Taurre

Zeste de savoir

Ce bon vieil ed

9 octobre 2022

Table des matières

Introduction

Si vous utilisez une distribution GNU/Linux ou un BSD, il est fort probable que vous ayez du vous frotter un jour ou l'autre à l'édition en ligne de commande. De là, vous avez certainement découvert les deux mastodontes de l'édition que sont `vi(m)` et `emacs`, ou peut-être d'autres comme `nano` ou `joe` (qui a dit `cat`? 🍐).

Toutefois, connaissez-vous `ed`, l'éditeur historique d'Unix? S'il peut sembler désuet et/ou austère aujourd'hui, notamment dû au fait qu'il n'a pas d'interface WYSIWYG, il reste un outil parfois précieux qui vaut la peine d'être découvert.

1. Un peu d'histoire

`ed` a été développé à une époque où les interactions avec un ordinateur se faisaient à l'aide d'un téléscripteur (ou téléimprimeur) et non à l'aide d'un clavier et d'un écran. Autrement dit, tant

ce qui était entré par l'utilisateur que ce qui était produit en sortie par l'ordinateur était tapé ou imprimé sur papier.

De ce fait, dans le cadre d'un éditeur de texte, d'une part l'ergonomie se devait d'être différente (impossible d'afficher constamment le contenu du fichier en cours d'édition) et, d'autre part, il était important d'être concis et peu verbeux afin de ne pas gaspiller du papier.

FIGURE 1.1. – Ken Thompson (assis devant un téléscripteur) et Dennis Ritchie travaillant sur le PDP-11

2. Première rencontre

Comme je vous l'ai précisé dans l'introduction, `ed` ne dispose pas d'une interface WYSIWYG. Ainsi, à l'inverse des éditeurs «modernes», `ed` n'affiche pas en temps réel le contenu de votre fichier, mais attend des **commandes** de votre part qui seront appliquées à des lignes de votre fichier.

Un exemple valant mieux qu'une longue explication, je vous invite à invoquer `ed` depuis votre terminal sans lui préciser aucun argument.

i

`ed` étant l'éditeur historique sous Unix, il est en général disponible de base et situé dans le répertoire `/bin`. Toutefois, certaines distributions GNU/Linux ne l'incluent pas par défaut (honte à elles! 🤘), il vous sera alors nécessaire de l'installer via votre gestionnaire de paquets.

```
1 $ ed
2 # Heu... ?
```

?

Heu…? 🍊

Oui, c'est probablement la réaction la plus fréquente (et sans doute la plus naturelle) lors de la première utilisation d'`ed`. 🍊

Comme vous le voyez, `ed` n'affiche rien, et pour cause, il attend des instructions de votre part ou, dans son langage, des **commandes**. Quelles sont elles? C'est ce que nous allons voir. 🍊

3. Premiers pas

Bien, nous venons de démarrer `ed` sans lui préciser le nom d'un fichier existant, il n'y a en conséquence pour l'instant aucune ligne de texte à manipuler puisque nous partons de zéro. Commençons donc par ajouter du texte à notre futur fichier.

La commande `a` (pour ***a**ppend*) permet d'ajouter une ou plusieurs lignes de texte. Une fois la commande entrée, `ed` considère tout ce qui suit comme du texte à ajouter jusqu'à rencontrer une ligne comprenant uniquement un point (`.`).

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
```

Les différentes commandes d'`ed` suivent à peu près toutes le même schéma: un numéro de ligne ou un intervalle de lignes (tous deux facultatifs), la commande à appliquer indiquée par *une* (et une seule) lettre et un argument (facultatif également) précédé par un espace.

Maintenant que nous avons ajouté trois lignes, voyons si elles sont bien présentes. La commande `p` (pour ***p**rint*) permet d'afficher une ou plusieurs lignes. Là où les lignes à afficher sont indiquées *avant* la commande. Par exemple, pour afficher la première ligne, nous pouvons employer la commande suivante.

```
1p
Une ligne
```

Si maintenant nous souhaitons afficher les lignes une à trois, nous pouvons recourir à un intervalle de lignes en séparant la première et la dernière ligne de ce dernier par une virgule.

```
1,3p
Une ligne
Deux lignes
Trois lignes
```

À noter que pour se simplifier la vie, il existe un numéro de ligne spécial: `$` qui correspond à la dernière ligne. Ainsi, l'intervalle `1,$p` désigne toutes les lignes.

```
1,$p
Une ligne
Deux lignes
Trois lignes
```

Un intervalle de lignes ne peut être construit que par numéros de ligne *croissants*. Autrement dit, si l'intervalle `1,3` est correct, l'intervalle `3,1` ne l'est pas.

Notre texte étant bien présent, il ne nous reste à présent plus qu'à le sauvegarder dans un fichier. La commande `w` (pour ***w**rite*) permet d'écrire une ou plusieurs lignes dans un fichier. Le nom du fichier est optionnel si `ed` a été démarré en lui précisant un nom de fichier en argument. Comme ce n'est pas notre cas, nous allons préciser le nom du fichier à créer.

```
1,$w test.txt
35
```

Comme vous le voyez, nous avons demandé à `ed` d'écrire toutes les lignes dans un fichier nommé «test.txt». Si tout se passe bien, `ed` retourne le nombre de caractères (dans les faits, *bytes*) écrits, caractères de contrôle (retour à la ligne, retour chariot, etc.) compris.

Si aucune ligne ou internvalle de lignes n'est précisé, la commande `w` écrit toutes les lignes dans le fichier indiqué. Dans le cas de l'exemple ci-dessus nous aurions donc pu nous contenter de la commande `w test.txt`.

Dans le cas où vous éditez un fichier existant, c'est-à-dire soit qui existait déjà soit que vous avez créé à l'aide de la commande `w`, il ne vous est plus nécessaire de préciser le nom du fichier à la commande `w` car `ed` conserve le nom du fichier en cours d'édition. Vous pouvez obtenir ce nom à l'aide de la commande `f` (pour «***f**ile*»).

Nous pouvons maintenant quitter l'éditeur à l'aide de la commande `q` (pour ***q**uit*).

```
q
$
```

La syntaxe présentée ci-dessus devrait être familière aux utilisateurs de `vi(m)` puisqu'il s'agit de celle employée par son coéquipier: `ex` (qui est un «successeur» d'`ed` signifiant

«***ex***tended», bien que leurs fonctionnalités ne se recoupent pas tout à fait), auquel il est possible de fournir des commandes en les précédant par `:`. Il est d'ailleurs possible de passer de `vi(m)` à `ex` et inversement à l'aide de la commande `Q` (sous `vi(m)`) et `vi` (sous `ex`).

Notez que si vous tentez de quitter l'éditeur sans avoir sauvegardé, vous verrez qu'`ed` ne vous laisse pas faire en vous fournissant un message des plus explicite: «?». 🍊

```
a
Une ligne
Deux lignes
Trois lignes
.
q
?
```

Oui, `ed` est assez peu verbeux de base, aussi existe-t-il une commande pour lui tirer les vers du nez: `h` (pour «***h****elp*»).

```
q
?
h
warning: file modified
```

Comme vous pouvez le lire, `ed` nous avertit que nous n'avons pas effectué de sauvegarde et que si nous quittons maintenant, le contenu sera définitivement perdu. Si nous souhaitons malgré tout abandonner ces modifications, il nous suffit d'entrer la commande `q` une seconde fois.

Notez qu'il est également possible d'employer la commande `H` (pour… ben «***H****elp*» aussi 🍊) afin de préciser à `ed` qu'il soit *toujours* verbeux durant la session d'édition.

```
H
q
?
warning: file modified
```

4. La ligne courante

Bien, reprenons à présent le fichier `test.txt` que nous avons créé dans la précédente section.

```
$ ed test.txt
35
```

Comme vous le voyez, `ed` nous affiche d'abord le nombre de *bytes* composant le fichier (qui correspond assez logiquement au nombre indiqué précédemment par la commande `w`), puis attend des commandes de notre part. Demandons-lui à présent d'afficher une ligne à l'aide de la commande `p`.

```
p
Trois lignes
```

De manière assez surprenante, `ed` nous affiche la dernière ligne de notre fichier: «Trois lignes». En fait, si aucune ligne ou intervalle de lignes ne lui est précisée, la commande `p` affiche la **ligne courante**.

FIGURE 4.2. – Non, pas celle-là.

La ligne courante est en fait la dernière ligne ayant été manipulée. En l'occurrence, `ed` a lu le contenu du fichier `test.txt` lors de son lancement, la dernière ligne du fichier est donc la dernière ligne à avoir été lue (et donc «manipulée»).

Maintenant, si nous affichons les deux premières lignes, puis affichons la ligne courante, nous afficherons une deuxième fois la seconde ligne s'agissant de la dernière ligne manipulée (ici affichée).

```
1,2p
Une ligne
Deux lignes
p
Deux lignes
```

Tout comme pour la dernière ligne du fichier, il existe un numéro de ligne spécial qui correspond à la ligne courante, il s'agit du point `.`. Ainsi, pour afficher le contenu du fichier de la ligne courante à la dernière, il est possible d'utiliser l'intervalle `.,$`.

```
1,2p
Une ligne
Deux lignes
p
Deux lignes
.,$p
Deux lignes
Trois lignes
```

4.0.1. Un peu d'arithmétique

Nous venons de voir qu'il existe deux symboles spéciaux: `$` pour désigner la dernière ligne et `.` pour désigner la ligne courante. Toutefois, sachez qu'il est également possible de désigner une ou plusieurs lignes par rapport à la position de la ligne courante ou de la dernière ligne en combinant ces symboles avec des additions ou soustractions. Par exemple, la suite `.+1` désigne la ligne suivant la ligne courante et la suite `$-1` désigne la ligne précédant la dernière ligne.

4.0.2. Intervalle à partir la ligne courante

En plus des intervalles construits à l'aide du symbole `,`, sachez qu'il est possible d'en construire avec le symbole `;` au lieu du symbole `,`. La différence entre les deux est que dans le cas du point-virgule, la ligne courante est modifiée pour correspondre à la première adresse de l'intervalle *avant* de calculer la seconde adresse de l'intervalle.

Ainsi,

— `2,+1p` signifie «afficher la ligne deux jusqu'à la ligne suivant la ligne courante»; et

— `2;+1p` signifie «faire de la ligne deux la ligne courante, puis affichez le contenu depuis la ligne courante jusqu'à la ligne qui la suit».

La différence peut paraître subtile, mais se comprend mieux avec un exemple.

```
$ ed
a
Première ligne
Deuxième ligne
Troisième ligne
Quatrième ligne
Cinquième ligne
.
4p
Quatrième ligne
2,+1p
Deuxième ligne
Troisième ligne
Quatrième ligne
Cinquième ligne
2;+1p
Deuxième ligne
Troisième ligne
```

Comme vous le voyez, dans le premier cas (`2,+1p`), nous avons affiché le texte depuis la ligne deux jusqu'à la ligne suivant la ligne courante (la ligne courante étant la quatrième) et dans le second (`2;+1p`), la ligne deux est devenue la ligne courante, puis celle-ci et la suivante ont été affichées.

4.0.3. Quelques raccourcis

Enfin, en plus des symboles `.` et `$`, sachez qu'il existe quelques autres raccourcis qui peuvent s'avérer utiles.

Raccourci	Équivalent à	Signification
`,`	`1,$`	De la première à la dernière ligne
`,x`	`1,x`	De la première à la ixième ligne
`;`	`.;$`	De la ligne courante à la dernière ligne
`;x`	`.;x`	De la ligne courante à la ixième ligne

5. Ajout, insertion, modification et suppression

5.0.1. Les commandes « a » et « i »

Les commandes `a` (pour «***a**ppend*») et `i` (pour «***i**nsert*») permettent, respectivement, d'ajouter du texte après ou d'insérer du texte avant une ligne ou un intervalle de lignes. Une fois l'une des commandes entrée, `ed` considère ce qui suit comme le texte qui doit être ajouté/inséré jusqu'à rencontrer une ligne ne comportant qu'un point (`.`).

```
$ ed
a
Un
Trois
.
1p
Un
1i
Zéro
.
2a
Deux
.
,p
Zéro
Un
Deux
Trois
```

L'exemple ci-dessus ajoute deux lignes après la dernière du fichier (aucune ligne n'étant renseignée à la commande `a`), affiche le contenu de la première ligne, insère une ligne *avant* la première ligne, ajoute une ligne après la seconde et affiche enfin à nouveau le contenu de la première ligne.

Notez bien ici que si vous précisez un intervalle de lignes, le texte fournit n'est ajouté qu'avant (ou après) l'intervalle et non avant (ou après) chaque ligne de celui-ci.

5.0.2. La commande « c »

La commande `c` (pour «***c**hange*») vous permet de remplacer une ou plusieurs lignes par de nouvelles, que vous entrez de la même manière que lors d'un ajout ou d'une insertion. Dans l'exemple ci-dessous, les deux lignes ajoutées sont finalement remplacées par une nouvelle.

```
$ ed
a
Un
Deux
.
1,2c
Zéro
.
1p
Zéro
```

5.0.3. La commande « d »

La commande `d` (pour *«**d**elete»*) supprime purement et simplement une ou plusieurs lignes.

```
$ ed
a
Zéro
Un
Deux
.
1d
1,2p
Un
Deux
```

i

Si aucune ligne ou intervalle de lignes ne leur est précisé:
- la commande `a` ajoute le texte renseigné après la dernière ligne du fichier;
- la commande `i` insère le texte spécifié avant la ligne courante;
- la commande `c` remplace la ligne courante;
- la commande `d` supprime la ligne courante.

6. Dénombrer une ou plusieurs lignes

La plupart des commandes d'`ed` étant basées sur des numéros de ligne, il est important de pouvoir obtenir ces numéros. La commande `n` (pour «***n**umber*») est identique à la commande `p` si ce n'est que chaque ligne est précédée de son numéro.

```
$ ed
a
Première ligne
Deuxème ligne
Troisième ligne
.
,n
1       Première ligne
2       Deuxème ligne
3       Troisième ligne
```

Si aucune ligne ou intervalle de lignes ne lui est précisée, la commande `n` dénombre et affiche la ligne courante.

La commande `=` peut également être utilisée pour connaître un numéro de ligne. Si aucune ligne n'est spécifiée, elle retourne le numéro de la dernière ligne. Si une ligne lui est précisée, alors elle retourne le numéro de cette ligne. Cette commande est surtout utile pour connaître le numéro de la ligne courante avec `.=`, ou le nombre de lignes composant un fichier.

```
$ ed
a
Première ligne
Deuxème ligne
Troisième ligne
.
1p
Première ligne
=
3
.=
1
```

7. Examiner une ou plusieurs lignes

Il arrivera que vous ayez un doute quant au contenu exact d'une ou plusieurs lignes. La commande `l` (pour «*list*»), comme les commandes `p` et `n`, affiche le contenu d'une ligne ou d'un intervalle de lignes. Toutefois, cette dernière affiche certains caractères spéciaux (comme les tabulations) et les caractères non ASCII à l'aide d'une séquence d'échappement.

Une séquence d'échappement est en fait une suite de caractères commençant par un *backslash* (\) et se terminant soit par une lettre soit par une suite de trois chiffres en base 8. Pour les premières, en voici la liste exhaustive.

Caractères	Séquence d'échappement
Caractère d'appel (*bell*)	`\a`
Espacement arrière (*backspace*)	`\b`
Saut de page (*form feed*)	`\f`
Retour chariot (*carriage return*)	`\r`
Tabulation horizontale (*horizontal tab*)	`\t`
Tabulation verticale (*vertical tab*)	`\v`
Backslash	`\\`

Pour les secondes, la valeur du nombre en base 8 correspond à la valeur de chaque *byte* ne correspondant pas à un caractère ASCII. Si vous souhaitez en apprendre plus sur les encodages, nous vous invitons à lire le cours de Maëlan sur les encodages .

Enfin, la commande `l` indique chaque fin de ligne par le caractère `$`. Si ce caractère fait partie du texte à afficher, il est précédé par un *backslash* (`\`).

Afin d'illustrer l'usage de cette commande, prenons le fichier suivant (que vous pouvez copier-coller).

```
Et pour une illustration, quoi de mieux qu'une citation élégante ?

        La perfection est atteinte, non pas lorsqu'il n'y a plus
        rien à ajouter, mais lorsqu'il n'y a plus rien à retirer.
        - Antoine de Saint-Exupéry
```

Si nous affichons ce texte à l'aide de la commande `l`, nous obtenons ceci.

```
$ ed citation.txt
,l
Et pour une illustration, quoi de mieux qu'une citation
    \303\251l\303\251gante ?$
$
\tLa perfection est atteinte, non pas lorsqu'il n'y a plus$
\trien \303\240 ajouter, mais lorsqu'il n'y a plus rien \303\240
    retirer.$
\t- Antoine de Saint-Exup\303\251ry$
```

Comme vous pouvez le voir, chaque fin de ligne est indiquée par un `$`. Vous pouvez voir également que les trois lignes comprenant la citation ainsi que son auteur commencent par une tabulation horizontale, indiquée par `\t`. Enfin, nous avons plusieurs suites de nombres en base 8

représentant en fait les caractères «à» et «é», ces derniers ne faisant pas partie de la table ASCII (pour les plus curieux d'entre vous, dans notre cas, ces caractères sont encodés en UTF-8).

Si aucune ligne ou intervalle de lignes ne lui est précisée, la commande `l` affiche la ligne courante.

8. Rechercher une ou plusieurs lignes

Pouvoir employer les numéros de lignes est une chose, il peut toutefois être plus intéressant de pouvoir rechercher une ligne spécifique. C'est ce que permettent les commandes `/texte recherché/` et `?texte recherché?`. Les deux commandes sont identiques si ce n'est que la première effectue la recherche depuis la ligne courante vers la fin du fichier alors que la seconde l'effectue depuis la ligne courante vers le début du fichier.

Dans le cas où la recherche est concluante, `ed` vous amènera à la première ligne qui contient le texte recherché et affichera cette ligne. Si aucune occurrence n'est trouvée, `ed` vous fera part de son habituelle concision en vous précisant: «?».

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
p
Trois lignes
?Deux?
Deux lignes
?Quatre?
?
h
No match
```

Comme vous pouvez le voir, après avoir trouvé une ligne contenant «Deux», `ed` nous y a amené et l'a affichée. Par contre, n'ayant pas trouvé le texte «Quatre», `ed` nous le précise en nous affichant «?».

Notez que les seconds `/` et `?` sont facultatifs. Ainsi, `?Deux` est équivalent à `?Deux?`.

Notez également qu'il est parfaitement possible de combiner une recherche avec une addition ou une soustraction. Ainsi, la suite `?Quatre?+1` désigne la ligne suivant celle qui

contient «Quatre».

8.0.1. Répéter la dernière recherche

Il est possible de répéter la dernière recherche en ne précisant pas de motifs de recherche. Ainsi, les commandes `//` (ou `/`) et `??` (ou `?`) effectuent la même recherche que précédemment.

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
?ligne
Deux lignes
?
Une ligne
```

8.0.2. Appliquer une commande lors d'une recherche

Notez que, comme pour un numéro ou intervalle de lignes, un motif de recherche peut être utilisé pour spécifier une ou plusieurs lignes auxquelles une commande doit être appliquée.

```
$ ed
a
Une ligne
Trois lignes
.
?Une?a
Deux lignes
.
1,$p
Une ligne
Deux lignes
Trois lignes
```

Comme vous le voyez, nous avons utilisé une recherche pour préciser après quelle ligne ajouter notre texte.

Dans le cas où vous utilisez une recherche au lieu d'un numéro ou intervalle de lignes, la forme longue est requise. Autrement dit, vous devez terminer votre recherche avec le second `/` ou `?`.

De la même manière, un intervalle de lignes peut être précisé à l'aide de deux recherches.

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
?Une?,?Deux?p
Une ligne
Deux lignes
```

Dans l'exemple ci-dessus, nous demandons à `ed` d'afficher l'intervalle de lignes qui va de la première ligne contenant «Une» à la première ligne contenant «Deux».

Rappelez-vous: un intervalle de lignes ne peut être construit que par numéros de ligne croissants.

9. Substitution

Nous l'avons vu, la commande `c` permet de remplacer une ou plusieurs lignes par de nouvelles. Cela peut s'avérer pratique en cas d'erreurs à corriger ou de modifications à opérer, toutefois cela reste assez lourd puisqu'il est par exemple nécessaire de réécrire une ligne entière simplement pour corriger une faute d'orthographe ou une faute de frappe…

Heureusement, il existe une alternative: la commande `s` (pour «*substitute*»). Cette dernière permet d'effectuer une *substitution*, c'est-à-dire de remplacer une portion de texte par une autre. La syntaxe de base est la suivante: `s/texte à remplacer/texte de remplacement/`.

```
$ ed
a
Un ligne
Deux lignes
.
1s/Un/Une/
1p
Une ligne
```

Dans l'exemple ci-dessus, le mot «Un» de la première ligne est remplacé par «Une».

Dans les faits, n'importe quel caractère autre qu'un retour à la ligne ou un espace peut être utilisé pour séparer le texte à remplacer et le texte de remplacement. Ainsi, les commandes `1s/Un/Une/`, `1s#Un#Une#` et `1s|Un|Une|` sont par exemple identiques.

9.0.1. Substitution sur plusieurs lignes

Comme la plupart des autres commandes, la commande `s` peut être appliquée à une ligne ou à un intervalle de lignes.

```
$ ed
a
Un ligne
Un autre ligne
.
1,2s/Un/Une/
1,2p
Une ligne
Une autre ligne
```

Comme vous le voyez, le mot «Un» a été substitué par «Une» sur la première et la deuxième ligne.

9.0.2. Afficher la ligne courante après substitution

Il est possible d'afficher la ligne courante après avoir effectué là où les substitutions en terminant la commande par `p`, `n` ou `l`. Celle-ci sera alors affichée de la même manière que si vous utilisiez cette commande. Notez bien que cela affiche la ligne courante, donc si vous avez modifié plusieurs lignes, vous ne verrez le résultat que pour la dernière ligne modifiée.

```
$ ed
a
Un ligne
Un autre ligne
.
1,2s/Un/Une/p
Une autre ligne
```

9.0.3. Répéter la dernière substitution

La commande `s` peut également être utilisée sans lui préciser de texte à remplacer. Dans un tel cas, la dernière substitution effectuée est répétée.

```
$ ed
a
Un ligne
Un autre ligne
.
1,2s/Un/Une/p
Une autre ligne
```

9.0.4. Substitutions multiples

Par défaut, la commande `s` substitue uniquement la première occurrence trouvée.

```
$ ed
a
Un deux trois
.
s/ /, /
p
Un, deux trois
```

Toutefois, il est possible de modifier ce comportement en ajoutant un ou plusieurs suffixes en fin de commande. Le suffixe `g` (pour «*global*») permet de préciser que *toutes* les occurrences rencontrées doivent être remplacées.

```
$ ed
a
Un deux trois
.
s/ /, /g
p
Un, deux, trois
```

Les suffixes `p` et `g` peuvent être combinés.

9.0.5. Substitution sélective

Si seule la énième occurrence rencontrée doit être remplacée, un nombre peut être utilisé comme suffixe. Ainsi, dans l'exemple ci-dessous, seule la seconde occurrence est remplacée.

```
$ ed
a
Un deux trois
.
s/ /, /2
p
Un deux, trois
```

i Il est possible de combiner un suffixe numéral et le suffixe `p`.

9.0.6. Substitution sur base de la dernière recherche

Dans le cas où le texte à remplacer n'est pas précisé, `ed` considère qu'il s'agit de celui de la dernière recherche.

```
$ ed
a
Un ligne
Deux lignes
Trois lignes
.
?Un?
Un ligne
s//Une/p
Une ligne
```

Cette opération peut également être combinée comme suit.

```
$ ed
a
Un ligne
Deux lignes
Trois lignes
.
?Un?s//Une/p
Une ligne
```

9.0.7. Recherche sur base de la dernière substitution

Inversement, il est possible d'effectuer une recherche sur base de la dernière substitution.

```
$ ed
a
Un ligne
Un autre ligne
.
s/Un/Une/p
Une autre ligne
??s//Une/p
Une ligne
```

9.0.8. Substitution sur base du dernier texte de remplacement

Dans le cas où le texte de remplacement est `%`, `ed` considère qu'il s'agit du dernier texte de remplacement utilisé. Ainsi, l'exemple précédent peut-être simplifié comme suit.

```
$ ed
a
Un ligne
Un autre ligne
.
s/Un/Une/p
Une autre ligne
??s//%/p
Une ligne
```

Dans ce cas-ci, `%` renvoie à «Une», qui est le dernier texte de remplacement utilisé.

Dans le cas où vous voulez remplacer du texte par `%`, précéder simplement ce dernier par un *backslash*, comme ceci: `\%`.

9.0.9. Utilisation du texte à remplacer dans le texte de remplacement

Dans le cas où le texte de remplacement est proche du texte à remplacer, il est pénible de devoir taper quasiment deux fois la même chose. Pour éviter cela, il est possible d'inclure le symbole `&` au sein du texte de remplacement. `ed` remplacera toutes les occurrences de ce symbole par le texte à remplacer.

```
$ ed
a
Un ligne
.
s/Un/&e/p
Une ligne
```

Comme vous pouvez le voir, le symbole `&` été remplacé par «Un» qui est bien le texte à remplacer.

Comme pour le symbole `%`, si vous devez utiliser le symbole `&` littéralement, précédez-le d'un *backslash*.

9.0.10. Utilisation de portions du texte à remplacer dans le texte de remplacement

Au lieu de faire référence au texte à remplacer dans son entièreté, il est possible de définir des portions de celui-ci. Ces portions peuvent alors être utilisées dans le texte de remplacement. Les portions sont délimitées dans le texte à remplacer par `\(` et `\)`. Notez bien l'emploi du *backslash* qui permet de spécifier à `ed` que ces parenthèses ne font pas partie du texte à remplacer.

Chaque portion se voit attribuer un numéro par `ed` selon leur ordre d'apparition. Il est alors possible de les utiliser dans le texte de remplacement à l'aide de ce numéro précédé par un *backslash*, par exemple `\1` pour la première portion définie.

```
$ ed
a
ligne Une
.
,s/\(ligne\) \(Une\)/\2 \1/p
Une ligne
```

Comme vous le voyez, nous avons défini deux portions: «ligne» et «Une», puis nous les avons utilisées dans le texte de remplacement pour inverser les deux mots.

10. Rechercher ou substituer par motif

Jusqu'à présent, nous n'avons fait que rechercher ou substituer des textes précis. Toutefois, si `ed` se limitait à cela, l'utilité de ces deux fonctions serait assez mince. Il est possible de les employer de manière plus large en utilisant des *motifs*.

Basiquement, un motif est un texte qui décrit de manière abstraite un ensemble possible d'autres textes. Par exemple, imaginez que vous voulez inverser les deux premiers mots de chaque ligne,

ceci est impossible sans employer des motifs, car les deux premiers mots de chaque ligne ne sont pas identiques. Il nous faut un moyen pour expliquer à `ed` ce qu'est un mot.

Pour construire un motif, `ed` utilise trois types de construction, toutes facultatives: des ensembles, des répétitions et des ancres.

10.0.1. Les ensembles

Un ensemble permet de préciser que l'on recherche non pas un caractère précis, mais un caractère *parmi* un ensemble de caractères. Cela est possible à l'aide de la construction `[...]` où «…» est un ensemble de caractères possibles. Par exemple, si l'on souhaite une voyelle, n'importe laquelle, il est possible d'écrire `[aeiouy]`.

Toutefois, il est heureusement possible de raccourcir l'écriture de certains ensembles. En effet, pour rechercher n'importe quelle lettre, il faudrait écrire `[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]` ce qui est… un peu long (et accessoirement insuffisant puisque cela ne regroupe pas les lettres avec accents par exemple). 🍊

Ainsi, les constructions `[A-Z]`, `[a-z]` et `[0-9]` désignent, respectivement: n'importe quelle lettre majuscule, n'importe quelle lettre minuscule et n'importe quel chiffre. Ces ensembles peuvent être réduits en changeant la lettre ou le chiffre de début et/ou de fin. Notez qu'il est possible de les combiner, de sorte que `[A-Za-z]` désigne n'importe quelle lettre.

Dans l'exemple ci-dessous, nous supprimons toutes les voyelles de la première ligne.

```
$ ed
a
Une ligne
.
s/[AEIOUYaeiouy]//gp
n lgn
```

Dans le cas où l'on souhaite désigner un caractère, n'importe lequel, le symbole `.` peut être utilisé. Les crochets ne sont alors pas nécessaires.

Dans l'exemple qui suit, nous utilisons le `.` pour supprimer le premier caractère de chaque ligne.

```
$ ed
a
Première ligne
Deuxième Ligne
Troisième ligne
.
,s/.//
,p
remière ligne
```

```
euxième Ligne
roisième ligne
```

Si vous souhaitez utiliser les caractères `[`, `]` ou `.` littéralement, il est nécessaire de les précéder d'un *backslash*.

Sauf lors de la définition d'un ensemble. Dans un tel cas, tous les caractères ont leur sens littéral, *backslash* compris. Si vous souhaitez utiliser le symbole `]` au sein d'un ensemble, celui-ci doit être le premier caractère. Ainsi,

- `[][]` signifie «`[` ou `]`»;
- `[]\[]` signifie «`[`, ou `\` ou `]`»; et
- `[\[\]]` est incorrecte.

10.0.1.1. Inverser un ensemble

Dans le cas où un ensemble commence par le symbole `^`, son sens est inversé. Ainsi, `[^aeiuoy]` signifie n'importe quel caractère *sauf* une voyelle.

Pour utiliser le symbole `^` lui-même au sein d'un ensemble, il suffit de ne pas le placer comme premier caractère de celui-ci. Par exemple, `[][^]` signifie «`[`, ou `]` ou `^`».

10.0.2. Les répétitions

Une répétition s'applique à un ensemble (`[...]` ou `.`) et se place immédiatement à la suite de l'ensemble auquel elle s'applique. Une répétition précise le nombre de caractères recherchés parmi un ensemble. Elle peut soit prendre la forme `\{x\}` où `x` défini un nombre exact de répétition, soit la forme `\{x,y\}` où `x` et `y` définissent un intervalle de répétitions. Toutefois, dans le cas d'un intervalle, le second nombre peut-être omit, auquel cas l'intervalle signifie «au moins x».

Ainsi, si l'on veut rechercher exactement deux voyelles, il est possible d'écrire `[aeiouy]\{2\}`. Alors que si l'on souhaite rechercher au moins deux voyelles, nous pouvons utiliser `[aeiouy]\{2,\}`.

Enfin, le caractère spécial `*` peut-être utilisé en lieu et place d'une répétition classique et signifie «zéro ou plus». Ainsi, `[a-z]*` signifie «zéro lettre ou plus».

10.0.3. Les ancres

Une ancre sert à préciser où la recherche du motif doit être opérée. Il en existe deux: `^` et `$`. L'ancre `^` précise que le motif doit être recherché *uniquement* au début de la ligne. À l'inverse, le symbole `$` précise que le motif doit être recherché *uniquement* en fin de ligne. L'ancre `^` se place au début d'un motif tandis que l'ancre `$` se place à la fin d'un motif.

Ainsi, le motif `^[A-Z]` signifie «une majuscule en début de ligne» et le motif `[a-z]$` signifie «une minuscule en fin de ligne».

Ces deux ancres peuvent également être utilisées seules, soit pour construire un motif auquel toutes les lignes répondent (nous y reviendront un peu plus tard lorsque nous verrons les commandes globales), soit pour insérer du texte en début ou en fin de ligne à l'aide d'une substitution.

```
$ ed
a

.
s/^/Première/
s/$/ ligne/
p
Première ligne
```

Dans l'exemple ci-dessus, nous avons ajouté une ligne vide, puis nous avons utilisé deux substitutions: une pour insérer «Première» au début de la ligne et une pour insérer «<espace>ligne» en fin de ligne.

Notez enfin qu'il est possible d'utiliser les deux ancres en même temps, auquel cas le motif s'applique uniquement à la ligne complète. Ainsi, `^[A-Za-z]\{1,\}$` signifie «une ligne ne comportant qu'un seul mot».

Avec ça, il nous est possible de résoudre le problème posé au début de la section, à savoir inverser le premier et le second mot de chaque ligne. Vous pouvez essayer par vous-même dans un premier temps, la correction est juste ci-dessous. 🍊

Contenu masqué n°1

Revoyons ce gros morceau ensemble. 🍊
Tout d'abord, nous cherchons à inverser les deux *premiers* mots de chaque ligne, nous allons donc utiliser l'ancre `^`. Ensuite, nous recherchons un mot. Un mot étant une suite d'*au moins une lettre*, nous pouvons donc utiliser l'ensemble `[A-Za-z]` (une lettre) et la répétition `\{1,\}` (au moins un). Enfin, nous divisons le texte à remplacer en portions à l'aide de `\(\)` afin d'inverser les deux mots trouvés. 🍊

11. Annuler la dernière commande

La commande `u` (pour «***u**ndo*») permet d'annuler la dernière commande *ayant effectué des modifications* (la commande `p` n'est par exemple pas concernée). Cette commande vous sera d'une aide précieuse en cas d'erreur, que ce soit lors d'un ajout, d'une substitution, d'une suppression, etc.

Notez bien que la commande `u` ne peut annuler que la *dernière* commande ayant effectué des modifications. Il n'est pas possible d'annuler plusieurs commandes.

```
$ ed
a

.
s/^/Première/
s/$/ ligne/
p
Première ligne
```

Dans l'exemple ci-dessus, nous supprimons la dernière ligne par erreur. Nous utilisons alors la commande `u` pour annuler celle-ci.

12. Marquer une ou plusieurs ligne

Il n'est pas toujours évident de retenir et/ou de retaper des numéros de lignes. Aussi, `ed` vous permet de marquer des lignes afin de faciliter leur désignation par après. Pour ce faire, rendez-vous sur la ligne que vous souhaitez marquer et utilisez la commande `k` (pour «*mar**k***») suivie d'une lettre minuscule. Cette ligne pourra ensuite être référencée à l'aide de la suite `'x` où «x» correspond à la lettre minuscule que vous avez choisie.

Notez bien que la commande `k` permet de marquer *une* ligne à l'aide d'une lettre minuscule et non un intervalle de lignes.

```
$ ed
a
Première ligne
Deuxième ligne
Troisième ligne
Quatrième ligne
.
?Deux?
Deuxième ligne
ka
/Trois/
Troisième ligne
kb
'a,'bp
Deuxième ligne
```

```
16 Troisième ligne
```

Comme vous le voyez, la ligne «Deuxième ligne» a été marquée comme «a» et la ligne «Troisième ligne» a été marquée comme «b». Après quoi nous avons affiché l'intervalle allant de la ligne marquée comme «a» jusqu'à la ligne marquée comme «b».

Une fois de plus, gardez à l'esprit qu'un intervalle de lignes ne peut être construit que par numéros de ligne *croissants*.

Si aucune ligne ne lui est précisée, la commande `k` marque la ligne courante.

13. Copier ou déplacer une ou plusieurs lignes

Les commandes `m` (pour «***m**ove*) et `t` (pour ~~«t'occupe, y avait plus de lettres disponibles»~~ «***t**ransfer*») permettent respectivement de déplacer une ou plusieurs lignes et de copier une ou plusieurs lignes.

Dans les deux cas, ces commandes insèrent là où les lignes à déplacer ou copier *après* une ligne donnée. Comme pour n'importe quelle autre commande, il peut s'agir d'un numéro de ligne précis, ou d'un motif de recherche.

```
1  $ ed
2  a
3  Première ligne
4  Troisième ligne
5  Deuxième ligne
6  .
7  $m1
8  ,p
9  Première ligne
10 Deuxième ligne
11 Troisième ligne
```

Dans l'exemple ci-dessus, nous avons déplacé la dernière ligne («Deuxième ligne») *après* la première ligne.

Après exécution d'une de ces commandes, la ligne courante est la dernière ligne déplacée ou copiée. Dans notre cas, il s'agit donc de la deuxième ligne.

Si aucune ligne ou intervalle de lignes ne leur est précisé, les commandes `m` et `t` copient ou déplacent la ligne courante.

14. Diviser ou fusionner une ou plusieurs lignes

14.0.1. Fusion

La commande `j` (pour «*join*») permet de fusionner deux ou plusieurs lignes.

```
$ ed
a
Un
Deux
Trois
.
1,3j
p
UnDeuxTrois
```

Comme vous le voyez, les trois lignes ont été fusionnées.
Notez que la commande n'ajoute aucun espace. Si cela est désiré, c'est à vous de l'ajouter auparavant, par exemple avec une substitution.

```
$ ed
a
Un
Deux
Trois
.
2,3s/^/ /
1,3j
p
Un Deux Trois
```

Si aucune ligne ou intervalle de lignes ne lui est précisé, la commande `j` fusionne la ligne courante et la ligne qui la suit.

14.0.2. Division

À l'inverse, il est possible de diviser une ligne en une ou plusieurs lignes à l'aide d'une substitution. Pour ce faire, il est nécessaire de remplacer un ou plusieurs caractères par un retour à la ligne. Cela est possible en précédant ce retour à la ligne d'un *backslash*.

```
$ ed
a
Un Deux Trois
.
s/ /\
/g
,p
Un
Deux
Trois
```

15. Charger ou insérer le contenu d'un fichier

Les commandes `e` (pour «***e**dit*») et `r` (pour «***r**ead*») permettent, respectivement, de charger ou d'insérer le contenu d'un fichier.

15.0.1. Charger le contenu d'un fichier

La commande `e` charge le contenu d'un fichier en lieu et place du contenu actuel. Elle modifie également le nom du fichier en cours d'édition.

```
$ ed
a
Première ligne
Deuxième ligne
Troisième ligne
.
w exemple.txt
49
f
exemple.txt
e test.txt
35
,p
Une ligne
Deux lignes
Trois lignes
f
```

```
18 test.txt
```

Comme vous le voyez, nous avons d'abord créé un fichier nommé «exemple.txt». `ed` a alors sauvegardé ce nom comme étant celui du fichier en cours d'édition, comme en témoigne la sortie de la commande `f`. Nous avons ensuite utilisé la commande `e` pour charger le contenu du fichier «test.txt». Cette dernière nous affiche alors le nombre de caractères lus (tout comme le fait la commande `w`, mais pour le nombre de caractère écrits). Comme le montre la sortie de la commande `,p`, seul le contenu du fichier «test.txt» est désormais présent. Également, `ed` a adapté le nom du fichier en cours d'édition.

15.0.2. Insérer le contenu d'un fichier

La commande `r` est identique à la commande `e` si ce n'est qu'elle insère le contenu d'un fichier ou la sortie d'une commande externe *après* la ligne indiquée.

> *i* Si aucune ligne ou intervalle de lignes ne lui est précisée, la commande `r` insère le contenu du fichier après la dernière ligne.

```
1  $ ed exemple.txt
2  49
3  ,p
4  Première ligne
5  Deuxième ligne
6  Troisième ligne
7  r test.txt
8  35
9  ,p
10 Première ligne
11 Deuxième ligne
12 Troisième ligne
13 Une ligne
14 Deux lignes
15 Trois lignes
```

Comme vous le voyez, nous avons inséré le contenu du fichier «test.txt» après la dernière ligne.

16. Exécuter une commande externe

La commande `!` permet d'exécuter une commande externe, comme si vous l'exécutiez via votre terminal. `ed` affichera la sortie de cette commande à la suite et indiquera sa fin par une ligne ne comportant qu'un point d'exclamation.

```
$ ed
!ls
test.txt
!
```

Comme vous le voyez, nous avons entré `!ls` afin d'exécuter la commande `ls`. La sortie de cette dernière est affichée à la suite et est terminée par une ligne ne comportant que `!`.

16.0.1. Répéter la dernière commande externe

Durant son exécution, `ed` conserve en mémoire la dernière commande externe exécutée, de sorte qu'il soit possible de l'utiliser à nouveau plus facilement. Si la commande externe à exécuter est `!` , alors `ed` exécutera celle utilisée précédemment.

```
$ ed
!ls
test.txt
!
```

Dans l'exemple ci-dessus, nous faisons appel à la dernière commande externe exécutée, à savoir `ls`. Juste après avoir entré `!!` , `ed` nous précise quelle commande est exécutée, ici `ls`, comme attendu.

16.0.2. Utiliser le nom du fichier courant

Si vous avez besoin du nom du fichier que vous éditez actuellement pour une commande externe, vous pouvez utiliser le caractère `%`. `ed` le remplacera par le nom du fichier en cours d'édition avant d'exécuter la commande externe. Ceci peut être utile si vous souhaitez par exemple lire votre fichier via `less` ou `more` avant de continuer l'édition.

```
$ ed mon_fichier.txt
!less %
!
```

16.0.3. Charger ou insérer la sortie d'une commande externe

Il est possible de charger ou d'insérer la sortie d'une commande afin de l'éditer directement dans `ed`. Pour ce faire, il est nécessaire de combiner les commandes `e` ou `r` avec la commande `!`.

```
$ ed mon_fichier.txt
!less %
!
```

Dans l'exemple ci-dessus, la sortie de la commande `ls` est chargée à l'aide de la commande `e !ls`. `ed` nous indique alors le nombre de caractères lus. Comme le montre le résultat de la commande `,p`, nous avons bien obtenu la sortie de la commande `ls`.

16.0.4. Fournir des lignes en entrée d'une commande externe

À l'inverse, il est possible de fournir des lignes en entrée d'une commande externe en combinant la commande `w` et la commande `!`.

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
w !sort
Deux lignes
Trois lignes
Une ligne
35
,p
Une ligne
Deux lignes
Trois lignes
```

Comme vous le voyez, nous avons envoyé les trois lignes en entrée de la commande `sort`. `ed` nous a alors fourni le résultat de cette dernière ainsi que le nombre de caractères le composant. Notez que le fichier, lui, n'est en rien modifié, ce qui est logique puisque nous avons simplement fait appel à une commande externe.

17. Les commandes globales

La commande `g` (pour «***g***lobal») permet d'exécuter une suite de commandes sur un ensemble de lignes respectant un motif. Cette commande est de loin la plus puissante d'`ed` car elle permet d'effectuer aisément des actions complexes.

Un exemple simple de son fonctionnement est par exemple d'afficher toutes les lignes comprenant un texte donné.

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
g/lignes/p
Deux lignes
Trois lignes
```

Comme vous le voyez, la commande `g` se compose d'un motif (comme la commande de recherche ou de substitution) suivie d'une ou plusieurs commandes à exécuter (ici une seule: `p`).

Si aucune ligne ou intervalle de lignes n'est spécifié, la commande `g` s'applique à toutes les lignes.

Pour préciser plusieurs commandes, il est nécessaire de les séparer par un retour à la ligne. Ces retours à la ligne doivent être précédés d'un *backslash*, comme dans le cas de la division d'une ligne.

```
$ ed
a
Une ligne
Deux lignes
Trois lignes
.
g/^/s/ /\
/\
s/li/Li/
,p
Une
Ligne
Deux
Lignes
Trois
Lignes
```

Dans l'exemple ci-dessus, nous effectuons deux substitutions: une pour diviser chaque ligne et une pour mettre en majuscule le début de chaque mot «Ligne(s)». Notez que nous avons utilisé deux *backslashs*: un pour le retour à la ligne utilisé par la division et un pour le retour à la ligne séparant les deux substitutions de la commande `g`.

Dans le cas des commandes `a`, `i` ou `c` leur comportement change lorsqu'elles sont utilisées via la commande `g`. Chaque ligne, *sauf* la dernière, doit être terminée par un *backslash* et la ligne finale ne comportant qu'un point n'est plus nécessaire.

```
$ ed
a
Un
Quatre
.
g/Un/a\
Deux\
Trois
,p
Un
Deux
Trois
Quatre
```

Sachez qu'il existe également la commande `v` (pour «*invert*») qui est identique à la commande `g` si ce n'est qu'elle applique la suite de commandes aux lignes ne respectant *pas* le motif.

17.0.1. Version interactive

Les commandes `g` et `v` disposent chacune d'un pendant interactif `G` et `V`. Ces deux commandes sont identiques aux commandes `g` et `v` si ce n'est qu'elles ne prennent qu'un motif comme argument car les commandes à appliquer sont demandées interactivement pour chaque ligne respectant le motif.

```
$ ed
a
Une ligne
Deux linges
Trois linges
.
G/linges/
Deux linges
s/linges/lignes/
Trois linges
&
```

Dans l'exemple ci-dessus, nous appliquons la commande `G` à chaque ligne contenant «linges». Pour chaque ligne respectant le motif de recherche, son contenu est affiché, puis `ed` attends la liste de commandes à lui appliquer. Notez que la syntaxe de cette liste de commandes (et notamment l'usage des *backslash*) est la même que pour `g` et `v`. Dans notre cas, nous avons appliqué une substitution.

Afin d'éviter de retaper une commande identique, il est possible d'entrer `&` pour appliquer la dernière suite de commandes utilisée. C'est ce que nous avons fait pour la seconde ligne.

Il est possible d'interrompre les commandes `G` et `V` en cours de route à l'aide de la suite `Ctrl-c`.

17.0.2. Exercices

La commande `g` est parfaite pour revoir un peu tout ce que nous avons vu au travers de quelques exercices. 🍊

17.0.2.1. Inverser les lignes d'un fichier Le premier exercice consiste à inverser les lignes d'un fichier. Étant donné le fichier ci-dessous, votre but est d'obtenir, via *une* commande `g`, le même fichier mais avec les lignes en ordre inverse.

```
Première ligne
Deuxième ligne
Troisième ligne
Quatrième ligne
Cinquième ligne
```

 Contenu masqué n°2

17.0.3. Supprimer les lignes vides

Pour le second exercice, essayer de supprimer toutes les lignes vides du fichier suivant à l'aide d'*une seule* commande `g`.

```
Première ligne

Deuxième ligne

Troisième ligne

```

```
Quatrième ligne

Cinquième ligne
```

Contenu masqué n°3

17.0.4. Fusionner les lignes vides

Comme troisième exercice, essayez de fusionner les lignes vides du fichier suivant. Autrement dit, il ne doit plus y avoir de lignes vides qui se suivent.

```
Première ligne


Deuxième ligne



Troisième ligne


Quatrième ligne

Cinquième ligne
```

Contenu masqué n°4

18. L'invite de commande

Vous aurez sans doute remarqué que dans un soucis de lisibilité, nous avons surligné dans les exemples précédents les lignes comprenant des commandes. En effet, par défaut, `ed` n'emploie pas d'invite de commande permettant de différencier une ligne de commande d'une ligne de texte. Pire, il vous est impossible de savoir si `ed` attend de vous une commande ou bien du texte (par exemple lors d'un ajout) sans faire appel à votre mémoire, ce qui peut vite conduire à des erreurs (par exemple à l'insertion de commande dans un texte à ajouter…).

```
$ ed
a
```

```
Un
Deux
w <-- Hé m**d* !
.
d
1,2p
Un
Deux
```

Heureusement, il existe la commande `P` (pour «***P**rompt*») qui précise à `ed` d'ajouter un invite de commande chaque fois qu'il attend une instruction de notre part. Par défaut, cet invite de commande est le caractère `*`.

```
$ ed
P
*a
Un
Deux
.
*1,2p
Un
Deux
```

Ce caractère peut-être remplacé par n'importe quelle suite de caractères en la précisant via l'option `-p` lors de l'invocation d'`ed`. Notez que le fait d'employer l'option `-p` active du coup l'utilisation de l'invite de commande.

```
$ ed -p "(ed) "
(ed) a
Un
Deux
.
(ed)
```

C'est tout de même un peu plus clair, non? 🍊

Conclusion

19. Liens et ressources utiles

— ed , The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004;
— (PDF) *A Tutorial Introduction to the UNIX Text Editor* (via), Brian W. Kernighan, dans *UNIX programmer's manual*, volume 2A, 7ème édition, 1979, pp. 53–64;

— (PDF) *Advanced Editing on UNIX* (via), Brian W. Kernighan, dans *UNIX programmer's manual*, volume 2A, 7ème édition, 1979, pp. 65–80;
— (PDF) The UNIX programming environment, Brian W. Kernighan, Rob Pike, 1984, pp. 319–328; et bien entendu
— ed(1).

Contenu masqué

Contenu masqué n°1

```
$ ed -p "(ed) "
(ed) a
Un
Deux
.
(ed)
```

Retourner au texte.

Contenu masqué n°2

Une solution simple est de recourir à la commande `m`, pour déplacer successivement chaque ligne au début du fichier. Étant donné que nous commençons par la première ligne, celle-ci reste à sa place, ensuite la seconde passe au-dessus de la première et ainsi de suite jusqu'à la dernière qui devient alors la première.

```
$ ed exercice.txt
83
g/^/.m0
,p
Cinquième ligne
Quatrième ligne
Troisième ligne
Deuxième ligne
Première ligne
```

i Notez que l'usage du `.` avant la commande `m` est facultatif puisque cette dernière s'applique par défaut à la ligne courante.

Retourner au texte.

Contenu masqué n°3

La solution la plus simple consiste à utiliser le motif `^$`, représentant précisément une ligne vide et ensuite d'appliquer la commande `d` à toutes les lignes respectant ce motif.

```
$ ed exercice.txt
87
g/^$/d
,p
Cinquième ligne
Quatrième ligne
Troisième ligne
Deuxième ligne
Première ligne
```

Retourner au texte.

Contenu masqué n°4

La première chose pour réaliser cet exercice est de rechercher toutes les lignes vides. Ainsi, nous allons à nouveau utiliser le motif `^$`. Ensuite, il nous faut fusionner les lignes depuis la ligne vide que nous avons trouvé jusqu'à une ligne vide précédant une ligne non vide. Ceci peut être traduit par un intervalle commençant à la ligne courante et terminant une ligne avant une ligne non vide, soit `/./-1` (le point correspondant à n'importe quel caractère, il nous permet de désigner n'importe quelle ligne non vide).

```
$ ed exercice.txt
91
g/^$/.,/./-1j
,p
Première ligne

Deuxième ligne

Troisième ligne

Quatrième ligne

Cinquième ligne
```

Retourner au texte.

Liste des abréviations

WYSIWYG What You See Is What You Get. 1, 2