Some useful command line tools

Nathan Warner

Northern Illinois University

Computer Science
Northern Illinois University
United States

Contents

Awk

1.1 What is awk?

Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform the associated actions.

1.2 Awk operations

(a) Scans a file line by line

(b) Splits each input line into fields

(c) Compares input line/fields to pattern

(d) Performs action(s) on matched lines

1.3 Syntax

```
awk options 'selection _criteria { action }' input-file >
```

1.4 Options

- **-f**: Reads the AWK program source from the file program-file, instead of from the first command line argument
- **-F**: Change the field separator

1.4.1 Example

```
# File that contains awk commands "filname: actions"

{ print $1 }

# File that contains data "filename: data"

0.999,0.18179,0.2711768644
1.0323,0.195853,0.47739746
1.0656,0.217119,0.6253140641

# In command line
aws -F ',' -f actions data

# Output
0.999
1.0323
1.0656
```

1.5 The print action

One of the simplest actions we can preform with awk is the print action

1.5.1 Example

```
awk '{ print }' <filename>
awk '{ print $1}' <filename>
awk '{ print $1,$3}' <filename>
```

Awk will scan the file line by line, split the input line into fields, with space as the field separator, and then perform the action print. With a bare print statement, each line will be printed. We can of course be more specific with our statement. We can access each field with $0-$n, where $0 is the entire line, and $n is a specific field number

1.6 Regex

We can as you might expect use regular expressions with awk, let's take a look

1.6.1 Whole lines

```
awk '/foo/ { print }' <filename>
```

This will print all lines that contain the word 'foo'

1.6.2 Field matching

```
man date | awk '$0~/day/'
man date | awk '$0!~/day/'
```

Here we use awk on the manual for the date command, and we output any line that contains the word day. In the second line, we do the same thing, but output any line that does not contain the word day

1.6.3 Range matching

```
man date | awk '/DESCRIPTION/,/EXAMPLES/'
```

1.7 Other Awk variables

- **NR**: Number of the current record
- **NF**: Number of fields in the current record
- **FS**: Field separator

1.8 BEGIN and END blocks

- **BEGIN{...}** Executes whatever is inside the brackets before starting to view the input file

  ```
  awk 'BEGIN { FS=":" } { print $1 }' /etc/passwd
  ```

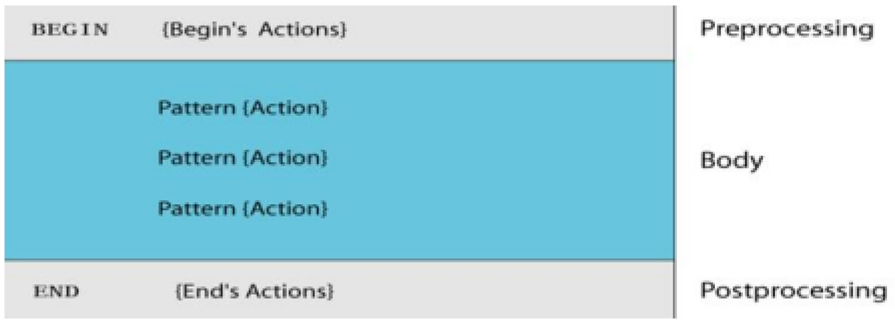
  This command sets the field separator to : before processing any lines of the input file /etc/passwd. It then prints the first field of each line, which typically contains the usernames in a Unix/Linux system.

- **END{...}** Executes whatever is inside the brackets after awk is finished reading the input file

  ```
  awk 'BEGIN { count=0 } { count++ } END { print "Number of
  ↪ lines: ", count }' /etc/passwd
  ```

  Before processing the file, it initializes a counter count to 0. For each line in /etc/-passwd, it increments count. After processing all lines, it prints the total count, effectively giving the number of lines in the file.

1.9 Typical awk script



Note:-

The symbol for comments are the same as bash, the hashtag #

1.10 If statements in awk commands

1.10.1 Syntax

awk supports conditional statements, allowing you to perform actions based on specific conditions. The syntax for an if statement in awk is similar to that in other programming languages. Here's the basic structure:

```
if (condition) {
    // actions to perform if condition is true
} else {
    // actions to perform if condition is false
}
```

Note:-

Awk does not have the concept of elseif

1.10.2 Example

```
echo -e "3\n-1\n0\n5" | awk '{ if ($1 > 0) print $1 " is
↪ positive"; else print $1 " is not positive" }'
```

1.10.3 Checking for existence in array

Similar to python, we can do the following

```
if (item in array) {
    ...
} else {
    ...
}
```

1.11 For loops in awk commands

1.11.1 Syntax

```
for (initialization; condition; increment) {
    // Code block to execute
}
```

1.11.2 Example

```
awk 'BEGIN {
    for (i = 1; i <= 5; i++) {
        print "Number is", i
    }
}
```

1.11.3 Iterating Over an Array

```
for (index in array) {
    // Code block using array[index]
}
```

1.12 The while loop in awk commands

1.12.1 Syntax

```
while (condition) {
    // Code block to execute
}
```

1.12.2 Example

```
awk '{
    while ($0 !~ /stop/) {
        print "Processing:", $0
        if (getline <= 0) break
    }
}' input.txt
```

1.13 Awk variables

In awk, we can define variables with an assignent statement. We can define integers, strings, and arrays. Variables only come into existence when first used. Variables are initialized to either 0 or ""

1.14 Awk arrays

Awk allows the creation of one dimensional arrays, the index (and the elements) can be either a number or a string. Unlike arrays in *c*, we need not declare the size or type of the array. The array elements are created when first used and initialized either 0 or ""

Note:-

When we use strings as indicies, we are essentially creating a map

1.14.1 Syntax

```
arrayName[index] = value
```

1.14.2 Examples

```
arr[1] = "some value"
arr[2] = 25
arr["foo"] = "bar"
```

1.14.3 Example: Process sales data

Suppose we have the data file

1	clothing	3141
1	computers	9161
1	textbooks	21321
2	clothing	3252
2	computers	12321
2	textbooks	15462
2	supplies	2242

We wish to output a summary of department sales, then we could write an awk program

```
# p.awk (awk -f p.awk data)
{
    deptsales[$2] += $3
}
END {
    for (x in deptsales) {
        print x, deptsales[x]
    }
}
```

1.15 Builtin Functions

1.15.1 Arithmetic

- sqrt
- rand

1.15.2 String

- index
- length
- split
- substr
- sprintf
- tolower
- toupper

1.15.3 Misc

- system
- systime

1.16 The split function

1.16.1 Syntax

```
split(string, array, fieldsep)
```

The split function divides **string** into pieces separated by **fieldsep** and stores the pieces in **arary**. If **fieldsep** is omitted, the value of FS is used.

1.16.2 Example

```
split("26:Miller:Comedian", fields, ":")
```

Now we have a fields array with the contents of the string

1.17 The gsub function

The gsub function in AWK is a powerful tool for performing global search and replace operations on strings. It searches for all occurrences of a pattern in a string and replaces them with a specified replacement text.

1.17.1 Syntax

```
gsub(regexp, replacement, target)
```

Stream Editor (sed)

2.1 What is sed?

Sed is a non-interactive stream editor. We use sed to

- Automatically perform edits on files
- Simplify doing the same edit on multiple files
- Write conversion programs
- Do editing operations from shell script

2.2 Syntax

```
sed -e 'address command' input_file # (inline script)
sed -f script.sed input_file # (script file)
```

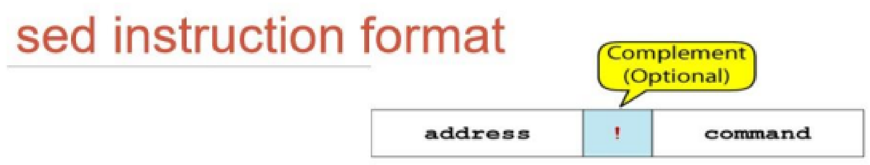
2.3 How does sed work?

Sed reads files line by line, each lien of input is copied into a temporary buffer called the **pattern space**, then the editing instructions are applied to line in the pattern space. Next, the line is sent to output (unless -n was used). Last, the line is removed from the pattern space. Sed does this for all lines until the end of the file

Note:-

Input file is unchanged unless -i option is used

2.4 Instruction format



The address determines which lines in the input file are to be processed by the commands, if no address is given, then the command is applied to each input line

2.4.1 Address types

- Single-line address
- Set-of-lines address
- Range address

2.4.2 Single-line address

A Single-line address specifies only one line in the input file

Note:-

The dollar sign denotes the last line of input file

```
sed -n -e "3 p" infile # Show only line 3
sed -n -e "$ p" inflie # Show last line
sed -e "10 s/endif/fi/" infile # Substitute "endif" with "fi" on
    ↪ line 10
```

2.4.3 Set-of-lines address

With this, we can use regex to match lines

- Written between two slashes
- Process only lines that match
- May match several lines
- Lines don't have to be consecutive

```
sed -i -e "/Key/ s/more/other" infile
sed -n -e "/r..t/ p" input-file
```

2.4.4 Range address

Defines a set of consecutive lines. Format: startAddr,endAddr (inclusive)

Examples:

- 10,50
- 10,/funny/

2.4.5 Address complement

Address with an exclamation point (!). Command applies to lines that **don't** match the address

```
sed -n -e '/Obsolete/!p' infile # (print lines that do not
↪  contain "Obsolete")
```

2.5 Sed commands

2.5.1 Modify

- insert
- append
- change
- delete
- substitute

2.5.2 Input/Output

- Next, print
- Read, write

2.5.3 Other

- quit

2.6 Commands i,a,c

- **i** adds lines before the address
- **a** adds lines after the address
- **c** replaces an entire matched line with new text

2.6.1 Syntax

```
[address] i\
text
```

2.7 Delete command

Deletes the entire pattern space

Note:-

Commands following the delete command are ignored since the deleted text is no longer in the pattern space

2.7.1 Syntax

```
[address] d
```

2.8 Substitute command (s)

2.8.1 Syntax

```
[address] s/search/replacement/[flag]
```

tr (translate)

The tr command in Unix and Unix-like operating systems is a utility for translating or deleting characters. It reads from standard input and writes to standard output. The tr command is commonly used in scripts and command-line operations to perform simple text transformations.

3.1 Syntax

```
tr [OPTION]... SET1 [SET2]
```

- **SET1:** The set of characters to be replaced or deleted.
- **SET2:** The set of characters to replace the characters in SET1 (if provided).

3.2 Common Options

- **-d:** Delete characters in SET1, do not translate.
- **-s:** Squeeze repeated characters in SET1 into a single character.
- **-c:** Complement the characters in SET1.

3.3 Basic Example: Convert lowercase to uppercase

```
echo "Hello World" | tr "[:lower:]" "[:upper:]"
```

3.4 Basic Example: Deleting characters

```
echo "Hello World" | tr -d "aeiou"
```

3.5 Basic Example: Squeezing Characters

```
echo "Hello     world" | tr -s " "
```

3.6 Basic Example: Complimenting characters

```
echo "abcd123efg" | tr -c "[:digit:]" "-"
```

Replaces all characters **except** digits with a hyphen:

rev (reverse)

4.1 Reversing Strings

To reverse a string, we can use the rev command. The rev command words similar to the bc command

```
echo "string" | rev
```

4.2 Reversing arrays

```
declare -a arr1=("a" "b" "c")
declare -a arr2=($(echo "${arr1[@]}" | rev))

echo "${arr1[@]}" // Output: a b c
echo "${arr2[@]}" // Output: c b a
```

fold

The fold command in Unix-like operating systems is used to wrap each input line to fit within a specified width. It's particularly useful for breaking long lines of text into shorter, more manageable pieces. By default, fold breaks lines at a width of 80 characters, but this can be adjusted using options.

5.1 Syntax

```
fold [OPTION]... [FILE]...
```

5.2 Common Options

- **-w, --width=WIDTH:** Specify the maximum line width. Lines longer than this width will be broken.
- **-s, --spaces:** Break lines at word boundaries (spaces) instead of exactly at the specified width.

5.3 Example

```
echo "This is a long line that needs to be wrapped" | fold -w 20
```

5.4 Example

```
str="Helloworld"
str=$(fold -w 5 <<< $str)

echo $str // Output: Hello world
```

bc (Basic Calculator)

The bc command in Unix-like systems is an arbitrary precision calculator language, which is useful for performing mathematical operations that go beyond the capabilities of standard shell arithmetic. Here's a detailed explanation of the bc command, how it works, and some examples to illustrate its use.

6.1 Basic Use

You can use bc interactively by simply typing bc in the terminal, or you can use it non-interactively within a script or from the command line by echoing expressions into it.

```
echo "expression" | bc

echo "2+2" | bc
```

6.2 Exponentiation

We use the carrot ($\wedge$) for exponentiation

```
echo "2^64" | bc
```

6.3 Floating-Point Arithmetic

```
echo "scale=2; 5/3" | bc
```

Where *scale=2* Sets the number of decimal places to 2

Basename

The basename command in Unix-like operating systems is used to strip the directory and suffix from filenames. It's particularly useful in shell scripts for manipulating paths and extracting the filename or the file's base name.

7.1 Syntax

```
basename NAME [SUFFIX]
```

- **NAME:** The name of the file (can include the path).
- **SUFFIX:** An optional suffix to remove from the base name.

7.2 Example

```
basename /usr/local/bin/script.sh
// Output: script.sh
```

7.3 Example

```
# Full path to a file
full_path="/usr/local/bin/my_script.sh"

# Get the base name without directory
file_name=$(basename "$full_path")

# Get the base name without directory and suffix
base_name=$(basename "$full_path" .sh)
```

expr

The expr utility in Unix-like systems is used for evaluating expressions. It can perform a variety of operations including arithmetic, string operations, and logical comparisons. While it is not as commonly used as other methods in modern scripting due to the availability of more powerful tools and built-in shell capabilities, it remains a useful tool for certain tasks.

8.1 Syntax

```
expr EXPRESSION
```

8.2 Arithmetic operations

```bash
#!/usr/bin/env bash

# Addition
sum=$(expr 5 + 3)
echo "5 + 3 = $sum"

# Subtraction
difference=$(expr 10 - 4)
echo "10 - 4 = $difference"

# Multiplication (note the backslash to escape the asterisk)
product=$(expr 7 \* 2)
echo "7 * 2 = $product"

# Division
quotient=$(expr 20 / 5)
echo "20 / 5 = $quotient"

# Modulus
remainder=$(expr 11 % 3)
echo "11 % 3 = $remainder"
```

8.3 String operations

expr can also handle string operations, such as finding the length of a string, extracting substrings, and locating substrings within a string.

8.3.1 String length

```bash
#!/usr/bin/env bash

str="Hello, world!"
length=$(expr length "$str")
echo "Length of '$str' is $length"
```

8.3.2 Substring Extraction

```bash
#!/usr/bin/env bash

str="Hello, world!"
substring=$(expr substr "$str" 8 5)
echo "Substring of '$str' starting at position 8 with length 5
↪  is '$substring'"
```

8.4 Logical Comparisons

8.4.1 Numeric comparisons

```bash
#!/usr/bin/env bash

a=10
b=20

if [ $(expr $a \< $b) -eq 1 ]; then
  echo "$a is less than $b"
fi

if [ $(expr $a \> $b) -eq 0 ]; then
  echo "$a is not greater than $b"
fi
```

8.4.2 String comparisons

```bash
#!/usr/bin/env bash

str1="abc"
str2="xyz"

if [ $(expr "$str1" = "$str2") -eq 0 ]; then
  echo "'$str1' is not equal to '$str2'"
fi

if [ $(expr "$str1" \> "$str2") -eq 1 ]; then
  echo "'$str1' is greater than '$str2'"
else
  echo "'$str1' is not greater than '$str2'"
fi
```

8.4.3 Combining expressions

```bash
#!/usr/bin/env bash

a=5
b=10
c=15

# Combining arithmetic operations
result=$(expr $a + $b \* $c)
echo "Result of $a + $b * $c = $result"

# Combining logical comparisons
if [ $(expr $a \< $b \& $b \< $c) -eq 1 ]; then
  echo "$a is less than $b and $b is less than $c"
fi
```

cut

The cut utility in Unix and Bash scripting is used for extracting sections from each line of input data (usually a text file). It is often used in data processing and text manipulation tasks.

9.1 Syntax

```
cut [OPTION]... [FILE]...
```

If no file is specified, cut reads from the standard input.

9.2 Common Options

- **-b**: Selecting by byte position

  ```
  cut -b LIST [FILE]...
  ```

  - LIST specifies the byte positions to extract.
  - **Example:** cut -b 1-3 file.txt extracts bytes 1 through 3 from each line.

- **-c**: Selecting by Character Position

  ```
  cut -c LIST [FILE]...

  // Example
  echo "abcdef" | cut -c 2-4

  // Output: bcd
  ```

  - LIST specifies the character positions to extract.
  - **Example:** cut -c 1-3 file.txt extracts characters 1 through 3 from each line.

- **-f**: Selecting by Field

  ```
  cut -f LIST [-d DELIMITER] [FILE]...

  // Example:
  echo "name,age,location" | cut -f 1,3 -d ','

  // Output: name,location
  ```

  - LIST specifies the fields to extract, where fields are separated by a delimiter.
  - -d DELIMITER specifies the delimiter that separates fields. The default delimiter is a tab.
  - **Example:** cut -f 1,3 -d ',' file.csv extracts the first and third fields from a CSV file.

- **-s**: Suppressing Lines without Delimiters

```
cut -s -f LIST [-d DELIMITER] [FILE]...
```

  - The -s option suppresses lines that do not contain the delimiter.
  - **Example:** cut -s -f 2 -d ':' file.txt extracts the second field but skips lines without a colon.

9.3 Reading from a file

Assume we have the file

```
// data.txt

alpha,beta,gamma,delta
epsilon,zeta,eta,theta
```

We can extract the second and fourth fields using:

```
cut -f 2,4 -d ',' data.txt

// Output:
beta,delta
zeta,theta
```

9.4 More on -s

The -s option in the cut command is used to suppress lines that do not contain the delimiter specified with the -d option. When you use the -s option, cut will ignore and not output any lines from the input that do not include the specified delimiter.

9.4.1 Without -s

The -s option in the cut command is used to suppress lines that do not contain the delimiter specified with the -d option. When you use the -s option, cut will ignore and not output any lines from the input that do not include the specified delimiter.

```
// example.txt

apple:banana:cherry
date:fig:grape
kiwi
lemon:mango:nectarine
```

If you run cut without the -s option to extract the second field:

```
cut -d ':' -f 2 example.txt
```

The output will include all lines, with empty lines for those that do not have the delimiter:

```
// Output

banana
fig

mango
```

9.4.2 With -s Option

When you use the -s option, cut will skip lines that do not contain the delimiter:

```
cut -d ':' -f 2 -s example.txt
```

The output will exclude the line kiwi because it does not contain the : delimiter:

```
// Output:

banana
fig
mango
```

9.5 Exercise: Extracting Data from a Sample File

Assume a file named students.txt with the following content

```
// Student.txt

ID,Name,Age,Grade,Email
1,John Doe,20,A,john.doe@example.com
2,Jane Smith,21,B,jane.smith@example.com
3,Emily Jones,19,A,emily.jones@example.com
4,Michael Brown,22,C,michael.brown@example.com
5,Jessica White,20,B,jessica.white@example.com
```

9.5.1 1. Extract the Names of All Students

Use the cut command to extract the Name field from the students.txt file.

```
cut -f 2 -d ',' studets.txt
```

tac

The tac command in Unix-like operating systems is used to concatenate and print files in reverse. Unlike the cat command, which outputs the contents of a file in the same order, tac outputs the contents in reverse order, meaning it prints the last line first and the first line last.

10.1 Syntax

```
tac [OPTION]... [FILE]...
```

10.2 Options

- **-b, –before:** Attach the separator before each line instead of after.
- **-r, –regex:** Treat the separator string as a regular expression.
- **-s, –separator=STRING:** Use STRING as the line separator instead of the default newline.

sort

The sort command in Unix-like operating systems is used to sort lines of text files. It is a powerful tool that can sort data in various ways based on different criteria such as numerical order, alphabetical order, and more.

11.1 Syntax

```
sort [OPTION]... [FILE]...
```

If no file is specified, sort reads from standard input.

11.2 Common Options

- **-n, --numeric-sort**: Sorts the input numerically.

  ```
  sort -n file.txt
  ```

- **-r, --reverse**: Reverses the sorting order.

  ```
  sort -r file.txt
  ```

- **-k, --key=POS1**[**,POS2**]: Specifies the field separator character.

  ```
  sort -k 2 file.txt
  ```

- **-t, --field-separator=SEP**: Specifies the field separator character.

  ```
  sort -t ',' -k 2 file.csv
  ```

- **-u, --unique**: Outputs only the first of an equal run (removes duplicates).

  ```
  sort -u file.txt
  ```

- **-o, --output=FILE**: Writes the result to the specified file.

  ```
  sort -o sorted.txt file.txt
  ```

- **-b, --ignore-leading-blanks**: Ignores leading blanks.

```
sort -b file.txt
```

- **-f, --ignore-case**: Ignores case differences.

```
sort -f file.txt
```

- **-M, --month-sort**: Sorts by month name.

```
sort -M file.txt
```

uniq

The uniq command in Unix-like operating systems is used to filter out or report repeated lines in a file. It is typically used in conjunction with the sort command because uniq only detects adjacent duplicate lines. The sort command ensures that all duplicate lines are next to each other

12.1 Syntax

```
uniq [OPTION]... [INPUT [OUTPUT]]
```

- **INPUT:** The input file to process. If omitted, uniq reads from standard input.
- **OUTPUT:** The file to write the results to. If omitted, uniq writes to standard output.

12.2 Common options

- **-c, --count**: Prefixes lines with the number of occurrences.

  ```
  uniq -c file.txt
  ```

- **-d, --repeated**: Only prints duplicate lines.

  ```
  uniq -d file.txt
  ```

- **-u, --unique**: Only prints unique lines.

  ```
  uniq -u file.txt
  ```

- **-i, --ignore-case**: Ignores differences in case when comparing lines.

  ```
  uniq -i file.txt
  ```

- **-f N, --skip-fields=N**: Ignores the first N fields when comparing lines.

  ```
  uniq -f 1 file.txt
  ```

- **-s N, --skip-chars=N**: Ignores the first N characters when comparing lines.

  ```
  uniq -s 2 file.txt
  ```

- **-w N, --check-chars=N**: Compares no more than N characters in lines.

```
uniq -w 5 file.txt
```

split

The split command in Unix-like operating systems is used to split a file into smaller files. It is a useful tool for managing large files by breaking them down into more manageable pieces. The split command provides various options to customize the splitting process, such as specifying the size of each split file, the number of lines per split file, or using a custom suffix for the split files

13.1 Syntax

```
split [OPTION]... [INPUT [PREFIX]]
```

- **INPUT:** The input file to be split. If omitted, split reads from standard input.
- **PREFIX:** The prefix to use for the split files. The default prefix is x.

13.2 Common options

- **-l, --lines=NUMBER:** Splits the file into pieces with NUMBER lines each.

  ```
  split -l 1000 input.txt
  ```

- **-b, --bytes=SIZE:** Splits the file into pieces of SIZE bytes each.

  ```
  split -b 1M input.txt
  ```

- **-b, --bytes=SIZE:** Splits the file into pieces of SIZE bytes each.

  ```
  split -C 1M input.txt
  ```

- **-d, --numeric-suffixes:** Uses numeric suffixes instead of alphabetic suffixes

  ```
  split -d input.txt
  ```

- **-a, --suffix-length=N:** Uses suffixes of length N.

  ```
  split -a 3 input.txt
  ```

- **--additional-suffix=SUFFIX:** Adds an additional suffix to the split file names.

```
split --additional-suffix=.txt input.txt
```

- **--number=CHUNKS:** Splits the file into CHUNKS number of files.

```
split --number=4 input.txt
```

shuf

The shuf command in Unix-like operating systems is used to generate random permutations of input lines. It can be used to shuffle lines in a file, select random lines, or generate random numbers within a specified range. shuf is part of the GNU core utilities and is commonly used for tasks that require randomization.

14.1 Syntax

```
shuf [OPTION]... [FILE]
```

If no file is specified, shuf reads from standard input.

14.2 Common options

- **-e, --echo:** Treats each command-line argument as an input line.

  ```
  shuf -e item1 item2 item3
  ```

- **-i, --input-range=LO-HI:** Acts as if input came from a file containing the range of numbers from LO to HI, one per line.

  ```
  shuf -i 1-10
  ```

- **-n, --head-count=COUNT:** Outputs at most COUNT lines.

  ```
  shuf -n 5 file.txt
  ```

- **-o, --output=FILE:** Writes the output to FILE instead of standard output.

  ```
  shuf -o shuffled.txt file.txt
  ```

- **-r, --repeat:** Repeats the input lines indefinitely.

  ```
  shuf -r -n 10 file.txt
  ```

- **--random-source=FILE:** Uses FILE as a source of random numbers instead of `/dev/urandom`.

  ```
  shuf --random-source=random.txt file.txt
  ```