Chapitre 10
grep, la commande de recherche de lignes

1. Description

grep est une commande permettant de rechercher des chaînes de caractères dans des fichiers texte. Elle traite indifféremment des fichiers dont le nom lui est passé en argument, ou des données qu'elle lit sur l'entrée standard (*stdin*).

Cette commande a été écrite très rapidement par Ken Thompson à partir de fragments de code source de *ed*. Son nom provient de la commande de recherche globale d'expressions régulières de *ed* :

```
g/re/p
```

qui signifie comme nous l'avons vu précédemment dans le chapitre consacré à *ed* : rechercher toutes les lignes du fichier (**g**) contenant une chaîne correspondant à l'expression régulière *re* (**re** : *Regular Expression*), puis les afficher (**p**).

grep a tout d'abord été une commande privée de Ken Thompson, avant qu'il ne se décide à la placer dans un répertoire d'utilitaires du système (*/bin*) avant la sortie d'Unix v4 (version 4 d'Unix interne aux laboratoires *Bell*).

Elle fait maintenant partie des commandes indispensables des utilisateurs d'Unix (et dérivés d'Unix) par qui elle est souvent utilisée comme outil de filtrage des résultats d'autres commandes.

Il en existe deux autres variantes : *egrep* et *fgrep*. *egrep* est capable de traiter des *expressions régulières étendues* (*Extended grep*, ou *Extended regular expression grep*), alors que *fgrep* ne traite que des chaînes fixes (*Fixed strings grep*).

2. Principe de fonctionnement

2.1 Généralités

La commande *grep* (ou sa variante *egrep*) recherche dans des fichiers ou dans l'entrée standard des lignes contenant des chaînes de caractères spécifiées par des chaînes fixes ou des expressions régulières.

L'expression à rechercher est passée en argument à la commande, ainsi qu'une éventuelle liste de fichiers à examiner.

Les expressions reconnues par *grep* sont des *expressions régulières basiques*, tandis que celles reconnues par *egrep* sont des *expressions régulières étendues*.

Chaque ligne dont une sous-chaîne correspond à l'expression régulière sera affichée (sauf demande contraire) sur la sortie standard.

Les nombreuses options disponibles permettent de modifier ce comportement, et de demander de n'afficher aucun résultat, dans le cas où l'on souhaite uniquement utiliser le code de retour pour savoir si les données contiennent ou non une chaîne correspondant à l'expression régulière spécifiée. Il est également possible, dans le cas de la commande provenant de la *FSF* (*Free Software Foundation*), d'afficher le contexte de la ligne correspondant à l'expression régulière, en affichant des lignes qui précèdent et qui suivent les lignes trouvées.

2.2 Rappel de quelques options

Parmi les options fréquemment utilisées de *grep*, nous citerons les suivantes :

– **-f** qui permet de spécifier le nom d'un fichier dans lequel *grep* doit aller lire les expressions régulières,

– **-h** qui demande de ne pas afficher les noms des fichiers en tête de lignes, dans le cas où plusieurs fichiers sont examinés,

– **-i** qui permet d'ignorer la casse des lettres (*-y* dans la commande d'origine et les anciennes versions de *grep*),

– **-l** qui demande d'afficher uniquement les noms des fichiers dans lequel il y a au moins une chaîne correspondant à l'expression régulière spécifiée,

– **-L** (absente de la commande d'origine) qui demande l'affichage des noms de fichiers dont aucune ligne ne contient une chaîne correspond à l'expression régulière spécifiée,

– **-q** (*-s* dans la commande d'origine) qui demande de n'afficher aucune ligne trouvée à l'écran (utile quand on veut uniquement connaître le code retour sans voir les lignes correspondantes),

– **-s** qui demande de ne pas afficher les messages d'erreurs (sans équivalent dans la commande d'origine),

– **-v** qui permet d'inverser la sélection, et donc de n'afficher que les lignes ne contenant pas de chaîne correspondant à l'expression régulière spécifiée.

3. Les expressions basiques

3.1 Généralités

Les *expressions régulières basiques* sont celles qui sont reconnues par la commande *grep*. Consultez le chapitre Synthèse des types d'expression en fin d'ouvrage pour un résumé des caractéristiques.

3.2 Utilisation du caractère ^

Le caractère **^** en début d'expression spécifie un début de ligne. À l'aide de ce caractère, il est possible de rechercher une chaîne en début de ligne. Les chaînes recherchées peuvent être exprimées sous forme de chaînes de caractères ou d'expressions régulières.

Remarque

*Si l'on recherche une chaîne commençant par le caractère **^**, il faudra enlever au caractère **^** sa signification générique en le faisant précéder d'un antislash.*

3.3 Utilisation du caractère $

Le caractère **$** en fin d'expression spécifie une fin de ligne. À l'aide de ce caractère, il est possible de rechercher une chaîne en fin de ligne. Les chaînes recherchées peuvent être exprimées sous forme de chaînes de caractères fixes ou d'expressions régulières.

Remarque

*Si l'on recherche une chaîne se terminant par le caractère **$**, il faudra enlever au caractère **$** sa signification générique en le faisant précéder d'un antislash.*

3.4 Utilisation du caractère .

Le caractère **.** dans une expression sert à désigner n'importe quel caractère.

Remarque

*Si l'on recherche spécifiquement le caractère **.** dans une ligne, il faudra lui enlever son caractère générique en le faisant précéder d'un antislash.*

3.5 Utilisation du caractère *

Le caractère * dans une expression est un facteur de répétition qui s'applique au caractère qui le précède. Il indique que le caractère peut figurer *N* fois dans la chaîne, *N* étant positif ou nul.

Le caractère précédant le caractère * peut être spécifié de plusieurs façons. Cela peut être :

– un caractère standard,

– le caractère **.** (qui désigne n'importe quel caractère),

– un antislash suivi de n'importe quel caractère, chiffres et parenthèses exceptés,

– un caractère spécifié à l'aide d'une énumération utilisant les caractères **[** et **]**.

Contrairement à sa signification pour le shell, le caractère * ne désigne pas une chaîne de caractères quelconques, mais uniquement un facteur de répétition.

L'équivalent (approximatif) du caractère * du shell serait ici **.***, pour désigner une répétition d'un caractère quelconque, représenté par **.**.

3.6 Utilisation des caractères [ ]

Les caractères **[** et **]** servent à désigner un caractère quelconque parmi un ensemble spécifié sous forme d'énumération ou sous forme de plage contiguë, ou encore sous toute combinaison de ces deux possibilités.

Dans les versions récentes de *grep*, le caractère **^** placé immédiatement après le crochet ouvrant sert à exprimer la négation, et donc à désigner l'ensemble complémentaire de celui spécifié.

Remarque

À l'intérieur des crochets, les caractères spéciaux perdent leur signification.

3.7 Utilisation des caractères \< et \>

La chaîne **\<** permet de spécifier un début de mot, tandis que la chaîne **\>** permet de spécifier une fin de mot.

Une chaîne est considérée être en début de mot si elle est en début de ligne, ou non précédée d'un caractère faisant partie d'un mot, c'est-à-dire d'un caractère alphanumérique ou du caractère souligné.

Une chaîne est considérée être en fin de mot si elle est en fin de ligne, ou non suivie d'un caractère faisant partie d'un mot, c'est-à-dire d'un caractère alphanumérique ou du caractère souligné.

3.8 Utilisation du caractère \

Le caractère antislash sert à supprimer sa signification particulière à un méta-caractère, ou à définir un caractère particulier, tel que ceux qui sont définis par le langage C : *\t* (tabulation), *\n* (saut de ligne, ou *newline*), *\r* (retour en début de ligne, ou *carriage return*)...

3.9 Classes de caractères

Certaines versions de *grep* et *egrep* savent gérer les classes de caractères qui permettent de désigner des ensembles de façon générique, et indépendante de la langue utilisée. Ces classes sont les suivantes :

- **[:alnum:]**, pour les caractères alphanumériques,
- **[:alpha:]**, pour les caractères alphabétiques,
- **[:cntrl:]**, pour les caractères de contrôle,
- **[:digit:]**, pour les chiffres décimaux,
- **[:graph:]**, pour les caractères graphiques,
- **[:lower:]**, pour les lettres minuscules,
- **[:print:]**, pour les caractères affichables,
- **[:punct:]**, pour les caractères de ponctuation,

– **[:space:]**, pour les espaces (au sens large),
– **[:upper:]**, pour les lettres majuscules,
– **[:xdigit:]**, pour les caractères hexadécimaux.

La variable **LC_ALL** qui gère la localisation doit être initialisée de façon cohérente avec la langue utilisée dans les données manipulées.

Remarque

*Les crochets ouvrant et fermant font partie intégrante du nom de la classe, et doivent être présents dans une énumération utilisant déjà des crochets. Ainsi un caractère hexadécimal s'écrira **[[:xdigit:]]**.*

4. Les expressions étendues

4.1 Généralités

Les *expressions régulières étendues* sont celles qui sont reconnues par *egrep*, ou par *grep* lorsqu'on l'utilise avec l'option **-E** (ou **--extended-regexp**).

Elles permettent, entre autres, de grouper des expressions grâce aux parenthèses, de déclarer des expressions optionnelles, et d'utiliser des *OU* logiques.

Dans les sections qui suivent, seules les spécificités des *expressions régulières étendues* par rapport aux *expressions régulières basiques* seront présentées.

Les métacaractères des *expressions régulières basiques* font partie des *expressions régulières étendues*.

Consultez le chapitre Synthèse des types d'expression en fin d'ouvrage pour un résumé des caractéristiques.

4.2 Utilisation des caractères ( )

Les parenthèses servent à grouper des expressions.

4.3 Utilisation du caractère |

Le caractère **|** est équivalent à un *OU* logique (booléen). Il sert à exprimer qu'une chaîne peut correspondre à une expression ou à une autre.

4.4 Utilisation du caractère ?

Le caractère **?** dans une expression sert à indiquer que le caractère ou l'expression qui le précède est optionnel, c'est-à-dire qu'il peut être présent ou absent.

4.5 Utilisation du caractère *

Le caractère * a la même signification que dans les *expressions régulières basiques*, mais il peut s'appliquer à une expression entière, et pas seulement au caractère qui le précède comme c'est le cas dans les *expressions régulières basiques*. Pour qu'il s'applique à une expression entière, il suffit d'encadrer cette expression de parenthèses et de faire suivre la parenthèse fermante du signe *.

Syntaxe

```
c*
```

ou

```
(expr)*
```

4.6 Utilisation du caractère +

Le caractère **+** dans une expression est un facteur de répétition qui s'applique au caractère ou à l'expression qui le précède. Il indique que le caractère ou l'expression peut figurer *N* fois dans la chaîne, *N* étant strictement positif.

4.7 Utilisation des caractères { }

Les accolades permettent de spécifier les nombres minimum et maximum d'occurrences d'une expression. Elles doivent encadrer deux nombres séparés par une virgule, le premier désignant le nombre minimum d'occurrences de l'expression précédant les accolades, et le second désignant le maximum d'occurrences.

Si le nombre avant la virgule est absent, cela signifie qu'il n'y a pas de nombre minimum d'occurrences.

De même, si le nombre après la virgule est absent, cela signifie qu'il n'y a pas de nombre maximum d'occurrences.

Les facteurs de répétitions s'appliquent à l'élément qui les précède, un caractère ou une expression régulière.

Syntaxe

```
c{min,max}
(expr){min,max}
```

Remarque

Seules certaines versions de egrep permettent l'usage d'accolades. Cette syntaxe n'est pas uniformément implémentée dans l'ensemble des commandes egrep, et de ce fait n'est pas portable. Son utilisation peut poser des problèmes sur un système où la commande egrep ne reconnaît pas cette syntaxe.

5. Exercices

5.1 Exercice 1

Comment rechercher dans le fichier */etc/passwd* tous les utilisateurs dont le nom contient un chiffre ?

Solution

```
< /etc/passwd egrep '^[^:]*[0-9][^:]*'
```

Cette commande recherche sur son entrée standard, qui a été redirigée par le shell vers le fichier */etc/passwd*, toutes les lignes dont le début (**^**) commence par une chaîne constituée d'une suite éventuellement vide de caractères différents du caractère : (**[^:]***), suivie d'un chiffre (**[0-9]**) suivi d'une chaîne constituée d'une suite éventuellement vide de caractères différents du caractère : (**[^:]***).

5.2 Exercice 2

Extraire de la commande */bin/ls -l* toutes les lignes correspondant à un fichier régulier accessible en écriture pour tout le monde.

Donner deux solutions différentes, l'une avec *grep* et l'autre avec *egrep*.

Solution 1

Avec *grep*, on peut écrire la commande suivante :

```
/bin/ls -l | grep '^-.......w'
```

qui va afficher toutes les lignes commençant par un tiret (**-**) désignant un fichier régulier, suivi d'une suite de 7 caractères quelconques (**.....**), suivie du caractère **w**.

Solution 2

Avec *egrep*, on peut écrire la commande suivante :

```
/bin/ls -l | egrep '^-.{7}w'
```

qui va afficher toutes les lignes commençant (**^**) par un tiret (**-**), suivi d'une suite de 7 caractères quelconques (**.{7}**), suivie du caractère **w**.

5.3 Exercice 3

Comment afficher les lignes du fichier *stop.data* les lignes contenant les mots :

– **stop**,

– **stopping**,

– **stopped**,

indifféremment en minuscules ou majuscules, en colorisant le mot correspondant ?

On utilisera le fichier de données *stop.data* suivant :

```
Ligne 1 : abcd stop 123
Ligne 2 : The server is stopping ...
Ligne 3 : Service 1 is starting.
Ligne 4 : The server has stopped.
Ligne 5 : Service 2 is starting.
Ligne 6 : STOP !!!
Ligne 7 : STOPPPPP !!!!!!
```

On proposera plusieurs solutions, dont au moins une avec *grep* et une avec *egrep*.

Solution 1

```
grep --color -i '\<stop[a-z]*' stop.data
```

Le résultat sera :

```
Ligne 1 : abcd stop 123
Ligne 2 : The server is stopping ...
Ligne 4 : The server has stopped.
Ligne 6 : STOP !!!
Ligne 7 : STOPPPPP !!!!!!
```

Cette commande recherche en fait un mot commençant (**\<**) par la chaîne *stop*, indifféremment en minuscules ou majuscules (**-i**). La recherche est donc plus large que la demande de l'énoncé, et la ligne 7 n'est pas souhaitée, car elle ne correspond pas à la recherche demandée : le mot **STOPPPPP** ne fait pas partie des mots énumérés.

Solution 2

Pour n'afficher avec grep que les lignes contenant les mots spécifiés, on utilisera un fichier (*stop.regex2*) contenant la liste des expressions souhaitées :

```
\<stop\>
\<stopping\>
\<stopped\>
```

La commande de recherche devient :

```
grep --color -i -f stop.regex2 stop.data
```

et le résultat est alors :

```
Ligne 1 : abcd stop 123
Ligne 2 : The server is stopping ...
Ligne 4 : The server has stopped.
Ligne 6 : STOP !!!
```

qui est bien le résultat recherché.

Solution 3

Pour éviter de créer un fichier contenant la liste des mots recherchés, on utilisera la commande *egrep* avec une *expression régulière étendue* :

```
egrep --color -i '\<(stop|stopping|stopped)\>' stop.data
```

Le résultat de cette commande est également le résultat recherché :

```
Ligne 1 : abcd stop 123
Ligne 2 : The server is stopping ...
Ligne 4 : The server has stopped.
Ligne 6 : STOP !!!
```

Il faut cependant faire attention à ne pas oublier les parenthèses, sinon le sens de l'expression serait différent. La commande suivante :

```
egrep --color -i '\<stop|stopping|stopped\>' stop.data
```

affiche effectivement un résultat non souhaité :

```
Ligne 1 : abcd stop 123
Ligne 2 : The server is stopping ...
Ligne 4 : The server has stopped.
Ligne 6 : STOP !!!
Ligne 7 : STOPPPPP !!!!!!
```

La signification de la première expression est :

– une chaîne contenant le mot **stop**, ou le mot **stopping**, ou le mot **stopped**.

La signification de la deuxième expression est :

– une chaîne contenant une chaîne commençant par **stop**, ou contenant la chaîne **stopping**, ou une chaîne se terminant par **stopped**.

La dernière commande est donc équivalente à la commande suivante :

```
egrep --color -i '(\<stop)|(stopping)|(stopped\>)' stop.data
```

qui n'a pas la fonction demandée.

5.4 Exercice 4

Comment lister les processus du serveur HTTP, qui a pour nom *apache2* sur Debian, sans afficher le processus *grep* qui va servir à sélectionner les processus *apache2* ?

Solution 1

La commande suivante :

```
$ ps -ef | grep apache 2
```

qui lance deux processus, affiche la liste complète des processus sauf les threads (**ps -ef**), puis filtre en sélectionnant les lignes contenant apache2 (**grep apache2**), et affiche le résultat suivant :

```
root           3386        1  0 Jan28 ?  00:00:06 /usr/sbin/apache2 -k start
www-data 18993  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18994  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18995  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18996  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18997  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
root          24543 20442  0 15:32 pts/10 00:00:00 grep --color apache 2
```

où apparaît le processus *grep* de sélection des lignes contenant apache2.

Remarque

*Ici, la commande apparaît sous la forme **grep --color apache2** car le shell utilisé a lu une directive **alias grep='grep --color'**.*

Ce dernier processus ne nous intéresse pas, et la tentation est grande d'ajouter un autre filtre *grep* pour enlever cette ligne du résultat affiché :

```
ps -ef | grep apache2 | grep -v grep
```

Cette commande, qui lance trois processus, affiche la liste complète des processus sauf les threads (**ps -ef**), puis filtre en sélectionnant les lignes contenant *apache2* (**grep apache2**), puis filtre en sélectionnant tout sauf les lignes contenant *grep* (**grep -v grep**).

Le résultat est le suivant :

```
root           3386        1  0 Jan28 ?  00:00:06 /usr/sbin/apache2 -k start
www-data 18993  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18994  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18995  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18996  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
www-data 18997  3386  0 07:38 ?           00:00:00 /usr/sbin/apache2 -k start
```

Cette commande fonctionne parfaitement, mais présente l'inconvénient de créer trois processus et deux tubes de communication (*pipes*). En utilisant les expressions régulières, on peut réduire la commande à deux créations de processus, et un seul tube de communication, de la façon suivante :

Solution 2

```
ps -ef | grep '[a]pache2'
```

Le résultat, qui est bien celui recherché, sera le suivant :

```
root           3386        1  0 Jan28 ?  00:00:06 /usr/sbin/apache2 -k start
www-data 18993  3386  0 07:38 ?          00:00:00 /usr/sbin/apache2 -k start
www-data 18994  3386  0 07:38 ?          00:00:00 /usr/sbin/apache2 -k start
www-data 18995  3386  0 07:38 ?          00:00:00 /usr/sbin/apache2 -k start
www-data 18996  3386  0 07:38 ?          00:00:00 /usr/sbin/apache2 -k start
www-data 18997  3386  0 07:38 ?          00:00:00 /usr/sbin/apache2 -k start
```

Cela fonctionne bien, mais comment ?

La réponse est la suivante : l'expression régulière *[a]* ne correspond qu'au seul caractère *a*, donc *[a]pache2* est strictement équivalente pour *grep* à *apache2*. Mais dans la liste des processus, cette commande *grep* apparaîtra sous la forme :

```
root     29097 28888  0 16:05 pts/10   00:00:00 grep --color [a]pache2
```

et dans cette ligne, la chaîne recherchée (**apache2**) ne figure pas : les caractères **[** et **]** sont ici des caractères de données, et non des métacaractères. Cette chaîne ne sera donc pas sélectionnée par le processus *grep* auquel il correspond, qui recherche spécifiquement la chaîne **apache2**.

On aurait pu utiliser n'importe quelle autre expression régulière sélectionnant uniquement le mot *apache*, mais comportant des différences avec la chaîne recherchée, et le résultat aurait été identique. Pour des raisons d'efficacité, on choisit généralement l'expression la plus simple possible.

Les commandes suivantes, par exemple, donnent le même résultat :

```
ps -ef | grep 'a[p]ache2'
ps -ef | grep 'ap[a]che2'
ps -ef | grep 'a[p][a]che2'
ps -ef | grep 'apache[2]'
ps -ef | grep 'a\pache2'
ps -ef | egrep 'apac{1}he2'
```

5.5 Exercice 5

Afficher les lignes du fichier */etc/hosts* dont les adresses IP font partie du réseau *10.20.0.0/16*, soient les adresses IP commençant par *10.20*.

On prendra comme fichier de test le fichier *hosts.data* suivant :

```
192.168.10.20   serveur01
192.168.10.201  serveur02
172.16.110.20   serveur03
10.10.20.30     serveur04
10.20.1.12      serveur05
10.200.30.40    serveur06
95.10.20.5      serveur07
95.10.202.7     serveur08
192.168.1.181   z10320xn
```

Solution

```
grep '^10\.20\.' hosts.data
```

Cette commande donnera le résultat correct suivant :

```
10.20.1.12      serveur04
```

Si au lieu de la commande précédente, on utilise une commande aussi simple que la suivante :

```
grep '10.20' hosts.data
```

le résultat sera :

```
192.168.10.20   serveur01
192.168.10.201  serveur02
172.16.110.20   serveur03
10.10.20.30     serveur04
10.20.1.12      serveur05
10.200.30.40    serveur06
95.10.20.5      serveur07
95.10.202.7     serveur08
192.168.1.181   z10320xn
```

ce qui ne correspond pas à la demande.

Toutes ces lignes contiennent la chaîne **10**, suivie d'un caractère quelconque (**.**), suivi de la chaîne **20**, mais la sélection n'est pas suffisamment précise. Les deux nombres ne doivent pas être séparés par un caractère quelconque, mais par un point (**.**). Il faut donc placer un antislash devant le caractère **.**, ce qui permet de supprimer du résultat la ligne :

```
192.168.1.181   z10320xn
```

Pour la commande :

```
grep '10\.20' hosts.data
```

le résultat sera :

```
192.168.10.20   serveur01
192.168.10.201  serveur02
172.16.110.20   serveur03
10.10.20.30     serveur04
10.20.1.12      serveur05
10.200.30.40    serveur06
95.10.20.5      serveur07
95.10.202.7     serveur08
```

On voit que les adresses contenant **10.201**, **10.200** et **10.202** sont sélectionnées. Il faut donc spécifier que nous ne souhaitons pas **10.20** suivie d'un chiffre, mais **10.20** suivie d'un point. La commande :

```
grep '10\.20\.' hosts.data
```

donne le résultat suivant :

```
10.10.20.30     serveur04
10.20.1.12      serveur05
95.10.20.5      serveur07
```

C'est mieux, mais c'est toujours incorrect par rapport à notre besoin.

Il faut donc spécifier que la chaîne **10.20.** doit se trouver en début de ligne. La commande :

```
grep '^10\.20\.' hosts.data
```

affiche alors le résultat suivant :

```
10.20.1.12      serveur05
```

qui correspond bien à une adresse IP du réseau 10.20.0.0/16.

La conclusion de cet exercice est qu'il faut toujours correctement spécifier une expression régulière avant de l'utiliser, que ce soit interactivement, dans un script ou dans un programme, si l'on ne veut pas avoir un résultat qui ne nous apporte pas uniquement les données que l'on souhaite récupérer.

D'autre part, on n'oubliera pas les quotes autour de l'expression régulière. En effet, la commande :

```
grep ^10\.20\. hosts.data
```

afficherait le résultat suivant :

```
10.20.1.12      serveur05
10.200.30.40    serveur06
```

car le shell interprète les arguments qu'on lui passe, et convertit la chaîne

```
^10\.20\.
```

en :

```
^10.20.
```

or on souhaite que la commande *grep* reçoive un antislash (\) devant chaque point afin de lui supprimer sa signification particulière de caractère quelconque.

5.6 Exercice 6

Comment extraire d'une liste les nombres strictement supérieurs à 255 ?

On validera le résultat avec la boucle de génération suivante (sous *bash*) :

```
for i in {1..1010}; do echo $i; done
```

Solution

Les nombres supérieurs à 255 peuvent être décomposés en plusieurs plages, qui sont les nombres :

– de 256 à 259 : **25[6-9]**,
– de 260 à 299 : **2[6-9][0-9]**,
– de 300 à 999 : **[3-9][0-9]{2}**,
– au-delà de 1000 : **[1-9][0-9]{3,}**.

En réunissant ces expressions en une seule, cela donne :

```
egrep '\<25[6-9]|2[6-9][0-9]|[3-9][0-9]{2}|[1-9][0-9]{3,}\>'
```

On notera l'usage de *egrep* à cause de la nécessité d'utiliser une expression *régulière étendue* pour exprimer différentes possibilités séparées par des *OU* logiques (**|**).

La même recherche avec *grep* nécessiterait de créer un fichier contenant la liste des expressions recherchées.