HW 2A in focus

anagram
raga Man

computer science (&) engineering
epic reticence ensuring gnome

notkin beard
drank one bit

David Notkin • Autumn 2009 • CSE303 Lecture 6

HW 2A

• Understanding vs. doing – this is not a straightforward barrier to overcome, and it doesn’t happen all at once
• Breaking down solutions into parts is crucial
• Some amount of “finding things on your own” is essential; perhaps I expected too much of this for this assignment
  – sed and regular expressions
  – loops
  – ...
• Performance – not the high priority
  – Linear vs. quadratic (or worse) complexity of the script

Editors

• To write and change your programs you should be using an editor – what’s the alternative?
• The most common editors on Unix are pico, emacs and vi – pico is simple, emacs is (arbitrarily) complex, and vi is still loved by many old-time Unix users
• You don’t need to become an expert in these, but it’s worth an investment to become capable

ed/sed

• But you didn’t mention sed on the previous slide? Isn’t it a “stream editor”? Indeed it is.
• It’s closely related to ed, a line editor from the first days of Unix
  – ed let you interactively edit lines, changing parts of specific lines – referred to by number and/or by content – inserting and deleting lines, etc.

Example ed session [Wikipedia]

```
$ ed
This is line number two.
1i
This is line number three.
2i
This is line number two.
3i
This is line number three.
$ w text
The end result is a simple text file containing the following text:
ed is the standard Unix text editor.
This is line number three.
```

sed: non-interactive ed

• But sometimes you wanted to use ed-like features – in particular regular expression matching – non-interactively
• That’s what sed is for – using ed-like commands on a string to do transformations that are hard or impossible to do with tr, etc.
• A core feature is the use of regular expressions – these are powerful and found in other Unix tools, most noticeably grep
What is a regular expression?

• "[a-zA-Z_\-]+@[a-zA-Z_\-]+\.[a-zA-Z_]{2,4}"  
  • regular expression: a description of a pattern of text  
    – can test whether a string matches the expression's pattern  
    – can use a regex to search/replace characters in a string  
    – regular expressions are powerful but can be tough to read  
      • the above regular expression matches basic email addresses

Regular expressions

• Appear throughout computer science, in tools, in theory, in practice  
  • Powerful enough to be very useful; other kinds of matching require more powerful languages than regular expressions, but they are more complex  
  • Lots of variations, but all have the same "power" – that is, they can match the same patterns, although the expressions themselves may be more or less complicated

egrep and regexes

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>egrep</td>
<td>extended grep; uses regexes in its search patterns; equivalent to grep -E</td>
</tr>
</tbody>
</table>

egrep "[0-9]{3}-[0-9]{3}-[0-9]{4}"

Basic regexes

• The simplest regexes simply match a particular substring: "abc"  
  • Matches any line containing "abc"  
    – YES: "abc", "abcdef", "defabc", ".-abc.-.", ...,  
    – NO: "fedcba", "ab c", "AbC", "Bash", ...

Wildcards and anchors

• . (a dot) matches any character except \n  – ".\n.y" matches "Doocy", "goofy", "LooPy", ...  
  – use \. to literally match a dot . character  
• ^ matches the beginning of a line; $ the end  
  – "^f\$$" matches lines that consist entirely of "f\n"  
• \< demands that pattern is the beginning of a word; \> demands that pattern is the end of a word  
  – "\<for\>" matches lines that contain the word "for"

Special characters

• | means or  
  – "abc|def|g" matches lines with "abc", "def", or "g"  
• precedence of ^\(Subject\|Date\): vs. \^\(Subject\|Date\):  
  • There's no and symbol. Why not?  
• () are for grouping  
  – "(Homer\|Marge) Simpson" matches lines containing "Homer Simpson" or "Marge Simpson"  
• \ starts an escape sequence: many characters must be escaped to match them: /\$/ \{ \} ^*+?
Quantifiers: * + ?

- * means 0 or more occurrences
  - "abc*" matches "ab", "abc", "abc", "abcc", ...
  - "a(bc)*" matches "a", "abc", "abc", "abcbcbc", ...
  - "a.*a" matches "aa", "aba", "a8ga", "a?_a", ...
- + means 1 or more occurrences
  - "a(bc)+" matches "abc", "abc", "abcbcbc", ...
  - "Goo+gle" matches "Google", "Goooogle", ...
- ? means 0 or 1 occurrences
  - "Martina?" matches lines with "Martin", "Martina"
  - "Dan(iel)??" matches lines with "Dan" or "Daniel"

More quantifiers

- {min,max} means between min and max occurrences
  - "a(bc){2,4}" matches "abc", "abcbcbc", or "abcbcbcbc"
- min or max may be omitted to specify any number
  - "{2,}" means 2 or more
  - ",{6}" means up to 6
  - "{3}" means exactly 3

Character sets

- [ ] group characters into a character set; will match any single character from the set
  - "[bcd]art" matches strings containing "bart", "cart", and "dart"
  - equivalent to "(b|c|d)art"

Character ranges

- Specify a range of characters with
  - "[a-z]" matches any lowercase letter
  - "[a-zA-Z0-9]" matches any lower- or uppercase letter or digit
- an initial ^ inside a character set negates it
  - "[^abcd]" matches any character other than a, b, c, d
- inside a character set, - must be escaped to be matched
  - "[+\-]{0-9}+" matches optional + or -,
    followed by at least one digit

sed

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sed</td>
<td>stream editor; performs regex-based replacements and alterations on input</td>
</tr>
<tr>
<td>s/REGEX/TEXT/g</td>
<td>substitutes (replaces) occurrence(s) of regex with the given text</td>
</tr>
<tr>
<td>s/REGEX/TEXT/g</td>
<td>if filename is omitted, reads from standard input</td>
</tr>
<tr>
<td>s/REGEX/TEXT/g</td>
<td>sed has other uses, but most can be emulated with substitutions</td>
</tr>
<tr>
<td>s/REGEX/TEXT/g</td>
<td>Example (replaces all occurrences of 143 with 303):</td>
</tr>
<tr>
<td>s/143/303/g</td>
<td>sed -r &quot;s/143/303/g lecturenotes.txt</td>
</tr>
<tr>
<td>g</td>
<td>find /usr</td>
</tr>
<tr>
<td>g</td>
<td>&quot;http://&quot;https://&quot;&quot; urls.txt</td>
</tr>
<tr>
<td>g</td>
<td>special characters must be escaped to match them literally</td>
</tr>
<tr>
<td>g</td>
<td>sed can use other delimiters besides / ... whatever follows</td>
</tr>
</tbody>
</table>

More about sed

- sed is line-oriented; processes input a line at a time
- -r option makes regexes work better
  - recognizes ( ), [ ], * , + the "right" way, etc.
- g flag after last / matches all occurrences
- special characters must be escaped to match them literally
  - sed -r "s/http://\"https://\"urls.txt |
- sed can use other delimiters besides / ... whatever follows s |
  - find /usr | sed -r "s#$usr/bin#home/billy#$g" |
Back-references

• every span of text captured by () is given an internal number
  – you can use \number to use the captured text in the replacement
  – \0 is the overall pattern
  – \1 is the first parenthetical capture
  – ...

• Example: swap last names with first names
  – sed -r "s/([^ ]*) , ([^ ]*)/\2 \1/g"

loops

while read line; do
  echo $line
done

Debugging

• “Debugging is important, especially since the shell is so sensitive to details. I recommend two things: (a) trying your commands individually in the command-line as you’re trying to build your shell scripts; and (b) assigning and echoing ‘unnecessary’ variables in your scripts that can be used to help see what’s happening step-by-step.”

• When things don’t work, what do you do?

Performance

• I’m not worried about performance (within a little bit of reason) on 2A. Bill Wulf, who served as president of the National Academy of Engineering for over a decade, once said something like: “More mistakes are made by premature optimization than for any other reason including sheer ignorance.”
  – OK, maybe it doesn’t work right, but at least it’s really fast.
  – Well, if it doesn’t have to work right, I can make it even faster!

Algorithmic complexity

• When dealing with a lot of data, what is usually most important about performance is the underlying algorithmic complexity
  – Very roughly, how many times do you need to touch each data item
• Examples
  – Finding a number in an unsorted list: linear search
  – Finding a number in a sorted list: linear or binary search
  – Sorting a list: O(N²) vs. O(N log N)
• HW2: if you touch every entry in the dictionary many times for each input string, that might be a problem – there are 479,829 entries
Wednesday

• I’d like to spend about 15 minutes having about three students present their solution to 2A to the class
• I’ll pick some varying approaches
• Please send me email if you are willing to present your solution

Questions?