem  Generate random English text that reads well
Program  Some data comes in, some data goes out, and the processing depends on a little ingenuity
Implementations  C, C++, Java
Failed attempts:

1. Generate random letters (10,000 monkeys typing at 10,000 typewriters)
   - Weighted choices, given letter frequency
2. Choose random words from a dictionary

We need a statistical model with more structure
- Frequency, given some context
Learn from input:

1. Look at all $n$-word phrases (prefixes)
   - Consider the word that follows each prefix
   - The same prefix might appear more than once, maybe with a different suffix

2. Store the (prefix, suffix list) in a dictionary
   - The key is the prefix
   - The satellite data (value) associated w/each prefix is the list of suffixes

Note, we’re creating a multi-map.
- Each prefix can have several possible suffixes
The following example uses a subset of Prof. Brooks’ quote.

- Prefix length of 2 words
- We won’t strip punctuation
- We won’t worry about capitalisation
  - So, "We" and "we" are different strings
- We will use a special (null, null) prefix to indicate the start
Markov Algorithm – Build (Learn)

1. Set $w_1$ and $w_2$ to the sentinel values
2. Read next word into $tok$
3. Add (prefix, suffix) pair to table (dictionary)
4. Replace $(w_1, w_2)$ with $(w_2, tok)$
5. Back to 2
### Example Markov Table

A subset of the states, parsing Brooks’ quote.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Suffix List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null) (null)</td>
<td>Show</td>
</tr>
<tr>
<td>(null) Show</td>
<td>me</td>
</tr>
<tr>
<td>Show me</td>
<td>your your</td>
</tr>
<tr>
<td>me your</td>
<td>flowcharts tables,</td>
</tr>
<tr>
<td>your flowcharts</td>
<td>and</td>
</tr>
<tr>
<td>flowcharts and</td>
<td>conceal</td>
</tr>
<tr>
<td>your tables,</td>
<td>and and</td>
</tr>
<tr>
<td>will be</td>
<td>mystified. obvious.</td>
</tr>
<tr>
<td>be obvious.</td>
<td>(null)</td>
</tr>
<tr>
<td>be mystified.</td>
<td>Show</td>
</tr>
<tr>
<td>mystified. Show</td>
<td>me</td>
</tr>
</tbody>
</table>
Store duplicate suffixes
- E.g., "and" must be a good word to follow "Your tables"
- The statistical bit. "and" is more likely to be chosen

Use (null) to mark the end of a story
Markov Algorithm – Generate

1. Set $w_1$ and $w_2$ to the sentinel values
2. Look up prefix in table, get suffix list
3. Randomly choose suffix $s$
   - If $s$ is sentinel, exit
   - Else, print $s$
4. Replace $(w_1, w_2)$ with $(w_2, s)$
5. Back to 2
Implementation

- See the lecture outline for links to different implementations
  - C (see Makefile)
  - C++
  - Java
  - Python
  - Perl
- What are the pros and cons of the different implementations?
Data Structures
The Data Structures

- Python and Perl have everything we need built in
- Java and C++ provide appropriate containers in their standard libraries
- in C We’ll need to roll these things ourselves
The dictionary \( \texttt{dict} \) is given to us

- The prefixes, the keys in the dictionary, will be stored in 2-element \texttt{tuple}s (immutable)
- The satellite data, list of suffixes, will be stored in a \texttt{list} (a vector)
  - If a prefix doesn’t already exist in the table, we insert it, with an empty \texttt{list}, \([\,]\)
  - Append the new suffix onto the end of this list
We’ll make an open hash table of size $M$ to store our table:

- The prefix (key) is *hashed*
  - Returns a value on $[0, M - 1]$
- Each entry in the table is a bucket of keys
  - Distinct keys that have the same hash value (collision)
  - We’ll use a linked list
- Each prefix is associated with a list of suffixes
The Hash Table – Overview

Figure: Sample State Table
The Hash Table

- The table itself
- An array of pointers to States
- Again, use lists of States (buckets) to handle collisions
The Prefix (State) – the Key

Each state stores
- The prefix of $N_{PREF}$ words
- The list of suffixes
- Pointer to next State in bucket

**Figure: A Single State**

$$N_{PREF} = 2$$

```c
typedef struct State State;
struct State {
    State* next;
    char* pref[NPREF];
    Suffix* suf;
} ;
```
The Satellite Data – Suffix List

- An entry might have several associated data
- Store values in a linked list
- Each Suffix is a node in a linked list
  - word, (a pointer to) the suffix
  - next, pointer to the rest of the list

```c
typedef struct Suffix Suffix;
struct Suffix {
    char* word;
    Suffix* next;
} ;
```

Figure: List of Suffixes
Other C Code – eprintf

- Print an error message to stderr, then exit:
  
  ```c
  void eprintf( char*, ... ) ;
  ```

- Print a warning message to stderr (don’t exit):
  
  ```c
  void weprintf( char*, ... ) ;
  ```

- Call `strdup( s )`, exits if it fails
  
  ```c
  char* estrdup( char *s ) ;
  ```
Other C Code – eprintf (cont.)

- Call \texttt{malloc( n )}, exits if it fails
  
  ```c
  void* emalloc( size_t n );
  ```

- Call \texttt{realloc( p, n )}, exits if it fails
  
  ```c
  void* erealloc( void* p, size_t n );
  ```

- Store program name in a static global
  
  ```c
  void setprogname( char* );
  ```

- Retrieve stored name:
  
  ```c
  char* progrname( void );
  ```
Other C Code – memmove

```c
memmove( t, s, n );
```

- Moves (low-level) a block of memory
- Reads \( n \) bytes, starting at \( s \), and writing at \( t \)
- Okay if regions overlap
- Given prefix \( (w_1, \ldots, w_{n-1}) \), with suffix \( \text{suffix} \):
  ```c
  memmove( prefix, prefix+1, (NPREF-1) * sizeof( prefix[0] ) );
  prefix[NPREF-1] = suffix ;
  ```
- Slides everybody down (left) 1, appends the current suffix
- \( \text{prefix} \) is now \( (w_2, \ldots, w_{n-1}, \text{suffix}) \)
Performance
Performance

On Linux 4.4 (64-bit) with an i7 quad core @ 2GHz:

<table>
<thead>
<tr>
<th>Language</th>
<th>Time (sec)</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.023</td>
<td>194</td>
</tr>
<tr>
<td>C++</td>
<td>0.274</td>
<td>58</td>
</tr>
<tr>
<td>Java</td>
<td>0.135</td>
<td>87</td>
</tr>
<tr>
<td>Python</td>
<td>0.050</td>
<td>55</td>
</tr>
<tr>
<td>Perl</td>
<td>0.056</td>
<td>19</td>
</tr>
</tbody>
</table>

1Using Bash’s `time`
2Using `cloc`