LINUX VOICE TUTORIAL

SOLVE WORD PUZZLES WITH BASH

BEN EVERARD

The humble command line interface is amazingly powerful, for both real work and playing games.

WHY DO THIS?

- Get to grips with egrep and extended regular expressions
- Never get stuck on word puzzles again
- Search through all the text files on your system with ease

It's no secret that Bash, the shell on most Linux systems, is an incredibly powerful tool, however it's one that many Linux users don't take the time to fully learn. A lot of tutorials focus on boring but practical uses like managing log files, but it doesn't have to be this way. Bash can be fun.

Here at Linux Voice, we want to give this tool some love, so we're inaugurating the Grep Games. This is an event where you use Bash together with grep to solve the sort of word puzzles you find in glossy magazines.

Here's an example: what is aedh an anagram of? To solve this, you're going to need a list of English words. This comes as standard on most Linuxes, and can usually be found at **/usr/share/dict/words** or **/usr/dict/words**. If it's not there, check for a **words** or **wordlist** package in your package manager. Failing that, you can grab it from the DVD or **linuxvoice.com**. In this article, we'll use **/usr/share/dict/words**, but you should change this if your **words** file is elsewhere.

We'll use egrep (like grep but uses extended regular expressions, which have a cleaner syntax than plain regular explessions) to find the right words. If you haven't come across this tool before, take a look at the boxout on grep and regular expressions, right.

You can find any word that contains just the letters aedh with this line:

```
egrep "^[aedh]*$" /usr/share/dict/words
```

The **^** matches the start of the line, **$** the end of the line and **[aedh]*** matches any string of the letters aedh. However, these aren't all anagrams. Any anagram must be exactly four letters long, so let's only match words of exactly four characters:

```
egrep "^[aedh]{4}$" /usr/share/dict/words
```

This is a bit better, but there are still some with repeated characters. To solve this we're going to pipe the output into a second instance of egrep, like this:

```
egrep "^[aedh]{4}$" /usr/share/dict/words |  egrep -v "(.).*\1"
```

If you run this, you'll find that it only returns one line, the anagram of aedh. The second **egrep** has the **-v** flag, which means that it works in reverse; that is, it only outputs lines that don't match the pattern. The pattern **(.).*\1** matches any line with a repeated character in it because **(.)** matches any character, **.*** matches any string of any length (including nothing) and **\1** is a back reference to the first character. For more details on this, see backreferences in the boxout on Grep and regular expressions.

Sometimes an anagram will contain a repeated letter, and that would be missed by the above. Take, for example, **eeeddh**. The previous method won't work, so instead we need to match different letters different numbers of times. The code for this is:

```
egrep "^[edh]{6}$" /usr/share/dict/words | egrep  "*^[^e]*(e[^e]*)
{3}[^e]*$" | egrep "^[^d]*(d[^d]*){2}[^d]*$" | egrep -v "([^ed]).*\1]*"
```

Here the second and third egreps both work in the same way. They make sure that a particular letter is repeated exactly a certain number of times. **[^e]** matches any character except **e**, so the second egrep matches any string that starts at a new line, has any character other than a letter 'e' zero or more times followed by three occurrences of the bracketed expression (which contains e once and any string of other characters), then anything that isn't an e zero or more times followed by an end of line.

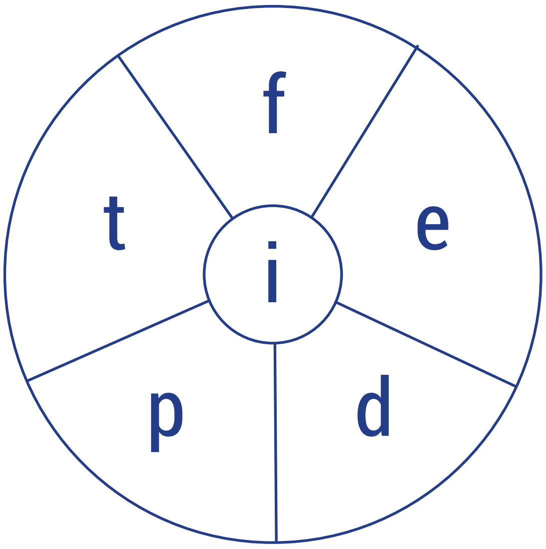
The final egrep makes sure that nothing other than e and d are repeated.

I'll have a vowel please Carol

This solves complete anagrams, but that's not always what you want to do. In the UK there's a quiz show called Countdown, in which the contestants have to make the longest word they can out of a given sequence of nine letters.

You can solve this in a similar manner to the above problem, but by using ranges for the number of characters rather than an absolute number. Take a look at this example for the letters a,e,e,f,d,m,t,t,i

```
egrep "^[aefdmti]{1,9}$" /usr/share/dict/words | egrep
"*^[^e]*(e[^e]*){0,2}[^e]*$" | egrep "^[^t]*(t[^t]*){0,2}[^t]*$" | egrep
-v "([^et]).*\1]*"
```



Word wheels: a challenging mental puzzle or a simple command?

However, this doesn't quite solve our problem. We don't want all the words that match, just the longest one. To get this, we need to go beyond a single line and create a script.

```
#!/bin/bash
longestLength=0
longestWord=""
while read word
do
 if (( ${#word} > longestLength ))
 then
  longestLength=${#word}
  longestWord=$word
 fi
done
echo $longestWord
```

This code reads each line from standard in (while read line) and checks its length against the previous longest word. At the end, it echos (prints) the longest word its found. To include this with the previous egrep commands, just use:

```
egrep "^[aefdmnti]{1,9}$" /usr/share/dict/words | egrep
"*^[^e]*(e[^e]*){0,2}[^e]*$" | egrep "^[^t]*(t[^t]*){0,2}[^t]*$" | egrep
-v "([^et]).*\1]*" | bash longest.sh
```

Where **longest.sh** is the filename of the above script (it's on the website and DVD).

Another puzzle similar to Countdown is the word wheel. This is where there's a series of letters on the outside of a circle and one in the middle. You then have to find as many words as possible that contain the letter in the middle and two or more of the letters on the outside. The example puzzle on the facing page can be solved with:

```
 egrep "^[fedpt]*i[fedpt]*$"  re/dict/words | egrep -v "(.).*\1" |
egrep ".{3,}"
```

Word ladders are a bit different to the puzzles we've looked at so far. Instead of arranging various letters into words, you start with a word, then each rung of the ladder you change a single letter from the word above until you end up with a final word.

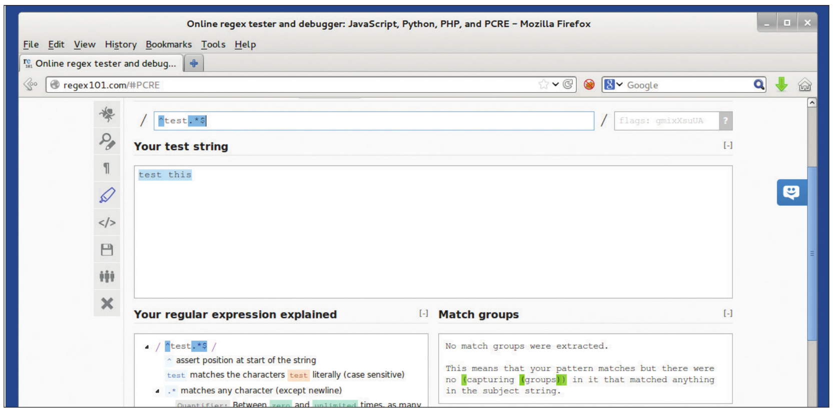
There are two separate parts to look at. The first part is finding all the words that can follow a particular word. The second part is finding out if a particular word can precede the final word.

Let's try the ladder:

```
live
----
----
----
raft
```

To solve this you have to come up with three words.

```
#!/bin/bash
for x in $(egrep "^liv.$|^li.e$|^l.ve$|^.ive$" /usr/share/dict/words)
do
    query='^.'${x:1:3}'$|^'${x:0:1}'.'${x:2:2}'$|^'${x:0:2}'.'${x:3:1}'$|
^'${x:0:3}'.$'
    for y in $(egrep $query /usr/share/dict/words)
    do
        query2='^.'${y:1:3}'$|^'${y:0:1}'.'${y:2:2}'$|^'${y:0:2}'.'${y:
3:1}'$|^'${y:0:3}'.$'
```


Online regex tester and debugger: JavaScript, Python, PHP, and PCRE – Mozilla Firefox

regex101.com/#PCRE

/ ^test.*$ / flags: gmixXsuUA

Your test string

test this

Your regular expression explained

/ ^test.*$ /
^ assert position at start of the string
test matches the characters test literally (case sensitive)
.* matches any character (except newline)
Quantifier: Between zero and unlimited times, as many

Match groups

No match groups were extracted.

This means that your pattern matches but there were no (capturing (groups)) in it that matched anything in the subject string.

www.regex101.com is an online tool to help you understand regular expressions. Unfortunately it uses regular expressions from PHP, Python and JavaScript, which are slightly different from egrep.

Grep and regular expressions

Grep is a popular tool for finding particular pieces of text. As well as solving word games, it's also useful in finding particular messages in log files and other 'real' work. egrep is like grep, but it uses extended regular expressions rather than ordinary regular expressions. These have a cleaner syntax, so it's these that we'll use here.

The basic usage is:

```
egrep <pattern> <file>
```

This will output every line in the file that matches **<pattern>**. It can also be used in a pipe like this:

```
cat <file> | egrep <pattern>
```

This just prints every line that **cat** outputs that matches **<pattern>**.

The trick with egrep is in mastering extended regular expressions.

A letter just matches itself, so for example, **abc** will match any line that contains the string abc anywhere in it. **^** matches the start of the line and **$** matches the end of the line, so **^abc** matches any line that starts with **abc**, **abc$** matches any line that ends with **abc** and **^abc$** matches any line that contains just **abc**. The "**.**" character matches any character, so **^a.c$** will match **abc**, **adc**, **aac**, but not **ac**. This is known as backreferencing.

You can also match groups of characters, eg **^[ab]** will match any line that starts with **a** or **b**, while **^[^ab]** will match any line that starts with any character other than **a** or **b**. **^[a-z]** will match any line starting with a lower-case letter. There are also a few special options here such as **[[:space:]]**, which matches any whitespace (space, tab, etc) and **[[:lower:]]** which matches any lower-case letter.

You can match characters more than once. ***** matches zero or more times, **+** one or more time, and **?** zero or one time. So, **^a*$** matches a line that contains a number of **a**'s but no other characters. **^a.*a$** matches a line that starts and finishes with a letter **a**. **^a.+a$** matches any line that starts and ends with an **a** and has at least one character in between. You can also specify a range of the number of matches you want by using **{}**. For example, **^a{2,3}$** will match the lines **aa** and **aaa**, but nothing else. You can bracket parts of regular expressions as well. This is useful because it allows you to refer to particular matches. **\1** matches whatever the first bracketed expression matched, **\2** matches what the second matched and so on. For example, **(.).\1** will match any two characters that are the same separated by a character, such as **bob**, **did**, **aaa**, but not **abc**.

The final part of extended regular expressions that we'll look at is |. This allows you to match against more than one pattern. For example, **^ab|^bc** will match anything that starts with either **ab** or **bc**, but not **ac** or anything else. **^(ab|bc)** does the same thing.

```
        for z in $(egrep $query2 /usr/share/dict/words | egrep
"^raf.$|^ra.t$|^r.ft$|^.aft$")
        do
                if [ $x != $y ] && [ $x != $z ] && [ $x !=
"live" ] && [ $x != "raft" ] && [ $y != $z ] && [ $y != "live" ] && [ $y
!= "raft" ] && [ $z != "live" ] && [ $z != "raft" ]; then
                echo "live"
                echo $x
                echo $y
                echo $z
                echo "raft"
                echo "---"
                fi
        done
    done
done
```

This code performs three for loops, one for each of the missing words. The first for loop runs on every word that matches the regular expression **"^liv.$|^li.e$|^l.ve$|^.ive$"** this is effectively four different regular expressions separated by **|**. Together, it will return any word that matches any one of these sub-expressions.

Inside this for loop it runs the line

```
query='^.'${x:1:3}'$|^'${x:0:1}'.'${x:2:2}'$|^'${x:0:2}'.'${x:3:1}'$|^'$
{x:0:3}'.$'
```

This just builds up a regular expression equivalent to the first one but for every word returned. **x** is the variable holding the word, and **${x:1:3}** (for example) returns characters 1 through 3 of the word held in variable x (the first character is 0). The second for loop works in exactly the same way as the first.

The final for loop is a bit different because it not only has to match the word above it, but the word below it as well. For this reason it runs two egreps on the words: one to match the words above, and the second to match the words below. The if statement simply removes any solutions that repeat words.

Playing GCHQ

Substitution ciphers are easy-to-break encryption systems where you take each letter of the alphabet and represent it with a different symbol. The point of the puzzle is to work out what letters the symbols represent. As an example, the cipher:

```
12334, 56 7852 90 a27
```

could correspond to:

```
hello, my name is ben
```

because h=1, e=2, l=3, o=4, m=5, y=6, n=7, a=8, i=9, b=a.

Now take a look at the following:

```
123452 672 8298a2 bc 9889dbeb9c
```

The main clue here are repeated letters which you can match using back references. You could try to build a script to match the whole lot in one go, but it's far easier and quicker to pick on part with quite a few repeated characters and just match that. Once you've got that, it should be quite trivial to finish it off. We decided to work with the final two words. A script to solve them is:

```
#!/bin/bash
list2=$(egrep "(.)(.)\2\1.(.).\3\1." /usr/share/dict/words)
for word1 in $(egrep "^.{2}$" /usr/share/dict/words)
do
        for word2 in $list2
        do
                echo $word1" "$word2 | egrep "^(.)(.)
[[:space:]](.)(.)\4\3.\1.\1\3\2$"
        done
done
```

The first loop goes through every two letter word while the second one loops through every word that matches the particular pattern of backreferences.

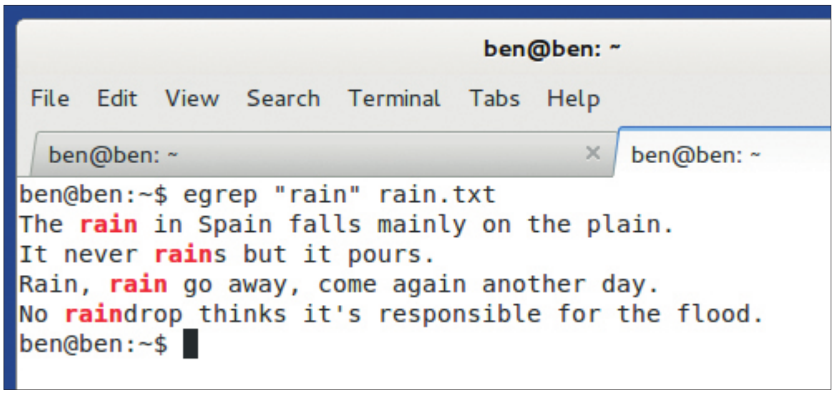
The guts of the code is the line:

```
echo $word1" "$word2 | egrep "^(.)(.)[[:space:]](.)
(.)\4\3.\1.\1\3\2$"
```

It checks every pair of words generated by the two wloops for a particular pattern of back references which correspond to repeated characters in the ciphertext.

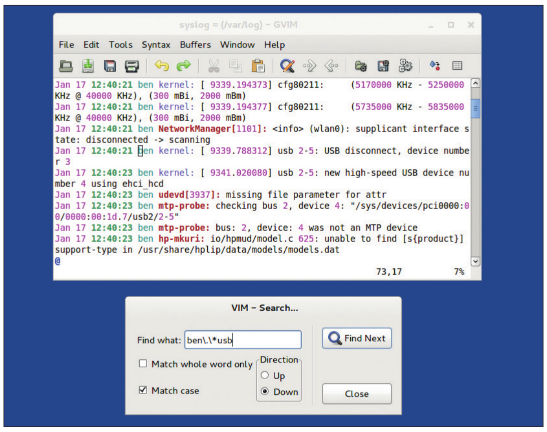
This method could be expanded to match three or more words, though it will slow down significantly with each new word.

Once you've got some of the letters, you should be able to come up with patterns based on the letters you know to find the other words.



```
ben@ben:~$ egrep "rain" rain.txt
The rain in Spain falls mainly on the plain.
It never rains but it pours.
Rain, rain go away, come again another day.
No raindrop thinks it's responsible for the flood.
ben@ben:~$
```

egrep will highlight the particular part of each line that matches the regular expression.


syslog = (/var/log) - GVIM
File Edit Tools Syntax Buffers Window Help
VIM – Search...
Find Next
Match whole word only
Direction
Up
Down
Close

```
Jan 17 12:40:21 ben kernel: [ 9339.194373] cfg80211:     (5170000 KHz - 5250000
KHz @ 40000 KHz), (300 mBi, 2000 mBm)
Jan 17 12:40:21 ben kernel: [ 9339.194377] cfg80211:     (5735000 KHz - 5835000
KHz @ 40000 KHz), (300 mBi, 2000 mBm)
Jan 17 12:40:21 ben NetworkManager[1101]: <info> (wlan0): supplicant interface s
tate: disconnected -> scanning
Jan 17 12:40:21 ben kernel: [ 9339.788312] usb 2-5: USB disconnect, device numbe
r 3
Jan 17 12:40:23 ben kernel: [ 9341.020080] usb 2-5: new high-speed USB device nu
mber 4 using ehci_hcd
Jan 17 12:40:23 ben udevd[3937]: missing file parameter for attr
Jan 17 12:40:23 ben mtp-probe: checking bus 2, device 4: "/sys/devices/pci0000:0
0/0000:00:1d.7/usb2/2-5"
Jan 17 12:40:23 ben mtp-probe: bus: 2, device: 4 was not an MTP device
Jan 17 12:40:23 ben hp-mkuri: io/hpmud/model.c 625: unable to find [s{product}]
support-type in /usr/share/hplip/data/models/models.dat
@
                                                          73,17          7%
```

Find what: ben\.*usb

[✓] Match case

Many programs have some form of regexes built in. Here, gvim is finding all USB messages for user ben in the syslog.

Ben Everard is the co-author of *Learning Python with Raspberry Pi*, soon to be published by Wiley. He's also pretty good at turning foraged fruit into alcohol.

Challenges

Test your skills by writing scripts to solve the following word puzzles

Anagrams

ainpprss
abeprrrsy
bbceirssu

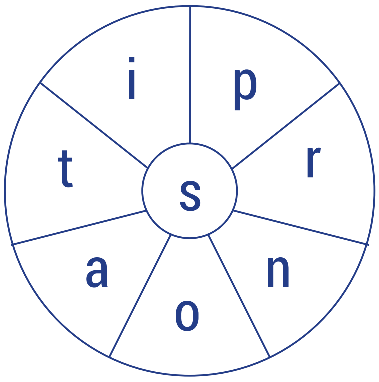
Countdown

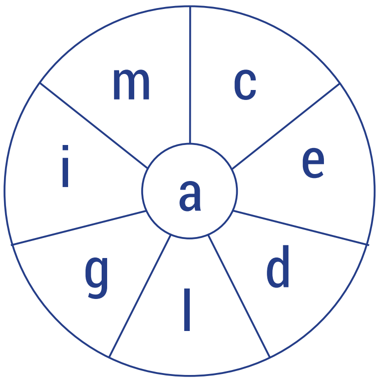
tnxpamies
dimtescat
hofanescp

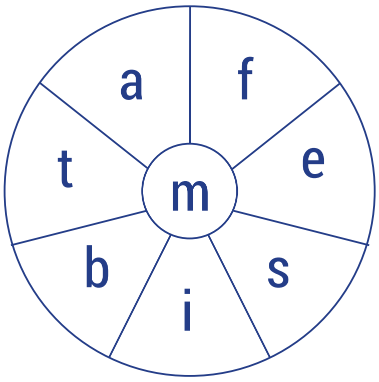
Encryption

1 2134 567894550 518824 1a4 a546b4
1234 34 5641 127 879300309
123 456 4 378936 8708a8034b

Word wheel







Word ladder

band	**brag**	**wire**
----	**----**	**----**
----	**----**	**----**
----	**----**	**----**
meat	**plan**	**pant**

Find words, win clothing!

The final challenge is something different, one we haven't covered so far: a word search. To make matters a little easier, there are only horizontal words, and none of them are backwards. The challenge is to write a Bash script that can go in the following pipe:

```
cat wordsearch.txt | bash yourscript.sh
```

and output all of the three- or more letter words from the words list (**/usr/share/dict/words** or **usr/dict/words**) that are in the file.

The word search is (on the DVD and website as **wordsearch.txt**):

```
zfghellohb
binarytwno
thisenthat
dfjunglwmr
scoffeeqwj
lhzniphoto
rlightovqx
yelsocketn
fbicycleow
ykerolbuha
```

To make things interesting, there will be two prizes, one for the smallest (in terms of characters in the Bash script), and one for the one that runs fastest (completes execution with all the words found in the shortest time). The words must be found with a form of GNU **grep** (**egrep**, **pgrep**, etc.) matching a regular expression.

There are few differences between versions of Bash on different Linuxes, so we'll be testing on a fresh install of Debian Unstable. This is only likely to matter if you're relying on particularly new or exotic features.

To be eligible, your program must be licensed under an OSI-approved licence compatible with the GPL v2 or v3. We recommend using GPLv3.

All entries must be sent to **ben@linuxvoice.com** by 31 March, and the winner will be announced in issue 3 (and on linuxvoice.com). You don't have to have bought a magazine to enter (details will be posted to the website) so feel free to pass details of the competition on others.

In the event of a tie, the solution that was sent in first will win. In all matters, the editor's decision will be final.