Perl - Partie 6
Compléments sur l'extraction et la transformation de données

Résumé. Ce qui suit n'est pas du tout exhaustif. Pour plus d'informations et des exemples, reportez vous à la documentation Perl.

Trouver, extraire, remplacer une sous-chaîne

- `$pos = index($chaine, $souschaine) :`
  `index` retourne la position du premier caractère de `$souschaine` dans `$chaine`, retourne -1 si pas trouvé. Possibilité de rajouter un 3ème paramètre indiquant l'indice dans `$chaine` à partir duquel on cherche `$souschaine`.
- `$ch = substr($chaine, $debut, $longueur) :`
  `substr` extrait la sous-chaîne de taille `$longueur` à partir de la position `$debut` dans `$chaine`.
- `$ch = substr($chaine, $debut) :`
  `substr` extrait toute la fin de `$chaine` à partir de la position `$debut`.
- `substr($chaine, $debut, $longueur) = $nouvsouschaine :`
  `substr` remplace la sous-chaîne de taille `$longueur` à partir de la position `$debut` dans `$chaine` par `$nouvsouschaine`.

Avec des expressions régulières

- `/$exprreg/` : opérateur d'extraction.
- `s/$exprreg/$chaine/` : opérateur de substitution.
- `tr/$caracteres/$caracteres/` : opérateur de transcription.

Ces trois opérateurs s'appliquent par défaut à la variable `$_`. Si on veut les appliquer à d'autres variables que `$_`, il faut utiliser l'opérateur `=~`.
Quelques exemples supplémentaires :

- `tr/ab/ba/` : change tous les a en b et tous les b en a.
- `tr/a-z/ABCD/` : les lettres a, b, c et d deviennent respectivement A, B, C et D, et les lettres de e à z deviennent toutes D.
- `tr/a-z/ABCD/d` : les lettres a, b, c et d deviennent respectivement A, B, C et D, et les lettres de e à z sont supprimées.
- `$count = s/...//` : compte le nombre de substitutions sans modifier `$_`.
- `$count = tr/...//` : compte le nombre de transcriptions sans modifier `$_`.

Exercice Mystère. Le script Perl `mystere.pl` a besoin d'au moins un fichier sur la ligne de commande. De quel type est ce fichier d'après vous ? Que calcule ce script ?

Exercice 1. Ecrivez un court script Perl qui prend en entrée une séquence cDNA, valide qu'elle contient bien uniquement les nucléotides A, T, G et C, et affiche à l'écran la séquence mRNA correspondante (remplacez tous les T par des U).

Remarque : proposez deux solutions différentes pour les substitutions de T en U, une avec l'opérateur `s///` et une avec l'opérateur `tr///`.

Exemple de séquences valides.

```
TTTTAATTAAACGTAAAAAGGCAGG
tcaccttcgacgacgcttagagcagatagacgat
```

Exemple d'une séquence non valide.

```
GTXCTTXXAAGGCNNNTACTTTYTCCRAGCC
```

Exercice 2. Recherche de motif exact
Réalisez un script Perl lisant un fichier contenant une séquence au format FASTA et comptant les occurences d'un **motif exact** dans cette séquence.
Exemple : si on recherche le motif `"AT"` dans la séquence `ATGTAGCATAA`, on obtient :

```
AT: 2
```

Indices.
- Reportez vous à l'Exercice 2 de la Partie 5 pour la lecture dans un fichier d'une séquence au format FASTA (reprenez les fichiers `p53_seq.txt` et `IOfasta.pl`).
- Le motif et le nom du fichier FASTA seront spécifiés à l'intérieur du script.
- Utilisez la fonction `substr`.

Exercice 3. Motif inexact : attention aux opérateurs `=~` et `//` ...

1. Réalisez un script Perl lisant un fichier contenant une séquence au format FASTA et comptant les occurences d'un **motif inexact** (ou **expression régulière**) dans cette séquence.
   Exemple : si on recherche l'expression régulière `"A."` dans la séquence `ATGTAAGCATAA`, on obtient :
   ```
   AT: 2
   AG: 1
   AA: 2
   ```
   Indices.
   - Lecture du fichier : idem que l'exercice précédent.
   - Utilisez les opérateurs `=~` et `//` avec l'option `g` (pour global).
2. L'opérateur `//g` a pour inconvénient de ne pas tenir compte des occurences chevauchantes d'un motif (par exemple, le motif `AGA` intervient deux fois dans `AGAGA` mais n'est vu qu'une fois). A l'aide d'une boucle `while` avançant sur la séquence en fonction de la présence de l'expression régulière, modifiez le script précédent pour compter TOUTES les occurences d'une expression régulière.
   Remarque : cette deuxième méthode est beaucoup plus lente à l'exécution que la première, mais les deux méthodes ne donnent pas le même résultat ! Donc, tout dépend du contexte ... Vous pouvez comparer les temps d'exécution des deux méthodes à l'aide de la commande unix `time` :
   ```
   > time ./exo3.1.pl
   > time ./exo3.2.pl
   ```