ETH Zürich
263-3850-00

Informal Methods
Autumn 2023

Assignment 1 — the Unix `diff` utility

1 Why `diff`?

Formal-Methods "over-enthusiasts" seem to think that *anything* can and should be formalised, no matter how subjective it seems, and that formalisation can and should be taken all the way down to the bedrock, no matter how fundamental. In this course we try to take a more moderate view; and part of the point of this (first) assignment is to demonstrate how there is room for both formality and informality.

The `diff` utility compares two text files and –in some sense– discovers a "small" set of text-editing style changes that will transform one into the other. It's the core of version-control systems with which many people can work collaboratively and in parallel on a common codebase. Its implementation, however, is not at all trivial: there have been scholarly papers in journals about how it works, and even a Ph.D. thesis on its essential algorithm [a,b]. [1]

The relevance of `diff` for us is that at its core is a tough, technical, formally specified algorithm, one that sits like a small "gem" within a larger, more ad-hoc setting of informal software-engineering techniques and rules of thumb: we have two quite distinct features. That gem is the part that implements the "in some sense" from the previous paragraph, and both its specification and its implementation are interesting.

Here we will construct both the gem *and* its setting. The point of such a heterogeneous assignment is to show that the skills of a successful software engineer have to include both ends of the spectrum: a deep understanding of the theory of static reasoning necessary to construct, from first principles, algorithms that are really intricate; and a broad command of general software engineering techniques, learned from others and part of a practitioner's toolkit, necessary to make programs usable and resource-efficient.

2 How `diff` works

Suppose we have two text files `Fm` and `Fn`, and we want to figure out a small set of editing changes that will take the first to the second; say that the files have M, N lines respectively. The `diff` utility works more or less as follows:

[1] [a] An algorithm for differential file comparison. J Hunt and D McIlroy.
`https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=967539`

[b] The longest common subsequence problem. D Hirschberg. Doctoral thesis. Reference [7] in [a] above.

$\mathcal{O}(M \min N)$ Scan `Fm` and `Fn` forwards, from the beginning, to determine their common prefix; remove it from both.

$\mathcal{O}(M \min N)$ Scan `Fm` and `Fn` backwards, from the end, to determine their common suffix; remove it from both.

$\mathcal{O}(M)$ Hash the (remaining) lines of `Fm`; associate the line numbers with the hashes.

$\mathcal{O}(M)$ Write the *actual* lines to disk; keep the hashes in memory.

$\mathcal{O}(M \log M)$ Sort the hash×line-number pairs on the hash value.

$\mathcal{O}(N)$ As for `Fm`, hash the (remaining) lines of `Fn`; associate the line numbers with the hashes.

$\mathcal{O}(N)$ Write the *actual* lines to disk; keep the hashes in memory.

$\mathcal{O}(N \log N)$ Sort the hash×line-number pairs on the hash value.

$\mathcal{O}(M \max N)$ $*$ Find the line numbers of equal lines in the two files, using a relational join: use the actual lines (on disk) to deal with hash collisions. This will be a set of line-number pairs: call it the *correspondence*.

$\mathcal{O}((M \max N) \log(M \max N))$ $*$ Sort the correspondence ascending on the first element of the pairs.

$\mathcal{O}(M \log N)$ $*$ **Find a longest ascending subsequence of the second elements of the (sorted) correspondence.**

$\mathcal{O}(M \max N)$ Use that longest ascending subsequence to construct a "small" editing recipe to get from `Fm` to `Fn`, based on the $*$ *longest common subsequence*.

Somebody without a good computer-science education will probably not be able to do any of this; somebody who *has* been through a good curriculum will be able to manage most of it. But we want to manage *all* of it, including the step in bold — though we will be building only a prototype, not the real thing as above, by concentrating on the steps marked $*$.

3 The assignment in detail

The assignment is to build a prototype of (most of) the Unix `diff` command.

What's here is in two very different parts. The first part is small, intricate and not at all obvious.[2] It requires careful, systematic and static reasoning to produce reliably the component that will be crucial for the assignment as a whole. The second part, on the other hand, is –basically– command-file hacking.

In the first part, you will write –in some procedural programming language (of your choice)– a small Unix-style "filter" program that inputs, and outputs, a sequence of integers: the output is to be a strictly ascending subsequence

[2] It is what was was discussed in the lectures of Week 5 and 6.

of the input that is as long as it is possible to be: we say that it is *maximal* (cannot be made longer) or that it has *maximum* length. This part is based on the techniques we have covered in the lectures so far, including invariant synthesis and "automatic" introduction of data structures, and corresponds to the boldface step in Sec. 2 above.

In the second part you are led through an exercise in prototyping. With the program you have written and various (other) Unix system utilities you will construct a cheap mock-up of most of the rest of the `diff` program. This corresponds to the remaining steps in Sec. 2. (We won't do the very last step of outputting editing instructions: for us, a longest common subsequence will be enough.)

To bring out your specific tasks from the surrounding text, an indication is in each case placed in the margin. ◁

4 Part 1, the gem: A longest upsequence

We say that a *longest upsequence* of a given sequence A of natural numbers is a strictly ascending subsequence of A of maximum length. (There might be several of that length; but there is no other strictly ascending subsequence that is longer.)

We developed in lectures a program to calculate this maximum length; code up a program in some non-obscure procedural language to do that. ◁ **Do not use recursion — we are for the moment concentrating on how to reason about loops.** You are advised to work out the binary search within it "from scratch," i.e. using its specification and a new invariant to make new code, rather than by tweaking code we have already written.

Make sure your program runs!

Include in your source file a comment stating what your invariant(s) are: you can write them in English, as long as they are reasonably clear. If you give them names, like

```
// Invariant I1: The variable c is the current length of...
```

then you can at other points in your code include comments like

```
// I1 is true here.
```

and

```
// Now re-establish I1 for n+1.
```

All of this is helpful (actually very important) for a "code-auditor" (like me) to see that your program likely to be correct. If I am reasonably confident of that, then probably I won't even run it.

4.1 Locating a longest upsequence, not just its length

We must extend the algorithm in lectures so that it reads a sequence of numbers from standard input and produces on standard output a longest upsequence of that input sequence (rather than just the length of one). Recall that in the algorithm we developed in lectures an auxiliary array M was used to record least-and-last elements of upsequences of A: in fact the *meaning* of M is that for $0<k\leq m$ we have that $M[k]$ is the minimum last element of any upsequence of length k which occurs in the suffix $A[0,n)$ "considered so far", where m is the maximum length of any upsequence in $A[0,n)$.

To find the sequence itself, you may need to introduce a second auxiliary array that contains sufficient extra information to figure out *afterwards* just what an actual maximum-length subsequence actually is — i.e. not just its length, but its actual elements.

Thus once the greatest length has been found, the extra information in your array is used –in a second pass– to find an upsequence of that length. Extend your algorithm to do this; *state any invariants you use — they will be properties of that second array (if you needed one).* Note that the $\mathcal{O}(M \log N)$ complexity of the overall algorithm must be preserved. In particular, you should (probably) not do this part using lists of partial subsequences, since they are too expensive in time and space to use in an $N \log N$ algorithm like this one. ◁

Extend your program with the extra sequence-finding code, and test it. Your program code will be part of what you hand in. Don't forget to include your invariants as comments. ◁

5 Part 2, the setting: A maximal common subsequence

There is a connection between the longest upsequence (of one file) and the longest *common* subsequence (between two files). Here we explore what it is, by example.

We consider two files *abcdedec* and *ceedba* where each letter stands for a whole line: thus we imagine the first file has 8 lines, and the second has 6. To get the longest common subsequence from a longest upsequence we construct the (1-origin) correspondence (defined in Sec. 2 above and Sec. 5.1 below)

$$(1,6)\ (2,5)\ (3,1)\ (8,1)\ (4,4)\ (6,4)\ (5,2)\ (5,3)\ (7,2)\ (7,3)\ , \tag{1}$$

then sort it on its first element to get

$$(1,6)\ (2,5)\ (3,1)\ (4,4)\ (5,3)\ (5,2)\ (6,4)\ (7,3)\ (7,2)\ (8,1)\ , \tag{2}$$

then extract the sequence of second components

$$6,5,1,4,3,2,4,3,2,1\ ,$$

and finally isolate a longest upsequence within that: one of them is $1, 3, 4$; another is $1, 2, 4$; and a third is $1, 2, 3$. Those longest upsequences of indices give longest common subsequences *ced* (twice) and *cee*.

5.1 Constructing a prototype

Given two textfiles, let their *correspondence* be a sequence whose elements are the pairs of indices of equal lines, one from each of the original files. (Recall (1) above.)

Show how to use the Unix commands `nl`, `sort` and `join` to produce such a correspondence for two textfiles. Illustrate your method with examples; they will be part of what you hand in. ◁

The manual entries for `nl`, `join` and `sort` are supplied in Section 8.

5.2 "Combing" a file

To produce on standard output those lines of *this* file whose line numbers appear ascending in *that* file, we can use the command `join` again, as in the following example where we are still using 1-indexing:

this	*that*	*standard output*
this	2	statement
statement	5	removed
mustn't	6	from
be	8	context.
removed		
from		
its		
context		

6 The `diff` prototype

Use the results of Secs. 4 and 5.1 to construct a shell script `lcss` (longest common subsequence) such that ◁

```
lcss first second
```

produces on its standard output a longest common subsequence of lines from the two files `first` and `second`.

Note that the "sorted correspondence" at (2) above is a sequence of pairs, whereas your program of Sec. 5.1 expects a sequence of single elements. Thus you must adjust either the correspondence or your program; for the former, you could use the Unix command

```
awk '{print $2}' .
```

7 What you should hand in, when it's due and how it will be evaluated

The following should be submitted via Moodle (use the link on the course website at `https://moodle-app2.let.ethz.ch/course/view.php?id=21255`):

1. An annotated program that prints on standard output an *actual* longest upsequence of an arbitrary sequence of natural numbers given on standard input, together with brief instructions on how I can run it. This should be a text-file (possibly with a dot-suffix in its name), so that I can run it if I suspect it's incorrect.

2. Examples of correspondences constructed from textfiles using Unix commands. These should be text files.

3. A shellfile that implements `lcss`, as described in Sec. 6. This should be a text file.

4. Sample runs of the program and shellfile, short enough to be checked easily. These should be text files.

The assignment is due (in Moodle) by **11:55 p.m. on Friday, November 3, 2023**.

Make sure all the individual files, in the zip file, have *within them* at the top your student number (but *not* your name).

The main criterion for evaluation will be whether it's possible by inspection *only* of your code and its comments (invariants, assertions etc) to be convinced that it works. Programs will be tested, if necessary, only to prove that they *don't* work (if in fact their code and documentation suggests that they might not).

Keep your work short and to the point. The length-finding code (without comments) can be as short as about 15 lines; the code to recover the actual sequence, in a second pass, might take another 10 lines; and the reading and printing of numbers perhaps another 10. The shellfile (again without its comments) is roughly 10 lines. If your approach is substantially longer, there's a risk that it will not be understood.

Finally — though the written work asked for is very short, it does require a lot of thought (for the first part) and a lot of experimentation (for the second). If you're having trouble, don't leave it too late to ask for help.

8 Manual entries

```
NL(1)                    BSD General Commands Manual                    NL(1)

NAME
     nl -- line numbering filter

SYNOPSIS
     nl [-p] [-b type] [-d delim] [-f type] [-h type] [-i incr] [-l num]
        [-n format] [-s sep] [-v startnum] [-w width] [file]

DESCRIPTION
     The nl utility reads lines from the named file or the standard input if
     the file argument is ommitted, applies a configurable line numbering fil-
     ter operation and writes the result to the standard output.

     The nl utility treats the text it reads in terms of logical pages.
     Unless specified otherwise, line numbering is reset at the start of each
     logical page.  A logical page consists of a header, a body and a footer
     section; empty sections are valid.  Different line numbering options are
     independently available for header, body and footer sections.

     The starts of logical page sections are signalled by input lines contain-
     ing nothing but one of the following sequences of delimiter characters:

           Line      Start of
           \:\:\:    header
           \:\:      body
           \:        footer

     If the input does not contain any logical page section signalling direc-
     tives, the text being read is assumed to consist of a single logical page
     body.

     The following options are available:

     -b type      Specify the logical page body lines to be numbered.  Recog-
                  nized type arguments are:

                  a       Number all lines.

                  t       Number only non-empty lines.

                  n       No line numbering.

                  pexpr   Number only those lines that contain the basic regu-
```

lar expression specified by expr.

The default type for logical page body lines is t.

-d delim Specify the delimiter characters used to indicate the start of a logical page section in the input file. At most two characters may be specified; if only one character is specified, the first character is replaced and the second character remains unchanged. The default delim characters are ‘‘\:’’.

-f type Specify the same as -b type except for logical page footer lines. The default type for logical page footer lines is n.

-h type Specify the same as -b type except for logical page header lines. The default type for logical page header lines is n.

-i incr Specify the increment value used to number logical page lines. The default incr value is 1.

-l num If numbering of all lines is specified for the current logical section using the corresponding -b a, -f a or -h a option, specify the number of adjacent blank lines to be considered as one. For example, -l 2 results in only the second adjacent blank line being numbered. The default num value is 1.

-n format Specify the line numbering output format. Recognized format arguments are:

ln Left justified.
rn Right justified, leading zeros suppressed.
rz Right justified, leading zeros kept.

The default format is rn.

-p Specify that line numbering should not be restarted at logical page delimiters.

-s sep Specify the characters used in separating the line number and the corresponding text line. The default sep setting is a single tab character.

-v startnum Specify the initial value used to number logical page lines; see also the description of the -p option. The default startnum value is 1.

-w width Specify the number of characters to be occupied by the line number; in case the width is insufficient to hold the line number, it will be truncated to its width least significant digits. The default width is 6.

ENVIRONMENT

The LANG, LC_ALL, LC_CTYPE and LC_COLLATE environment variables affect the execution of nl as described in environ(7).

EXIT STATUS

The nl utility exits 0 on success, and >0 if an error occurs.

SEE ALSO

jot(1), pr(1)

STANDARDS

The nl utility conforms to IEEE Std 1003.1-2001 (‘‘POSIX.1’’).

HISTORY

The nl utility first appeared in AT&T System V.2 UNIX.

BUGS

Input lines are limited to LINE_MAX (2048) bytes in length.

NAME
sort - sort lines of text files

SYNOPSIS
sort [OPTION]... [FILE]...

DESCRIPTION
Write sorted concatenation of all FILE(s) to standard output.

Mandatory arguments to long options are mandatory for short options too. Ordering options:

-b, --ignore-leading-blanks
ignore leading blanks

-d, --dictionary-order
consider only blanks and alphanumeric characters

-f, --ignore-case
fold lower case to upper case characters

-g, --general-numeric-sort
compare according to general numerical value

-i, --ignore-nonprinting
consider only printable characters

-M, --month-sort
compare (unknown) < ‘JAN’ < ... < ‘DEC’

-n, --numeric-sort
compare according to string numerical value

-r, --reverse
reverse the result of comparisons

Other options:

-c, --check
check whether input is sorted; do not sort

-k, --key=POS1[,POS2]

```
       start a key at POS1, end it at POS2 (origin 1)

-m, --merge
       merge already sorted files; do not sort

-o, --output=FILE
       write result to FILE instead of standard output

-s, --stable
       stabilize sort by disabling last-resort comparison

-S, --buffer-size=SIZE
       use SIZE for main memory buffer

-t, --field-separator=SEP
       use SEP instead of non-blank to blank transition

-T, --temporary-directory=DIR
       use DIR for temporaries, not $TMPDIR or /tmp;  multiple  options
       specify multiple directories

-u, --unique
       with  -c, check for strict ordering; without -c, output only the
       first of an equal run

-z, --zero-terminated
       end lines with 0 byte, not newline

--help display this help and exit

--version
       output version information and exit

POS is F[.C][OPTS], where F is the field number  and  C  the  character
position  in  the  field.   OPTS  is one or more single-letter ordering
options, which override global ordering options for that  key.   If  no
key is given, use the entire line as the key.

SIZE  may be followed by the following multiplicative suffixes: % 1% of
memory, b 1, K 1024 (default), and so on for M, G, T, P, E, Z, Y.

With no FILE, or when FILE is -, read standard input.

*** WARNING *** The locale specified by the  environment  affects  sort
order.  Set LC_ALL=C to get the traditional sort order that uses native
byte values.
```

AUTHOR

Written by Mike Haertel and Paul Eggert.

REPORTING BUGS

Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT

SEE ALSO

The full documentation for sort is maintained as a Texinfo manual. If the info and sort programs are properly installed at your site, the command

```
info sort
```

should give you access to the complete manual.

NAME

join -- relational database operator

SYNOPSIS

join [-a file_number | -v file_number] [-e string] [-o list] [-t char]
[-1 field] [-2 field] file1 file2

DESCRIPTION

The join utility performs an ‘‘equality join’’ on the specified files and writes the result to the standard output. The ‘‘join field’’ is the field in each file by which the files are compared. The first field in each line is used by default. There is one line in the output for each pair of lines in file1 and file2 which have identical join fields. Each output line consists of the join field, the remaining fields from file1 and then the remaining fields from file2.

The default field separators are tab and space characters. In this case, multiple tabs and spaces count as a single field separator, and leading tabs and spaces are ignored. The default output field separator is a single space character.

Many of the options use file and field numbers. Both file numbers and field numbers are 1 based, i.e., the first file on the command line is file number 1 and the first field is field number 1. The following options are available:

-a file_number
In addition to the default output, produce a line for each unpairable line in file file_number.

-e string
Replace empty output fields with string.

-o list
The -o option specifies the fields that will be output from each file for each line with matching join fields. Each element of list has the either the form ‘file_number.field’, where file_number is a file number and field is a field number, or the form ‘0’ (zero), representing the join field. The elements of list must be either comma (‘,’) or whitespace separated. (The latter requires quoting to protect it from the shell, or, a simpler approach is to use multiple -o options.)

-t char

Use character char as a field delimiter for both input and output. Every occurrence of char in a line is significant.

-v file_number
Do not display the default output, but display a line for each unpairable line in file file_number. The options -v 1 and -v 2 may be specified at the same time.

-1 field
Join on the field'th field of file 1.

-2 field
Join on the field'th field of file 2.

When the default field delimiter characters are used, the files to be joined should be ordered in the collating sequence of sort(1), using the -b option, on the fields on which they are to be joined, otherwise join may not report all field matches. When the field delimiter characters are specified by the -t option, the collating sequence should be the same as sort(1) without the -b option.

If one of the arguments file1 or file2 is ''-'', the standard input is used.

EXIT STATUS
The join utility exits 0 on success, and >0 if an error occurs.

COMPATIBILITY
For compatibility with historic versions of join, the following options are available:

-a
In addition to the default output, produce a line for each unpairable line in both file 1 and file 2.

-j1 field
Join on the field'th field of file 1.

-j2 field
Join on the field'th field of file 2.

-j field
Join on the field'th field of both file 1 and file 2.

-o list ...
Historical implementations of join permitted multiple arguments to the -o option. These arguments were of the form

‘file_number.field_number’ as described for the current -o option. This has obvious difficulties in the presence of files named ‘1.2’.

These options are available only so historic shell scripts do not require modification. They should not be used in new code.

LEGACY DESCRIPTION

The -e option causes a specified string to be substituted into empty fields, even if they are in the middle of a line. In legacy mode, the substitution only takes place at the end of a line.

Only documented options are allowed. In legacy mode, some obsolete options are re-written into current options.

For more information about legacy mode, see compat(5).

SEE ALSO

awk(1), comm(1), paste(1), sort(1), uniq(1), compat(5)

STANDARDS

The join command conforms to IEEE Std 1003.1-2001 (‘‘POSIX.1’’).