Cryptographie : substitutions monoalphabétiques

4 Compléments du chapitre

Voici trois compléments au chapitre *Cryptographie : subtitutions monoalphabétiques* de l'ouvrage : Divertissements mathématiques et informatiques (Dmi dans la suite).

Le premier complément permet de présenter les cryptogrammes à l'écran sous leur forme traditionnelle et lisible (groupes de 5 lettres). Quelques notions techniques sur la gestion des objets en Ruby y sont exposées.

Le second complément est plus mathématique et propose de dénombrer les substitutions monoalphabétiques qui modifient « largement » le texte à chiffrer (peu de lettres restent identiques). Ce complément propose aussi un exercice de programmation de méthodes de dénombrements.

Enfin, le dernier complément présente une manière alternative (à celle des Dmi) de programmer le chiffre de César, en utilisant la représentation Ascii des caractères.

1 Compléments sur l'affichage des cryptogrammes

Les cryptogrammes sont généralement présentés avec des groupes de cinq lettres. Nous allons programmer un outil qui fait ce travail. Plutôt qu'une méthode qui affiche directement un cryptogramme (ou n'importe quelle chaîne de caractères), nous allons plutôt écrire une méthode qui renvoie une chaîne formatée, ce qui sera plus général. Nous pourrons ainsi afficher la chaîne, mais aussi éventuellement l'enregistrer dans un fichier, ou encore l'utiliser à autre chose.

Pour cela, nous allons devoir créer une copie de la chaîne d'origine, dans laquelle nous ajouterons les espaces et les retours à la ligne.

1.1 Copie d'un objet

Manipuler les objets – Voici une petite difficulté qu'il est important de bien comprendre avant d'aller plus loin dans la découverte des méthodes de manipulation des chaînes de caractères.

Lorsque nous utilisons une variable `s` qui désigne une chaîne, l'objet que nous manipulons est la chaîne de caractères, et non la variable. En particulier, un même objet peut être désigné par plusieurs variables :

```
irb> s1="Il porte un joli nom : Jupiter"
irb> s2=s1
irb> puts s2
Il porte un joli nom : Jupiter
irb> s1[23..29]="Saturne"
irb> puts s2
Il porte un joli nom : Saturne
```

Dans cet exemple, il y a un seul objet, c'est la chaîne « Il porte un joli nom : Jupiter » qui est modifiée en « Il porte un joli nom : Saturne ». Cet objet est désigné par deux variables, `s1` et `s2`.

Ruby : identifiant d'un objet

En Ruby, les objets ont des identifiants. Il est possible de connaître l'identifiant d'un objet à partir d'une variable qui le désigne. Nous pouvons donc savoir si deux variables désignent le même objet en comparant les identifiants obtenus (les objets désignés sont les mêmes si les identifiants sont égaux). Essayez, à la suite de la session `irb` précédente :

```
irb> s1.object_id
=> -610295338
irb> s2.object_id
=> -610295338
```

Les identifiants sont les mêmes [1], donc il n'y a qu'un seul objet pour deux variables. Modifier `s1` ou `s2` revient au même.

Dans l'exemple qui précède, la ligne `s2=s1` n'a pas pour effet de copier l'objet `s1`. Elle a simplement permis de donner un nom supplémentaire au même objet.

Pour réellement *copier* un objet, et en obtenir deux, il faut utiliser la méthode `clone` :

```
irb> s1="Il porte un joli nom : Jupiter"
irb> s2=s1.clone
irb> puts s2
Il porte un joli nom : Jupiter
irb> s1[23..29]="Saturne"
irb> puts s2
Il porte un joli nom : Jupiter
irb> puts s1
Il porte un joli nom : Saturne
```

Dans cet exemple, nous avons bien deux objets (vérifiez les valeurs de `s1.object_id` et `s2.object_id`)) et seul `s1` a été modifié. Ce comportement n'est pas spécifique aux chaînes, c'est la règle générale. Ce sont les objets « simples » comme les nombres qui sont l'exception, et c'est pour cette raison que nous n'avions pas été confrontés à cette difficulté auparavant.

1.2 Insertion des espaces et des retours à la ligne

Une fois la chaîne d'origine copiée, il nous reste à insérer des espaces tous les cinq caractères et des retours chariot tous les `k` groupes de caractères (`k` sera un paramètre de notre méthode). Pour l'insertion des caractères, nous utiliserons la méthode `insert` dont le fonctionnement est indiqué par l'exemple qui suit :

```
irb> str="Ce cnard n'a bec"
irb> str.insert(4,"a")
=> "Ce canard n'a bec"
irb> str.insert(14,"qu'un ")
=> "Ce canard n'a qu'un bec"
irb> puts str
Ce canard n'a qu'un bec
```

Notons qu'il est possible d'insérer plusieurs caractères et que l'indice donné en paramètre est l'indice qu'*occupera* la première lettre insérée dans la chaîne finale.

Attention : la méthode `insert` modifie l'objet

Curieusement, la méthode `insert` n'a pas de point d'exclamation dans son nom, alors qu'elle modifie pourtant l'objet appelant. Ce point sera peut être modifié dans les versions ultérieures de Ruby.

Rappelons l'existence du caractère spécial `\n` qui correspond à un retour à la ligne. Voici une version de notre méthode de formatage qu'il convient d'insérer dans la classe `String` [2].

`cryptographie.rb`

```
class String
  ...

  def formate(k)
     n=5
     i=n
     count=1
```

1. Vous aurez sans doute des nombres différents de ceux donnés ici, mais vous devez néanmoins obtenir deux fois le même nombre.
2. Si vous suivez aussi le contenu des Dwi, vous avez déjà un fichier `cryptographie.rb`. Sinon, vous pouvez entrer ce qui suit dans un fichier vide.

```
    fo=self.clone
    while(i<=fo.length)
      if (count==k)
        fo.insert(i,"\n");
        count=0
      else
        fo.insert(i," ");
      end
      i=i+n+1
      count=count+1
    end
    return fo
  end
end
```

Bien qu'un peu longue, la méthode n'est pas difficile à comprendre : `i` est le compteur des positions dans la chaîne (il avance de 6 en 6 dans l'exemple). La variable `count` compte les paquets de caractères. Lorsque nous avons `k` paquets, nous insérons un passage à la ligne.

Vérifions que la méthode fonctionne :

```
irb> load "cryptographie.rb"
irb> str="ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMN"
irb> puts str.formate(5)

ABCDE FGHIJ KLMNO PQRST UVWXY
ZABCD EFGHI JKLMN
```

Pour réaliser le test qui suit, vous devez aussi avoir programmé les méthodes `nettoie`, `substitution` et `cesar`. Elles sont données dans les Dwi, mais vous les trouverez aussi en téléchargement sur le site http://www.div-math.fr.

```
irb> load "cryptographie.rb"
irb> str="Ce canard n'a qu'un bec, et n'eut jamais envie, ou de n'en plus avoir ou bien ↩
    d'en avoir deux."
irb> str=str.nettoie
=> "CECANARDNAQUUNBECETNEUTJAMAISENVIEOUDENENPLUSAVOIROUBIENDENAVOIRDEUX"
irb> str2=str.substitution(cesar(3))

irb> puts str2.formate(5)
FHFDQ DUGQD TXXQE HFHWQ HXWMD
PDLVH QYLHR XGHQH QSOXV DYRLU
RXELH QGHQD YRLUG HXA
```

Remarque : renvoyer n'est pas afficher

Notez l'utilisation de la méthode formate : `str.formate(5)` *renvoie* une chaîne formatée, mais ne l'affiche pas. Nous provoquons l'affichage en écrivant :

```
puts str.formate(5)
```

.

2 Dénombrement des substitutions

Dans les Dwi, nous avons généré des substitutions aléatoires[3]. Mais nous n'avons pas évoqué la possibilité que ces substitutions soient « de mauvaise qualité » et changent, par exemple, chaque lettre en elle-même. Auquel cas, le message chiffré sera identique au message en clair.

Dans cette section, qui fait largement appel aux dénombrements, nous allons évaluer la probabilité pour qu'une substitution aléatoire « brouille bien » un texte, c'est-à-dire en change presque toutes les lettres.

3. En informatique, il n'y a jamais de vrai hasard, et nous devrions donc dire pseudo-aléatoire.

Nous verrons comment programmer les formules de dénombrement en Ruby dans la section 2.1, et le lecteur intéressé par la justification de ces formules trouvera des compléments mathématiques dans la section 2.2.

2.1 Dénombrer avec l'ordinateur

À une substitution monoalphabétique particulière correspond une permutation de l'alphabet. Une substitution est donc équivalente à la liste des 26 lettres de l'alphabet, dans un certain ordre.

Combien de tels alphabets avons-nous ? Le nombre de façons différentes d'ordonner un ensemble de n objets est $n\,!$, qui se lit *factorielle n* et vaut le produit de tous les nombres de 1 à n. Pour arranger les trois lettres `ABC`, nous aurons donc $3\,! = 6$ possibilités qui sont : `ABC`, `ACB`, `BAC`, `BCA`, `CAB` et `CBA`. Ce résultat est assez facile à concevoir : nous pouvons mettre la première lettre (le `A`) en première, deuxième ou troisième position (nous avons trois choix). Une fois que ce choix est fait, nous n'avons plus que deux choix pour le `B`, et enfin un seul choix pour le `C` : c'est-à-dire $3 \times 2 \times 1 = 6$ choix.

Pour tout l'alphabet, nous aurons donc : $26\,!$ choix. Ce nombre est considérable. Pour nous en faire une idée, demandons à Ruby de nous le calculer, nous avons vu qu'il s'accomodait justement très bien des grands nombres entiers :

```
irb> s=1
irb> for i in 2..26 : s=s*i end
irb> s
=> 403 291 461 126 605 635 584 000 000
```

Le résultat est approximativement 403 millions de milliards de milliards ($4,03 \times 10^{26}$).

Parmi toutes ces permutations de l'alphabet, certaines donneraient cependant un message trop similaire à l'original. En effet, certaines lettres de la substitution monoalphabétique associée resteraient à leur place. En particulier, parmi les 403 millions de milliards de milliards de permutations possibles, il y en a exactement une qui correspond à la substitution : remplacer `A` par `A`, `B` par `B`, jusqu'à `Z` par `Z`. Dans ce cas, le message reste inchangé.

La justification des formules employées ici est donnée dans la section 2.2.

Voyons combien de permutations « brouillent bien » le message.

Pour calculer ceci, nous devons utiliser un résultat qui énonce que le nombre de façons différentes de choisir un ensemble de p objets parmi un ensemble de n objets (nombre de combinaisons de p objets parmi n) est :

$$C_n^p = \frac{n!}{p!(n-p)!}$$

Par exemple, pour sélectionner 2 objets parmi 4, nous avons :

$$C_4^2 = \frac{4!}{2!(4-2)!} = \frac{4 \times 3 \times 2 \times 1}{(2 \times 1)(2 \times 1)} = 6 \text{ choix}$$

Nous pouvons en effet prendre les objets : 1,2 ou 1,3 ou 1,4 ou 2,3 ou 2,4 ou 3,4.

Nous avons aussi besoin d'un autre résultat, qui comptabilise les *dérangements* ou *permutations complètes*. Un dérangement est une permutation des éléments d'un ensemble qui ne laisse aucun élément à sa place. Le nombre de dérangements d'un ensemble de n éléments vaut :

$$d_n = \sum_{0 \le k \le n} (-1)^k \frac{n!}{k!} = n! \left(\frac{1}{0!} + \frac{-1}{1!} + \frac{1}{2!} + \cdots + \frac{(-1)^n}{n!} \right)$$

Le nombre de dérangements nous permet de calculer, par exemple, le nombre de façons de mélanger les trois lettres `ABC` en n'en laissant aucune à sa place :

$$d_3 = 3! \left(\frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} \right) = 3! \left(1 - 1 + \frac{1}{2} - \frac{1}{6} \right) = 3! \times \frac{2}{3} = 2$$

Effectivement, il n'y a que deux solutions, qui sont : `BCA` et `CAB`.

Il existe une autre manière (dite par récurrence) d'exprimer ces mêmes nombres :

$$\begin{cases} d_0 = 1 \\ d_n = n d_{n-1} + (-1)^n \end{cases}$$

L'expression qui précède est facile à programmer en Ruby. Créez un nouveau fichier, appelé `denombrement.rb`, et entrez la méthode suivante [4] :

denombrement.rb

```
def derangements(n)
  return 1 if n<=0
  return n*d(n-1)+(-1)**n
end
```

Testons :

```
irb> load "denombrement.rb"
irb> derangements(3)
=> 2
irb> derangements(26)
=> 148 362 637 348 470 135 821 287 825
```

Le nombre de dérangements pour les 26 lettres de l'alphabet avoisine donc les 148 millions de milliards de milliards ($1,48 \times 10^{26}$), que nous devons comparer avec le nombre de permutations obtenu page ci-contre : $4,03 \times 10^{26}$.

Il y a donc environ trois fois moins de dérangements que de permutations de l'alphabet complet. Ce qui signifie qu'en choisissant une permutation au hasard, nous avons environ une chance sur trois pour que ce soit un dérangement, c'est-à-dire pour qu'absolument toutes les lettres de l'alphabet soient modifiées.

Il n'est pas très difficile de programmer une méthode qui dénombre les permutations laissant au plus trois lettres en place (nous considèrerons que la substitution associée sera une bonne substitution puisque le message d'origine sera très largement modifié).

Pour cela, il faut dénombrer les cas où aucune lettre ne reste à sa place (ce que nous venons de faire), les cas où une lettre exactement reste à sa place (ce que nous allons voir), et enfin les cas où deux lettres exactement puis trois lettres exactement restent à leur place. Nous devrons ensuite ajouter le tout.

Pour compter le nombre de permutations laissant exactement une lettre en place, il suffit de choisir cette lettre (C_1^{26} choix), et pour chacun de ces choix, de déranger complètement les 25 autres lettres (d_{25} choix). Nous avons donc au total $C_1^{26} \times d_{25}$ permutations qui laissent exactement une lettre à sa place. On procède de la même manière pour compter le nombre de permutations laissant exactement deux ou exactement trois lettres à leur place.

En ajoutant le tout (d'aucune lettre à trois lettres à leur place), vous devriez trouver qu'il y a :

$$395\,633\,699\,595\,920\,362\,190\,100\,774 \approx 3,95 \times 10^{26} \text{ possibilités}$$

Si nous comparons ce nombre au nombre total de permutations, qui vaut $4,03 \times 10^{26}$, nous constatons qu'en prenant une permutation au hasard, nous avons plus de 49 chances sur 50 pour qu'il y ait au moins 23 lettres de modifiées sur les 26 de l'alphabet. Cela revient à dire qu'il y a moins d'une chance sur 50 pour que plus de trois lettres sur les 26 ne soient pas modifiées par la substitution. Avec le même raisonnement, nous pourrions calculer qu'il y a moins d'une chance sur 1 500 pour que plus de 5 lettres restent en place.

Exercice : Un peu de dénombrement

solution p. 11

Vous pouvez obtenir ces résultats numériques en programmant la fonction factorielle, ainsi que le nombre de combinaisons. Vous trouverez dans les solutions des versions « naïves » de ces programmes, qui fonctionnent correctement pour les nombres que nous mettons en jeu. Essayez de les écrire, vous tirerez une grande satisfaction en étant capable de retrouver vous-même les résultats annoncés ci-dessus.

Ces résultats justifient le choix d'une substitution monoalphabétique quelconque (au hasard) : dans la grande majorité des cas, cette substitution brouillera correctement le texte d'origine. La « clé » de chiffrement sera alors la permutation de l'alphabet utilisée.

4. L'opérateur ** est l'opérateur puissance.

2.2 Compléments mathématiques sur les dénombrements

Nombre de permutations : Le nombre de permutations d'un ensemble de n éléments, c'est-à-dire le nombre de façons différentes de les ordonner, est :

$$n! = n \times (n-1) \times \cdots \times 3 \times 2$$

Nous pouvons nous en convaincre ainsi : Supposons que les n éléments soient des boules numérotées de 1 à n et que nous devions les disposer dans n cases.

La première case peut être occupée par n'importe quelle boule parmi les n disponibles : nous avons n choix. Une fois cette boule choisie, nous devons remplir la seconde case. Il ne nous reste plus que $n-1$ choix (puisqu'il ne reste que $n-1$ boules). Pour remplir la troisième case, nous avons $n-2$ choix... et ainsi de suite jusqu'à la dernière case, pour laquelle nous n'avons qu'un seul choix, l'unique boule restante. Le nombre de façons (de choix) possibles d'agencer les boules est donc :

$$n \times (n-1) \times (n-2) \times \cdots \times 1 \text{ qui se note } n! \text{ (factorielle } n)$$

Nombre d'arrangements : Le nombre d'arrangements de p objets parmi n est le nombre de façons différentes de choisir et d'ordonner p objets parmi n. Cette quantité est notée A_n^p. Elle vaut :

$$A_n^p = \frac{n!}{(n-p)!}$$

En effet, compter le nombre d'arrangements de p objets parmi n revient à compter le nombre de tirages différents (l'ordre ayant son importance) de p boules numérotées dans un sac qui en contient n. Pour la première boule tirée, nous avons n choix. Puis pour la seconde, puisqu'il en reste $(n-1)$ dans le sac, nous avons $(n-1)$ choix. Pour la dernière (la $p^{\text{ème}}$), nous avons donc $(n-p+1)$ choix. En tout, nous avons donc :

$$\begin{aligned} n \times (n-1) \times \cdots \times (n-p+1) &= \frac{n \times (n-1) \times \cdots \times 2 \times 1}{(n-p) \times (n-p-1) \times \cdots \times 2 \times 1} \\ &= \frac{n!}{(n-p)!} \text{ choix, qui se note } A_n^p \end{aligned}$$

Nombre de combinaisons : Le nombre de combinaisons de p objets parmi n est le nombre de façons différentes de choisir p objets parmi n, l'ordre des objets choisis n'ayant pas d'importance. Cette quantité est notée C_n^p. Elle vaut :

$$C_n^p = \frac{n!}{p!(n-p)!}$$

En effet, pour compter le nombre de combinaisons, il suffit de compter le nombre d'arrangements, et de regrouper ensemble les tirages identiques à l'ordre près. Puisqu'il y a $p!$ manières d'ordonner p éléments, chaque tirage non ordonné donne lieu à $p!$ tirages ordonnés. Il y a donc $p!$ fois moins de combinaisons de p objets parmi n que d'arrangements. Autrement dit :

$$C_n^p = \frac{1}{p!} A_n^p = \frac{n!}{p!(n-p)!}$$

Nombre de dérangements : Le nombre de dérangements d'un ensemble de n objets est le nombre de permutations qui ne laissent aucun élément à sa place. Nous le noterons d_n. Nous allons montrer que :

$$\begin{cases} d_0 = 1 \\ d_n = n\, d_{n-1} + (-1)^n \end{cases}$$

Pour cela, montrons tout d'abord le premier résultat suivant :

$$d_n = (n-1)\left(d_{n-1} + d_{n-2}\right)$$

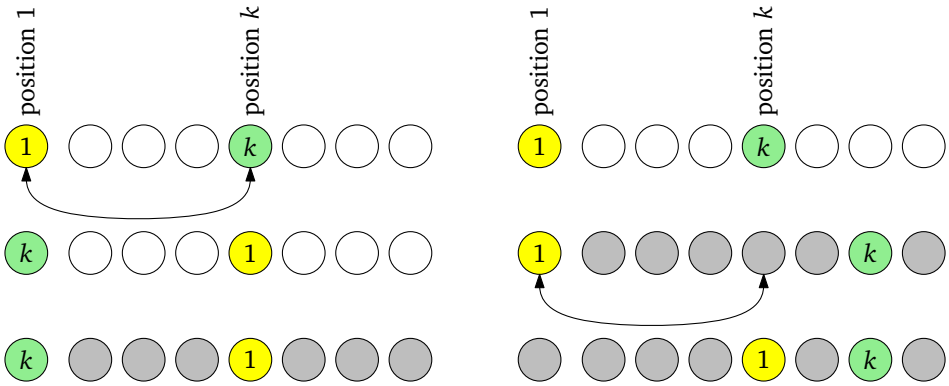
Les dérangements de n objets peuvent être divisés en deux catégories :

1. La catégorie des dérangements où le premier objet est *échangé* avec un autre.
2. La catégorie des dérangements où le premier objet est placé en position k et l'objet k est placé dans une autre position que 1.

On conçoit aisément qu'un dérangement appartient nécessairement à l'une ou l'autre de ces deux catégories.

Pour dénombrer les dérangements de la première catégorie (figure 4.1(a)), il faut choisir l'élément k avec lequel 1 est échangé ($n-1$ choix), puis déranger les $n-2$ éléments restants (d_{n-2} choix). Cette première catégorie comporte donc $(n-1)\,d_{n-2}$ dérangements.

Pour dénombrer les dérangements de la seconde catégorie (figure 4.1(b)), nous pouvons déranger tous les objets sauf le premier (il y a d_{n-1} choix), puis échanger l'objet 1 avec un des $n-1$ objets dérangés (il y a $n-1$ choix).



(a) On échange les objets 1 et k, puis on dérange tous les autres objets.

(b) On dérange tous les objets sauf le premier, puis on échange celui-ci avec n'importe quel autre objet.

Figure 4.1 – Une méthode pour dénombrer les dérangement en les divisant en deux catégories

Nous trouvons donc au total :

$$d_n = (n-1)\,d_{n-2} + (n-1)\,d_{n-1} = (n-1)\,\left(d_{n-1} + d_{n-2}\right)$$

Il n'y a aucune façon de déranger 1 seul objet, donc $d_1 = 0$. Il y a une seule façon de déranger 2 objets (on les échange), donc $d_2 = 1$.

Remarquons qu'en choisissant $d_0 = 1$, la formule permet de recalculer d_2 :

$$d_2 = (2-1)(d_1 + d_0) = 1$$

Nous disposons donc maintenant d'un premier moyen pour calculer le nombre de dérangements :

$$\begin{cases} d_0 = 1 \\ d_1 = 0 \\ d_n = (n-1)(d_{n-1} + d_{n-2}) \text{ pour } n > 1 \end{cases}$$

Écrivons la relation de récurrence ainsi :

$$\begin{aligned} d_n &= (n-1)(d_{n-1} + d_{n-2}) \\ d_n - n\,d_{n-1} &= -d_{n-1} + (n-1)\,d_{n-2} \end{aligned}$$

Posons à présent :

$$u_n = d_n - n\,d_{n-1}$$

Nous avons donc aussi (en appliquant le résultat qui précède à u_{n-1} et en prenant l'opposé) :

$$-u_{n-1} = -d_{n-1} + (n-1)\,d_{n-2}$$

Nous obtenons ainsi :

$$\begin{cases} u_1 = d_1 - d_0 = -1 \\ u_n = -u_{n-1} \end{cases}$$

Par conséquent, $u_n = (-1)^n$.

En reportant ce résultat dans $u_n = d_n - n\,d_{n-1}$, nous obtenons :

$$(-1)^n = d_n - n\,d_{n-1}$$

Autrement dit :

$$\begin{cases} d_0 = 1 \\ d_n = n\,d_{n-1} + (-1)^n \text{ pour } n > 0 \end{cases}$$

Notons au passage qu'il existe d'autres « formules » pour calculer le nombre de dérangements. Le résultat qui suit est facilement obtenu en utilisant la relation de récurrence que nous avons déjà montrée :

$$d_n = \sum_{0 \leq k \leq n} (-1)^k \frac{n!}{k!}$$

En effet, la relation qui précède est vraie pour $n = 0$ car nous avons bien [5] :

$$d_0 = \sum_{0 \leq k \leq 0} (-1)^k \frac{0!}{k!} = (-1)^0 \frac{0!}{0!} = 1$$

Supposons à présent que la formule est vraie au rang $n-1$:

$$d_{n-1} = \sum_{0 \leq k \leq (n-1)} (-1)^k \frac{(n-1)!}{k!}$$

En utilisant la relation de récurrence que nous venons de montrer, nous obtenons :

$$\begin{aligned} d_n &= (-1)^n + n\,d_{n-1} \\ &= (-1)^n + n \times \sum_{0 \leq k \leq (n-1)} (-1)^k \frac{(n-1)!}{k!} \\ &= (-1)^n + \sum_{0 \leq k \leq (n-1)} (-1)^k \frac{n!}{k!} \\ &= (-1)^n \frac{n!}{n!} + \sum_{0 \leq k \leq (n-1)} (-1)^k \frac{n!}{k!} \\ &= \sum_{0 \leq k \leq n} (-1)^k \frac{n!}{k!} \end{aligned}$$

Nous venons donc de démontrer que :

$$\forall n \geq 0, d_n = \sum_{0 \leq k \leq n} (-1)^k \frac{n!}{k!}$$

3 Compléments sur les chaînes de caractères

Cette section propose une alternative, basée sur l'utilisation des codes Ascii des caractères, à la programmation du chiffre de César présentée dans les Dmi.

5. Rappelons que la factorielle de 0 vaut 1.

3.1 Code Ascii

L'équivalence entre les nombres et les caractères est plus complexe qu'il n'y paraît. Historiquement, l'alphabet utilisé dans les premiers ordinateurs a été l'alphabet latin pour anglophones (les 26 lettres, sans caractères accentués). En comptant les capitales, les minuscules, les chiffres, et les différents caractères de ponctuation, une correspondance a été établie entre 128 caractères et les nombres de 0 à 127[6]. Cette correspondance s'appelle le code Ascii. C'est une norme internationale. Ainsi, la lettre `A` par exemple, est représentée par le code 65. À mesure que les ordinateurs se sont répandus, d'autres alphabets ont été utilisés et d'autres symboles ont été nécessaires. La numérotation des symboles étant possible jusqu'à 255[7], il s'en est suivi que les caractères numérotés de 128 à 255 ne sont pas normalisés, et dépendent du langage utilisé. Il y a en effet beaucoup trop de caractères manquants dans le code Ascii pour leur affecter à chacun un code qui va de 128 à 255. Autrement dit, si en France le caractère `é` a pour code 233, il est possible que dans un autre pays, en particulier s'il n'utilise pas l'alphabet latin, le code 233 corresponde à un autre caractère. Il existe beaucoup de ces extensions de l'Ascii à 8 bits, et Ruby utilise probablement celle par défaut de votre machine (sur la machine utilisée lors de la rédaction de cet ouvrage, il s'agit de l'extension Latin-1 ou Iso-8859-1).

Pour compenser l'inconvénient des codages ne contenant pas tous les caractères, et donc leur diversité, il en existe d'autres utilisant des tables de plus de 255 caractères et des codes de longueur variable, comme Utf-8... Ruby peut utiliser certains de ces codages, mais la représentation par défaut reste l'Ascii étendu « local » avec des codes allant de 0 à 255.

En Ruby, passer du code Ascii au caractère est très simple. Le code associé au caractère `M`, par exemple, est donné par `?M`. Inversement, le caractère représenté par le code 77 est donné par `77.chr`. Notons au passage qu'il existe un moyen de mentionner les caractères spéciaux comme tabulation ou retour à la ligne autrement que par leur code Ascii. Il suffit d'utiliser le caractère d'échappement `\` suivi d'une lettre : `\n` pour un retour à la ligne (*newline*), `\t` pour une tabulation, `\b` pour l'effacement arrière (*backspace*).

3.2 Accès à des portions de chaînes

Voyons tout d'abord comment accéder à des « parties » de chaînes de caractères :

```
irb> str="Le temps tue le temps comme il peut"
irb> str[5]   # Renvoie le code Ascii du 6ème caractère de la chaîne
=> 109
irb> str[3..7] # Renvoie une tranche de la chaîne de caractères
=> "temps"
irb> str[5..5] # Renvoie une tranche réduite à un seul caractère
=> "m"
```

Il y a deux manières d'accéder à des tranches de caractères :
– en donnant un intervalle : `str[0..2]`
– en donnant le point de départ et le nombre de caractères : `str[0,3]`
Ainsi `str[0,3]` fait référence à la même tranche de chaîne que `str[0..2]`. De même, `str[3,5]` équivaut à `str[3..7]` :

```
irb> str="Le temps tue le temps comme il peut"
irb> str[3..7]
=> "temps"
irb> str[3,5]
=> "temps"
```

3.3 Arithmétique du code Ascii

Nous pouvons programmer le chiffre de César en travaillant sur les codes Ascii[8] :

6. Sur ces 128 « caractères », le premier (numéro 0) est plutôt un « non-caractère » (c'est le caractère nul). Les 127 autres peuvent être imprimables (lettres, chiffres, ponctuation), ou non (retour à la ligne, bip clavier, tabulation...).
7. La valeur 255 correspond au plus grand nombre qui peut être stocké dans un octet, c'est-à-dire sur 8 chiffres binaires, ou 8 bits.
8. Les détails du fonctionnement du chiffre de César sont donnés dans les Dwi. Vous trouverez facilement sa description sur Internet.

Les lettres étant rangées dans l'ordre alphabétique dans la table ASCII, décaler de 3 lettres revient à ajouter 3 au code, en prenant garde de revenir sur `A` après `Z`.

Testons notre idée :

```
irb> str="G"
irb> str[0]
=> 71
irb> str[0]+3
=> 74
irb> (str[0]+3).chr
=> "J"
```

Le principe fonctionne.

Pour chiffrer toute une chaîne, nous allons l'épeler, octet par octet, à l'aide de la méthode `each_byte` :

```
irb> s="SALUT"
irb> s.each_byte { |c| p c }
83
65
76
85
84
```

Notez l'utilisation de la méthode `p` (elle a un nom très court, mais c'est pourtant une méthode) qui prend en paramètre un objet et « l'affiche » proprement à l'écran.

La méthode de cryptage par le chiffre de César pourrait donc être écrite ainsi :

```
class String
...
  def cesar1(k)
      k=k+26 if k<0
      res=""
      self.each_byte do |c|
          c=c+k
          c=c-26 if c>?Z
          res << c.chr
      end
      return res
  end
end
```

Après avoir créé une chaîne vide (`res`), nous itérons sur la chaîne à chiffrer, et pour chaque code, ajoutons `k` (et retranchons éventuellement 26 si le nombre dépasse le code de la lettre lettre `Z`). Puis, nous ajoutons au bout de la chaîne de caractères la lettre correspondant au nouveau code calculé (nous voyons que l'opérateur `<<` des tableaux fonctionne aussi pour les chaînes de caractères).

Bien que cette méthode fonctionne, l'utilisation de l'itérateur `each_byte` est un peu gênante. En effet, si les caractères venaient à être représentés par un code sur plus d'un octet (c'est-à-dire si le code d'un caractère était supérieur à 255), notre système ne fonctionnerait plus car `each_byte` « épellerait » octet par octet plutôt que caractère par caractère. C'est pourquoi une autre solution, basée sur `each_char` et indépendante du codage utilisé, est proposée dans les DMI.

Solution de l'exercice : Dénombrement des substitutions

énoncé p. 5

denombrement.rb

```
# Factorielle de n
# = nombre de permutations de n objets
def factorielle(n)
  r=1
  for i in 2..n : r=r*i end
```

```
  return r
end

# Nombre de combinaisons de p objets parmi n
def combinaisons(n,p)
  return factorielle(n)/factorielle(p)/factorielle(n-p)
end

# Nombre de dérangements de n objets
# = nombre de permutations de n objets ne laissant
#   aucun objet à sa place
def derangements(n)
  return 1 if n<=0
  return n*derangements(n-1)+(-1)**n
end

# Nombre de permutations de n objets laissant k objets
# exactement à leur place
def derangements_enplace(n,k)
  return combinaisons(n,k)*derangements(n-k)
end

# Nombre de permutations de n objets laissant au plus
# k objets à leur place
def derangements_auplus(n,k)
  r=derangements(n)
  for i in 1..k
    r=r+derangements_enplace(n,i)
  end
  return r
end
```

Nous pouvons vérifier que ces méthodes donnent les résultats donnés dans la section 2 :

```
irb> load "denombrement.rb"
irb> fact(26) # factorielle de 26
=> 403 291 461 126 605 635 584 000 000
irb> combinaisons(4,2) # nb de combinaisons de 2 objets parmi 4
=> 6
irb> derangements(26) # nb de dérangements de 26 objets
=> 148 362 637 348 470 135 821 287 825

# nb de dérangements laissant exactement deux lettres en place
irb> derangements_enplace(26,2)
=> 74 181 318 674 235 067 910 643 925
```

Le résultat indiquant le nombre de permutations qui laissent au plus 3 lettres en place est obtenu ainsi :

```
irb> load "denombrement.rb"
irb> derangements_auplus(26,3)
=> 395 633 699 595 920 362 190 100 774
```

Nous retrouvons bien la valeur $3,95 \times 10^{26}$ de la page 5.

N'hésitez pas à aller pus loin [9] :

```
irb> load "denombrement.rb"

# Combien de substitutions laissent au plus 5 lettres en place ?
```

9. Notez la présence du `1.0` dans le calcul de `c`, qui évite que Ruby ne calcule une valeur entière.

```
irb> a=derangements_auplus(26,5)
=> 403 051 831 463 343 868 981 162 924

# Combien de substitutions laissent strictement plus de 5 lettres en place ?
irb> b=fact(26)-a
=> 239 629 663 261 766 602 837 076
# Probabilité pour que strictement plus de 5 lettres restent en place ?
irb> c=1.0*b/fact(26)
=> 0.000594184817581693

irb> puts("Cet événement arrivera une fois sur #{(1/c).to_i} environ")
Cet événement arrivera une fois sur 1682 environ
```