Regular expressions, redirections: I/P, O/P, Error

Regular Expressions (Regex)

- Think of regular expressions as a way to search for patterns in text, like a "find" tool with superpowers.

- In Linux, they’re used in commands to spot specific words or formats, like finding all lines with "error" in a file.

Regular Expressions (Regex)

- Tools You'll See It In:
  - **grep**: Searches for stuff in files.
  - **sed**: Changes text (like find-and-replace).
  - **Scripts**: You can use it in Bash to check if something matches a pattern.

Regular Expressions (Regex)

–Key Symbols:

- . (any character),
- * (zero or more),
- + (One or more)
- ^ (start of line),
- $ (end of line),
- [abc] (match a, b, or c),
- [^abc] (not a, b, or c).

Regular Expressions (Regex)

- **grep**:
  - Searches files or input for lines matching a pattern.
  - E.g.,
- grep "error.*2025" logfile.txt
- It finds lines in logfile.txt containing "error" followed by anything and then "2025".

Regular Expressions (Regex)

- Example:
  - Imagine you've got a file called notes.txt with lines like:
- I like cats
- Dogs are cool
- I like birds

Regular Expressions (Regex)

- grep "like" notes.txt
  - I like cats
  - I like birds
- Here, "like" is a simple regex pattern—it finds lines with that word.

Regular Expressions (Regex)

•Sed:

– Stream editor for transforming text.

–E.g.,

• sed 's/foo/bar/g' replaces all instances of "foo" with "bar" in a file or input.

Regular Expressions (Regex)

- **Why It’s Useful:**
  - **You can use regex to grab phone numbers, emails, or anything following a pattern.**
  - **For example, grep "[0-9]" notes.txt looks for lines with any number (0-9).**

Regular Expressions (Regex)

- First, let’s create a sample file. Open a terminal and type:

  – **echo -e "cat123\nDOG456\ndog789\nbird\n2025-01-01\nerror: crash" > test.txt**

Regular Expressions (Regex)

- This makes a file test.txt with these lines:
  - **cat123**
  - **DOG456**
  - **dog789**
  - **bird**
  - **2025-01-01**
  - **error: crash**

Example 1: Finding Lines with Numbers

- **Command:** **grep "[0-9]" test.txt**
- What It Does: Looks for any line with at least one digit (0-9).
  - Cat123
  - DOG456
  - dog789
  - 2025-01-01
- Explanation: [0-9] means "**match any single digit**." This catches all lines with numbers, skipping **"bird"** and **"error: crash."**

Example 2: Matching Words Starting with "d"

- Command: **grep "^d" test.txt**
- What It Does: **Finds lines that start with "d" (case-sensitive)**.
- Output:
- dog789
- Explanation: ^d means “the line must begin with a lowercase d.” So "DOG456" (uppercase) doesn’t match, but "dog789" does.

Example 2a: Matching Words Starting with "D" or “d”

- Command : **grep "^[dD]" test.txt**
- Output:
- DOG456
- dog789
- Explanation: [dD] means “d or D,” so now it grabs both.

Example 3: Finding Lines Ending with Numbers

- Command: **grep "[0-9]$" test.txt**
- What It Does: Matches lines that end with a digit.
- Output:
- Cat123
- DOG456
- dog789
- Explanation: [0-9]$ says “the line must end with a number.”
- The $ marks the end of the line, so "bird" and "error: crash" don’t match.

Example 4: Matching a Specific Pattern (Dates)

- Command: **grep "[0-9]\{4\}-[0-9]\{2\}-[0-9]\{2\}" test.txt**
- What It Does: Finds lines with a date-like pattern (YYYY-MM-DD).
- Output:
- 2025-01-01
- Explanation:
- **[0-9]\{4\}**: Exactly 4 digits (like "2025").
- **-**: A literal hyphen.
- **[0-9]\{2\}**: Exactly 2 digits (like "01").

Example 5: Replacing Text with sed

- Command: **sed 's/dog/PUPPY/' test.txt**
- What It Does: Replaces the first "dog" on each line with "PUPPY."
- Output:
- cat123
- DOG456
- PUPPY789
- bird
- 2025-01-01

Example 5: Replacing Text with sed

- Explanation: **s/dog/PUPPY/** means “**substitute dog with PUPPY.**”
- It’s case-sensitive, so "DOG456" stays unchanged.
- Add /g (s/dog/PUPPY/g) to replace all instances (though here it’s just one per line).

Example 6: Finding Lines with "error" and Saving Them

- Command: **grep "error.*" test.txt > errors.txt**
- What It Does: Finds lines with "error" followed by anything and saves them to errors.txt.
- Output:

–error: crash

- Explanation:
- error: Literal word "error."
- .*: Zero or more of any character after "error."
- > errors.txt: Saves the result to a file.

Example 7: Bash Script with Regex

- Let’s write a tiny script to check if a line has only numbers:

```
.echo "Enter something: "
.read input
.if [[ $input =~ ^[0-9]+$ ]]; then
echo "That’s all numbers!"
.else
echo "Not just numbers."
.fi
```

- Save it as check.sh, then run chmod +x check.sh and

Example 7: Bash Script with Regex

- Try It:
- Enter "123" → "That’s all numbers!"
- Enter "123abc" → "Not just numbers."
- Explanation:
  - ^[0-9]+$ means “start to end, only digits.”
  - ^ is the start, [0-9] is digits,
  - + means one or more, and
  - $ is the end.

Example 8: Finding Words with Three Letters

- Command: **grep "^[a-z]\{3\}$" test.txt**
- What It Does: Matches lines with exactly three lowercase letters.
- Output:
- Cat
- dog

Example 8: Finding Words with Three Letters

- Explanation:
- ^: Start of line.
- [a-z]: Any lowercase letter.
- \{3\}: Exactly 3 of them.
- $: End of line.
- So "bird" (4 letters) and "cat123" (6 characters) don’t match.

Example 9 Add “hello” to test.txt and find it

- **Step 1: Add to test.txt “hello”**
- Command: **echo "hello" >> test.txt**
- What It Does: Appends "hello" to the end of test.txt.

Example 9 Add “hello” to test.txt and find it

- Cat123
- DOG456
- dog789
- bird
- 2025-01-01
- error: crash
- hello

Example 9 Add “hello” to test.txt and find it

- Command: **grep "h.*o" test.txt**
- What It Does: Searches for lines that start with "h", followed by zero or more characters (.*), and end with "o".
- Output:
- hello

Example 9 Add “hello” to test.txt and find it

- Explanation →
- h: Starts with "h".
- .*: Any characters (or none) in between.
- o: Ends with "o".
- Only "hello" matches this pattern in our file.

Play with sed:

Replace all digits with "X."

- Command: **sed 's/[0-9]/X/g' test.txt**
- catXXX
- DOGXXX
- dogXXX
- bird
- XXXX-XX-XX
- error: crash

Explanation:

- **s/[0-9]/X/**: Substitute any digit with "X".
- **/g**: Do it globally (for every digit in the line, not just the first one).
- Lines **without digits** (like "bird" and "hello") stay **unchanged**.

Mix with Redirection –

- **Command:** **grep "[0-9]" test.txt > numbers.txt**
- What It Does: Finds all lines with at least one digit ([0-9]) and redirects the output to numbers.txt instead of showing it on the screen.
- Result: After running this, numbers.txt will contain:
- cat123
- DOG456
- dog789
- 2025-01-01

Explanation:

- [0-9]: Matches any single digit.
- Lines like "bird", "error: crash", and "hello" have no digits, so they’re skipped.
- > numbers.txt: Saves the matches to numbers.txt, overwriting it if it already exists.
- **Verification**: Run cat numbers.txt to see the result.

1. Input Redirection ("<")

- In a shell, redirection lets you control where data comes from or goes to when running commands.
- "I/P" likely means input, and
- "O/P" means output, while
- "Error" refers to error messages.
- The symbols ">" and ">>" are tools to manage these flows.

1. Input Redirection ("<")

- This tells a command to take its input from a file instead of waiting for you to type something.
- Imagine you've got a file called names.txt with this inside:
- Alice
- Bob
- Charlie

1. Input Redirection ("<")

- Now, say you use a command like **sort**, which organizes lines alphabetically.

- Normally, you'd type the names and hit Ctrl+D to finish, but with **input redirection**, you can do this:

- **sort < names.txt**

1. Input Redirection ("<")

- The output on your screen would be:
- Alice
- Bob
- Charlie
- Here, "<" feeds names.txt into sort as if you’d typed it yourself.

2. Output Redirection (">")

- This sends a command's output to a file, overwriting whatever's already there.
- use **echo**, which just prints text.
- Run this:
- **echo "Hello, world!" > greeting.txt**
- Now, check greeting.txt (with cat greeting.txt), and you'll see:
- Hello, world!

2. Output Redirection (">")

- If you run it again with different text, like:
- **echo "Goodbye!" > greeting.txt**

- The file gets overwritten, and greeting.txt now just says:
- Goodbye!

3. Appending Output (">>")

- If you want to add to a file instead of replacing it, use ">>".
- Start with that same greeting.txt containing "Goodbye!".
- **echo "See you later!" >> greeting.txt**

3. Appending Output (">>")

- Check the file again, and it’ll look like this:
- Goodbye!
- See you later!
- The ">>" tacked the new line onto the end instead of wiping out what was there.

4. Error Redirection ("2>")

- Commands can produce errors, and you might want to save those separately.

- Say you try to list a nonexistent file with ls :

- **ls nofilehere 2> errors.txt**

- You'll get an error like

–"ls: cannot access 'nofilehere': No such file or directory," but instead of seeing it onscreen, it goes into errors.txt.

4. Error Redirection ("2>")

- Check the file error.txt :
- **ls: cannot access 'nofilehere': No such file or directory**
- The "2>" targets error messages (called "standard error") specifically, leaving regular output alone.

5. Combining Them

.You can mix these too. Suppose you've got a script or command that reads input, writes output, and might fail. Create a file numbers.txt with:

.3

.1

.2

5. Combining Them

- Now run:

- **sort < numbers.txt > sorted.txt 2> sort_errors.txt**

- "< numbers.txt" feeds the numbers into sort.

- "> sorted.txt" saves the sorted result to sorted.txt.

- "2> sort_errors.txt" catches any errors (though in this case, there probably won’t be any).

5. Combining Them

- Afterward, sorted.txt will have:

- 1
- 2
- 3
- And sort_errors.txt will be empty unless something went wrong.

5. Combining Them

- Suppose you've got a file called **mixedbag.txt** with some lines— numbers and junk that won't sort nicely:

- 42
- apple
- 17
- banana
- 9

6. Combining Them

- The sort command can handle numbers with the -n flag (for numeric sorting), but it'll choke on words and spit out errors.

- Let's redirect everything:

.sort -n < mixedbag.txt > sorted.txt 2> sort_errors.txt

6. Combining Them

- Explanation:

- **< mixedbag.txt** feeds the file into sort -n, which tries to sort numerically.

- **> sorted.txt** captures the successful output (the numbers, sorted).

- **2> sort_errors.txt** grabs any error messages about the non-numeric lines.

6. Combining Them

- After running it, check the files:
- sorted.txt will have:
- 9
- 17
- 42
- The numbers got sorted, and only they made it here.

6. Combining Them

- **sort_errors.txt** will have something like:

- **sort: mixedbag.txt:2: disorder: apple**

- **sort: mixedbag.txt:4: disorder: banana**

- These are the complaints about "**apple**" and "**banana**" not being numbers, with line numbers from the original file.

Filters in Linux

- These are **command-line tools** that process text streams, files, or input in various ways.
- These filters are the **backbone** of text processing in Linux.
- They’re **simple** individually but incredibly **powerful** when combined.

Filters in Linux

- more
- less
- wc
- diff
- sort
- uniq
- paste
- cut
- nl
- tee
- head
- tail
- tr

Filters in Linux

1. more

- Purpose: Displays file content page-by-page, useful for reading long files.

- Example: Suppose you have a file story.txt with 50 lines of text.

- more story.txt

- This shows the file one screen at a time. Press Space to scroll down, q to quit.

Filters in Linux

- **Example: Viewing a Simple File**

- Scenario: You have a short text file notes.txt with the following content:

- Meeting at 10am

- Bring project updates

- Coffee provided

Filters in Linux

- **Command:** more notes.txt

- Explanation:

- This displays the entire notes.txt file. Since it’s short (fewer lines than your terminal height), it fits on one screen, and more exits immediately after showing it.

- Press q if you want to exit sooner.

- **Output:** The full file content as above.

Filters in Linux

- **Example: Paging Through a Longer File with Search**
- Scenario: You have a longer file log.txt with 50 lines, like:
- Line 1: System start
- Line 2: Checking disk
- ...
- Line 25: Error detected
- ...
- Line 50: Shutdown

Filters in Linux

- **Command**: more log.txt
- Explanation:
- This shows log.txt one screen at a time.
- Press Space to move to the next page, b to go back
- type **/Error** and press Enter to jump to the first occurrence of "Error" (Line 25). Then press n to find the next match (if any).
- Output: Starts with Line 1: System start, and you navigate manually or search for "Error" to see Line 25: Error detected.

Filters in Linux

- **Example: Combining more with Pipes and Large Input**

- Scenario: You're analyzing a directory with many files and want to paginate the detailed listing, filtering for specific file types. Imagine a directory with hundreds of files, including .txt, .log, and .sh files.

Filters in Linux

- **Command:** ls -l | grep ".txt\|.log" | more

- Explanation:

- ls -l lists all files in long format (e.g., -rw-r--r-- 1 user user 1024 Mar 1 12:00 file.txt).

- grep ".txt\|.log" filters for lines containing .txt or .log (the \| is an OR operator in grep).

Filters in Linux

- more takes this filtered output and paginates it.
- If there are 100 matching files, you'll see the first screenful (e.g., 24 lines), then press Space to continue,
- /file to search for a specific name, or
- q to quit.

Filters in Linux

- more is great for basic pagination, but its interactivity is **limited** compared to less (e.g., no smooth up-scrolling).

Filters in Linux

2. less

- Purpose: Similar to more, but more powerful—allows scrolling up and down.
- Example: Using the same story.txt:
- less story.txt
- Use arrow keys to navigate, / to search for text, and q to exit.

Filters in Linux

- **Example: Viewing a Small File**
- Scenario: You have a simple file todo.txt with a few lines:
- Buy milk
- Call mom
- Finish report

Filters in Linux

- **Command:** less todo.txt
- Explanation:
- This opens todo.txt in less. Since the file is short, it fits on one screen.
- Use the arrow keys to move up/down, or
- press q to quit.
- It’s a basic way to view a file without flooding the terminal.
- Output: The full content of todo.txt as listed above.

Filters in Linux

- **Example:** **Navigating and Searching a Larger File**
- Scenario: You have a file server.log with 100 lines, such as:
- 2025-03-01 08:00: Server started
- 2025-03-01 08:05: User login
- ...
- 2025-03-01 09:10: Error: Disk full
- ...

Filters in Linux

- **Command:** less server.log
- Explanation:
- less displays the file one screen at a time.
- Use Space to scroll down, b to scroll up, or
- /Error then Enter to search for "Error" (jumps to "Error: Disk full").
- Press n to find the next match (if any) or Shift + g to go to the end.
- This shows less’s power over more with bidirectional navigation and search

Filters in Linux

- less is more feature-rich than more, especially for navigation and real-time use.

Filters in Linux

3. wc (Word Count)

- Purpose: Counts lines, words, and characters in a file or input.
- Example:
- echo "Hello world, Linux is fun" | wc
- Output: 1 5 25 (1 line, 5 words, 25 characters, including spaces and newline).
- With a file: wc story.txt might output 50 200 1000 (lines, words, chars).

Filters in Linux

- **Example: Counting in a Simple File**
- Scenario: You have a file shopping.txt with a short list:

- Bread
- Milk
- Eggs

Filters in Linux

- **Command:** wc shopping.txt

- Explanation:

- wc counts lines, words, and characters in shopping.txt. By default, it outputs all three metrics plus the filename.

- Output: 3 3 16 shopping.txt

- 3 lines (one per item), 3 words (each line is one word), 16 characters (including newlines: "Bread\nMilk\nEggs\n" = 5 + 1 + 4 + 1 + 4 + 1 = 16).

Filters in Linux

- **Example: Counting Specific Metrics with Options**
- Scenario: You have a file poem.txt with:

- Roses are red
- Violets are blue
- Linux is great
- So are you

Filters in Linux

- **Command:** wc -l -w poem.txt
- Explanation:
- -l counts only lines, -w counts only words.
- This skips character count for a more focused result.
- Output: 4 13 poem.txt
- 4 lines (one per line), 13 words (sum of words across lines).

Filters in Linux

- wc is simple but becomes powerful when paired with pipes and filters, as in the "hard" example.

- Options like -l (lines), -w (words), and -c (characters) let you tailor it to specific needs.

Filters in Linux

4. diff

- Purpose: Compares two files and shows differences.
- Example: Create two files:
- file1.txt: "apple\nbanana\ncherry"
- file2.txt: "apple\nbanana\ndate"

- diff file1.txt file2.txt
- Output: 3c3\n< cherry\n---\n> date

Filters in Linux

- **Example:** **Comparing Two Simple Files**
- Scenario: You have two short files:
- **list1.txt:**
- Apple
- Banana
- Cherry
- **list2.txt:**
- Apple

Filters in Linux

- **Command:** diff list1.txt list2.txt
- Explanation:
- diff compares list1.txt and list2.txt line-by-line and reports differences.
- It’s a basic check to spot what’s changed.
- Output:
- 3c3 < Cherry
- [illegible]
- > Date

Filters in Linux

- **Example: Ignoring Case and Whitespace**
- Scenario: You have two files with formatting differences:
- **code1.txt:**
- print("Hello") IF x > 0
- exit
- **code2.txt:**
- PRINT("Hello") if x > 0
- exit

Filters in Linux

- **Command:** diff -i -w code1.txt code2.txt
- Explanation:
- -i ignores case differences (e.g., "print" vs. "PRINT"), -w ignores all whitespace (e.g., spaces before "if").
- This focuses on meaningful changes rather than style.
- Output:
- 2c2 < IF x > 0
- ---
- > if x > 0

Filters in Linux

5. sort

- Purpose: Sorts lines alphabetically or numerically.
- Example: Using file1.txt:
- **Command:** sort file1.txt
- Output: apple\nbanana\ncherry
- Numeric sort: sort -n for numbers.

Filters in Linux

- **Command:** sort fruits.txt

- Explanation: sort arranges the lines in fruits.txt alphabetically (ASCII order) by default. It’s a basic sort with no extra options.

- Output:

- Apple

- Banana

- Cherry

- Lines are reordered from A to Z.

Filters in Linux

- **Example: Sorting Numbers with Reverse Order**
- Scenario: You have a file scores.txt with numerical data:
- 85
- 92
- 78
- 95

Filters in Linux

- Command: sort -n -r scores.txt

- Explanation: -n sorts numerically (treats lines as numbers, not strings, so "10" comes after "2"), -r reverses the order (descending instead of ascending). This is useful for ranking scores.

- Output:

- 95

- 92

- 85

- 78

Filters in Linux

- **Example: Sorting a CSV File by a Specific Column with Unique Output**

- Scenario: You have a CSV file employees.csv with:

- Name, Age, Department

- Bob, 30, IT

- Alice, 25, HR

- Charlie, 30, IT

- Dave, 28, Finance

Filters in Linux

- **Command:** sort -t',' -k2 -n -u employees.csv

- Explanation:

–-t',' sets the field separator to a comma (for CSV),

–-k2 sorts by the second field (Age),

–-n ensures numeric sorting (25 comes before 30, not after like in string sort),

–-u removes duplicate lines (if two people had identical full lines, only one would remain).

Filters in Linux

- Output:
- Name, Age, Department
- Alice,25,HR
- Dave,28,Finance
- Bob,30,IT
- Charlie,30,IT

Filters in Linux

6. uniq

- Purpose: Removes duplicate consecutive lines (often used with sort).
- Example: Create list.txt: "apple\nbanana\napple\ncherry"
- **Command:** sort list.txt | uniq
- Output: apple\nbanana\ncherry (duplicates removed after sorting).

Filters in Linux

- **Easy: Removing Duplicates from a Simple List**
- Scenario: You have a file colors.txt with:
- Red
- Blue
- Red
- Green
- Blue

Filters in Linux

- **Command:** sort colors.txt | uniq

- Explanation:

- sort ensures duplicates are consecutive (since uniq only removes adjacent duplicates),

- and uniq then keeps only the first occurrence of each line. This is the simplest common use case.

Filters in Linux

- **Output:**

- Blue
- Green
- Red
- All unique colors, alphabetically sorted.

Filters in Linux

- **Example: Counting Occurrences**
- Scenario: You have a file votes.txt from a poll:
- Yes
- No
- Yes
- Yes
- No

Filters in Linux

- **Command:** sort votes.txt | uniq -c

- Explanation:

- -c adds a count of how many times each line appears.

- After sort groups identical lines together, uniq -c tallies them. This is great for summarizing data.

Filters in Linux

- **Output:**

- 2 No 3 Yes

- "No" appeared 2 times, "Yes" 3 times (counts are right-aligned with spaces).

Filters in Linux

7. paste

- Purpose: Merges lines from files side-by-side.
- Example:
- fruits.txt: "apple\nbanana\ncherry"
- colors.txt: "red\nyellow\npink"
- **Command:** paste fruits.txt colors.txt
- Output: apple red\nbanana yellow\ncherry pink.

Filters in Linux

- **Example: Merging Two Simple Files Side-by-Side**
- Scenario: You have two files:
- names.txt:
- Alice
- Bob
- Charlie
- ages.txt:
- 25
- 30

Filters in Linux

- **Command:** paste names.txt ages.txt
- Explanation: paste combines corresponding lines from names.txt and ages.txt with a tab as the default delimiter.
- This is a basic way to pair data from two lists.
- Output:
- Alice 25
- Bob 30
- Charlie 35
- Each line merges one name and one age, separated by a tab.

Filters in Linux

- **Example: Custom Delimiter with Multiple Files**
- Scenario: You have three files:
- items.txt: prices.txt: stores.txt:
- Pen 2 StoreA
- Book 10 StoreB
- Ruler 5 StoreC

Filters in Linux

- **Command:** paste -d',' items.txt prices.txt stores.txt

- Explanation:

- -d',' specifies a comma as the delimiter instead of the default tab.

- paste merges lines from all three files into a single line per row, useful for creating CSV-like output.

Filters in Linux

- Output:
- Pen,2,StoreA
- Book,10,StoreB
- Ruler,5,StoreC
- Each line combines an item, price, and store, separated by commas.

Filters in Linux

8. cut

- Purpose: Extracts sections from lines (by characters, fields, etc.).
- Example: With data.txt: "john,25,USA\nmary,30,UK"
- **Command:** cut -d',' -f1 data.txt
- Output: john\nmary (extracts first field, delimiter is comma).

Filters in Linux

9. nl

- Purpose: Numbers lines in a file.
- Example: Using file1.txt:
- **Command:** nl file1.txt
- Output: 1 apple\n2 banana\n3 cherry.

Filters in Linux

10. tee

- Purpose: Reads from standard input and writes to both a file and stdout.

- **Command**: echo "Hello Linux" | tee output.txt

- Output on screen: Hello Linux, and output.txt now contains "Hello Linux".

Filters in Linux

11. head

- Purpose: Shows the first few lines of a file (default: 10).
- Example: With story.txt (50 lines):
- **Command :** head -n 3 story.txt
- Output: First 3 lines of the file.

Filters in Linux

12. tail

- Purpose: Shows the last few lines of a file (default: 10).
- **Example:** tail -n 2 story.txt
- Output: Last 2 lines of the file.

Filters in Linux

13. tr

- Purpose: Translates or deletes characters.
- Example:
- echo "HELLO linux" | tr 'A-Z' 'a-z'
- Output: hello linux (converts uppercase to lowercase).
- Another use: tr -d 'o' removes all 'o' characters.

Filters in Linux

- **Example**
- Let’s say names.txt has:
- bob
- alice
- bob
- charlie
- alice

Filters in Linux

- **Command**: sort names.txt | uniq | nl
- Output:
- 1 alice
- 2 bob
- 3 charlie
- Explanation: Sorts, removes duplicates, then numbers the lines.

Filters in Linux

Scenario: You have a file colors.txt with possibly repeated entries:

- Red
- Blue
- Red
- Green
- Blue
- Yellow

Filters in Linux

- **Command:** sort colors.txt | uniq | wc -l | tee count.txt | more
- Explanation:
- sort arranges lines alphabetically, grouping duplicates together.
- uniq removes consecutive duplicates, leaving unique lines.
- wc -l counts the number of unique lines.
- tee count.txt saves the count to count.txt while passing it to the next command.
- more displays the result (though here it’s just one line, it’s useful if output grows)

Filters in Linux

- Output:
- 4
- 4 unique colors (Blue, Green, Red, Yellow).
- Also saved in count.txt.

THE END