8. Quelques notions de combinatoire

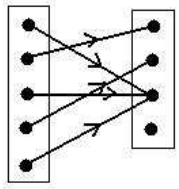
Dans le chapitre précédent, nous avions vu comment énumérer des mots ou des configurations. Nous allons maintenant ajouter quelques considérations théoriques donnant les formules permettant de compter le nombre de ces mots.

1. Nombre de mots de longueur donnée à base de deux lettres 0 et 1

Cela revient à compter les nombres de longueur donnée *P* écrits en binaire. Par exemple, pour *P* = 3, on a les mots 000, 001, 010, 011, 100, 101, 110, 11I, soit 8 mots. Quelle est la formule donnant le nombre de tels mots de longueur *P* ? On commence par choisir la première lettre, 0 ou 1, soit deux cas. Une fois choisie cette lettre, on a deux cas pour la deuxième lettre. Et à chaque fois, on a encore deux cas pour la troisième, etc. Finalement le nombre de ces mots est $2 . 2 . 2 \ldots 2 = 2^P$.

2. Généralisation : Nombre d'applications d'un ensemble à *p* éléments dans un ensemble à *n* éléments

Généralisons : le nombre de mots de longueur *p* à base de *n* lettres est n^p. Cela peut être vu autrement, en termes d'applications.



Par définition, une application d'un ensemble de départ *E* à *p* éléments dans un ensemble d'arrivée *F* à *n* éléments fait correspondre à chaque élément de l'ensemble de départ un élément unique de l'ensemble d'arrivée. Cela se visualise avec des flèches issues de chaque élément du départ et tombant dans l'ensemble d'arrivée.

En appelant 1,2,3,4,5 les *p* éléments du départ et *a,b,c,d* les *n* éléments de l'arrivée dans le dessin précédent (avec *p* = 5 et *n* = 4), l'application peut aussi bien s'écrire sous forme d'un tableau avec ses cases indexées de 1 à 5 et contenant les lettres correspondantes. On retrouve ainsi un mot de longueur 5 à base de quatre lettres, les lettres *a* et *b* étant présentes une fois, la lettre *c* trois fois, et la lettre *d* absente, soit :

1	2	3	4	5
c	a	c	b	c

Reprenons le raisonnement pour compter le nombre d'applications d'un ensemble à *p* éléments dans un ensemble à *n* éléments. A partir du premier élément, on envoie une flèche dans l'arrivée, ce qui donne *n* possibilités. Chaque fois que l'on a fait cela, à partir du deuxième élément, on envoie une flèche qui a aussi cinq éléments à l'arrivée possibles, etc. On trouve bien n^p applications.

Remarque : comme on a une chance sur deux d'écrire p^n au lieu de n^p, mieux vaut ne pas apprendre la formule par cœur, mais refaire le raisonnement du comptage.

Exercice : *Enumérer par programme tous les mots de longueur P à base de N lettres.*

Il suffit de généraliser ce que l'on a fait avec les mots en binaire dans le chapitre précédent :

```
for(i=0 ;i<P ;i++) a[i]=0 ;
for( ;;)
{ afficher le tableau a[P]
i=P-1;   while(I > = 0 && a[i]==N-1) i--;
pospivot=I;   if (pospivot==-1) break;
a[pospivot]++;
for(i=pospivot+1; i<P;i++) a[i]=0;
```

}

3. Nombre de tiercés dans l'ordre dans une course de chevaux

Prenons par exemple une course avec cinq chevaux numérotés de 1 à 5. Les tiercés possibles dans l'ordre sont : 123, 124, 125, 132, 134, 135, 142, 143, 145, 152, 153, 154, 213, 214, etc. Remarquons qu'on a trouvé 12 tiercés où le cheval arrive premier. Il y en a autant où c'est le cheval 2 qui arrive premier. D'où 5. 12 = 60 tiercés dans l'ordre. Petite question : combien de chevaux peuvent arriver premier ? Les cinq évidemment. Combien peuvent arriver deuxième ? Les cinq aussi. Mais il convient de s'y prendre autrement si l'on veut connaître le nombre des tiercés possibles. Voici comment :

Pour le premier cheval à l'arrivée, il y a cinq cas. A chaque fois (qu'on a choisi le premier cheval à l'arrivée), il reste quatre possibilités pour le deuxième. Et à chaque fois, il reste trois cas pour le troisième. D'où 5.4.3= 60 tiercés dans l'ordre.

Remarque : si l'on oublie le « à chaque fois », le raisonnement devient complètement faux. En plus c'est ce « à chaque fois » qui oblige à faire des multiplications, et non pas des additions !

4. Généralisation : Nombre d'arrangements de *p* objets pris parmi *n* objets

Par définition, comme son nom l'indique, un arrangement de *p* objets pris parmi *n* est une certaine façon de prendre ces *p* objets parmi les *n* en tenant de l'ordre dans lequel on les prend. Par exemple, les tiercés dans l'ordre dans une course de *n* chevaux sont les arrangements de *p* = 3 objets pris parmi *n*. En généralisant, on obtient le nombre d'arrangements de *p* objets pris parmi *n*, ce nombre d'arrangements étant noté A_n^p :

$\boldsymbol{A_n^p = n\,(n\text{-}1)(n\text{-}2)\ldots\,(n\text{-}p\text{+}1)}$
avec la présence de *p* facteurs (d'où le dernier facteur qui est *n*-(*p*-1) puisque le premier est *n*-0.

Remarquons que si *p* est supérieur à *n*, le nombre des arrangements est nul, et que si *p* = *n*, on retombe sur le nombre de permutations : *n* !

5. Nombre de tiercés dans le désordre

Reprenons notre course de cinq chevaux. Un tiercé dans le désordre est par exemple 123, mais il peut aussi bien s'écrire 132, 213, 231, 312 ou 321. En fait il peut s'écrire de six façons, autant qu'il y a de permutations de trois objets (3 ! = 6). Sur ces six façons nous en choisissons une, et évidemment nous prenons 123. Autrement dit, pour un tiercé dans le désordre, on choisit une écriture où l'ordre joue : on met les lettres (les chiffres) dans l'ordre croissant (ou alphabétique) à l'intérieur du mot. En tenant compte de ce choix d'écriture, cela donne les tiercés : 123, 124, 125, 134, 135, 145, 234, 235, 245, 345. Comme nous l'avons déjà vu sur l'exemple de 123, chaque tiercé dans le désordre correspond à six tiercés dans l'ordre. Il y a six fois moins de tiercés dans le désordre que de tiercés dans l'ordre, soit 60 / 6 = 10.

6. Généralisation : Nombre de combinaisons de *p* objets pris parmi *n*

Par définition, comme son nom ne l'indique pas vraiment, une combinaison de *p* objets pris parmi *n* est une certaine façon de prendre *p* objets parmi les *n* sans tenir compte de l'ordre. Cela revient à prendre un paquet de *p* objets.

Par analogie avec les tiercés dans le désordre, le nombre de combinaisons de *p* objets pris parmi *n*, noté C_n^p, s'obtient en divisant le nombre d'arrangements A_n^p par le nombre de permutations de *p* objets.

$$C_n^p = \frac{n(n-1)(n-2)\ldots(n-p+1)}{p!}$$

A noter la présence de *p* facteurs au numérateur. [1]

Si C_n^p désigne le nombre de façons de prendre un paquet de *p* objets parmi *n*, on peut voir aussi voir cela autrement. C_n^p est aussi le nombre de mots de longueur *n* à base de deux lettres, avec la présence de *p* lettres 0 et *n-p* lettres 1. En effet comment fait-on pour obtenir de tels mots ? On se place devant un casier avec *n* cases vides. Puis on lance dans ces cases les *p* lettres 0 (jamais plus d'une par case). Il y a autant de façons de placer ces lettres 0 qu'il y a de façons de choisir *p* cases parmi *n*, soit C_n^p . Chaque fois que l'on a fait cela, les lettres 1 prennent les cases restantes. Finalement le nombre des mots ainsi construits est C_n^p. Rappelons que c'est en écrivant de tels mots dans le chapitre précédent que l'on a pu énumérer toutes les combinaisons de *p* objets parmi *n*.

7. Deux formules sur les combinaisons

En matière de combinaisons, ce ne sont pas les formules qui manquent, mais voici les deux à savoir d'office.

- $C_n^p = C_n^{n-p}$

En effet il y a autant de façons de prendre *p* objets parmi *n*, soit C_n^p, qu'il y a de façons de laisser *n-p* objets parmi les *n*, soit C_n^{n-p}.

Ainsi, si l'on nous demande de calculer C_{15}^{12}, on s'empressera de remplacer cela par C_{15}^{3} avec trois facteurs en haut et en bas au lieu de 12. On fait ce remplacement pour C_n^p dès que *p* dépasse *n* / 2.

- $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$

Pour le démontrer, privilégions l'un des *n* objets, par exemple en le coloriant en rouge. Lorsque l'on prend *p* objets parmi les *n* de toutes les façons possibles, soit C_n^p façons, il se présente deux éventualités :

* Soit l'objet rouge fait partie des *p* objets que l'on a pris. Alors les *p*-1 objets autres que lui sont choisis par *n*-1 objets (tous sauf le rouge). Cela donne C_{n-1}^{p-1} façons.

* Soit l'objet rouge ne fait pas partie des p objets que l'on a pris. Ces *p* objets sont choisis parmi *n*-1 objets (tous sauf le rouge). Cela donne C_{n-1}^p cas.

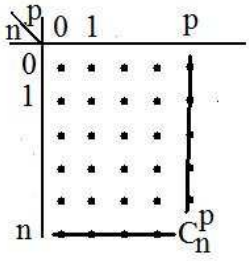
Finalement on a bien $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$. Reste à voir l'intérêt de cette formule.

---

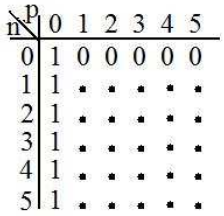
[1] Signalons que C_n^p s'écrit aussi $\binom{n}{p}$ et que la formule $C_n^p = \frac{n(n-1)(n-2)\ldots(n-p+1)}{p!}$ peut aussi s'écrire $C_n^p = \frac{n!(n-p)!}{p!}$.

Cette formule, unifiée par la présence de factorielles partout, a parfois des avantages dans les calculs théoriques, notamment quand on utilise la formule de Stirling qui donne une approximation des factorielles, mais en général elle est ridicule, puisqu'elle présente des redondances au numérateur et au dénominateur. Comme si en allant au marché on demandait à acheter, en signe de supériorité intellectuelle, 6/3 kilos de pommes de terre.

8. Le triangle de Pascal

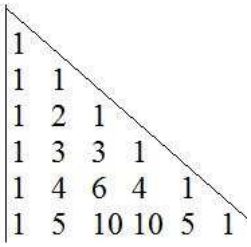


Prenons un tableau à deux dimensions, avec ses lignes et ses colonnes. On va le remplir en plaçant sur la ligne *n* et la colonne *p* le nombre C_n^p. On sait déjà que pour *p>n* on a $C_n^p = 0$. D'autre part, dans la colonne 0, on met partout $C_n^0 = 1$. On considère en effet que le nombre de façons de ne prendre aucun objet parmi les *n* vaut 1 : ne rien faire compte pour un (même pour C_0^0). On peut ainsi remplir la ligne 0 ainsi que la colonne 0. A partir de là, tout le reste s'ensuit, ligne après ligne. Il ne reste plus qu'à appliquer la formule : $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$. Sur la ligne *n*, en colonne *p*, on obtient C_n^p en additionnant ce qui se trouve dans la ligne d'au-dessus sur la même colonne et sur celle d'avant.



Cela revient à faire coulisser une sorte d'équerre le  long d'une ligne, et de remplir à chaque fois la case du bas en additionnant ce qui est dans les deux cases du dessus.

On remplit ainsi le triangle de Pascal, le reste étant formé de 0 :



Cela permet de calculer C_N^P pour *P* et *N* donné. Il suffit de remplir le rectangle délimité par *P* en longueur et par *N* en descente, en partant de la ligne 0 et de la colonne 0, et en remplissant le reste par des additions successives. [2] D'où le programme :

```
 int C[N+1][P+1] ;  /* à déclarer au départ */
for(ligne=0 ; ligne<= N ; ligne++) C[ligne][0]=1 ;  /* colonne 0 */
for(colonne=1 ; colonne<=P ; colonne++) C[0][colonne]=0 ; /* ligne 0 */
/* et maintenant le remplissage du rectangle */
for(ligne=1 ; ligne<=N ; ligne++) for(colonne=1 ; colonne<=P ; colonne++)
C[ligne][colonne]=C[ligne-1][colonne-1]+C[ligne-1][colonne] ;
afficher C[N[P]
```

Mais il n'est pas indispensable de prendre un tableau à deux dimensions. Il suffit de prendre deux tableaux linéaires : à partir de l'un (*olda*[]) qui correspond à une ligne, on remplit l'autre (*newa*[]) qui correspond à la ligne suivante, puis on met dans l'ancien ce qui était dans le nouveau pour préparer l'étape suivante.

```
int olda[P+1], newa[P+1] ;
olda[0]=1 ; for(i=1 ;i<=P ; i++) olda[i]=0 ;
for(ligne=1 ; ligne<=N ; ligne++)
  { for(i=1 ;i<=P ;i++) newa[i]=olda[i-1]+olda[i] ;
    for(i=1 ;i<=P ;i++) olda[i]=newa[i] ;
```

[2] Lorsque l'on fait le calcul à la main, on utilise plutôt la formule :
$C_n^p = \frac{n(n-1)(n-2)\ldots(n-p+1)}{p!}$,
et l'on procède ensuite à des simplifications en haut et en bas jusqu'à ce qu'il ne reste plus rien en bas. Mais cela est inadapté au calcul automatique, puisqu'il faudrait d'abord décomposer chaque nombre en produit de facteurs premiers, avant de pouvoir lancer les simplifications en chaîne.

```
  }
afficher olda[P] ;
```

Enfin, mieux encore, on peut se contenter d'utiliser un seul tableau, en écrasant les anciennes valeurs par les anciennes, du style *a*[*i*]+=*a*[*i*-1] : pour avoir le nouvel *a*[*i*], celui de la ligne d'après, on prend l'ancien *a*[*i*] et on lui ajoute *a*[*i*-1]. Mais en faisant cela, encore faut-il que l'on prenne l'ancien *a*[*i*-1], et si l'on va de gauche à droite, cela ne sera pas valable. Par exemple à partir de la ligne 0 : 1 0 0 0 …, on obtiendrait 1111… Il convient d'aller de droite à gauche, le nouvel *a*[*i*-1] étant alors calculé après le nouvel *a*[*i*] qui lui utilise l'ancien *a*[*i*]:

```
int a[P+1] ;
a[0]=1 ;
for(ligne=1 ; ligne<=N ; ligne++)
for(i=P ;i>=1 ; i--) a[i]+=a[i-1] ;
afficher a[P]
```

9. La formule du binôme

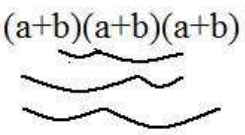
Souvenons-nous des identités remarquables, comme $(a + b)^2 = a^2 + 2ab + b^2$. Les coefficients successifs 1, 2, 1 correspondent à la ligne 2 du triangle de Pascal.

Prenons l'identité remarquable suivante, un peu moins connue :
$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$. Les coefficients 1, 3, 3, 1 correspondent encore à une ligne du triangle de Pascal. Cela se généralise, et l'on aboutit à ce que l'on appelle la formule du binôme (de Newton) :

$$\mathbf{(a + b)^n = C_n^0\, a^n + C_n^1\, a^{n-1}\, b + C_n^2\, a^2\, b^{n-2} + \ldots + C_n^n\, b^n},$$

où l'on reconnaît la $n^{\text{è}}$ ligne du triangle de Pascal pour les coefficients.

Pour comprendre d'où cela vient, reprenons l'exemple de $n = 3$, en écrivant :
$(a+b)^3 = (a+b)\,(a+b)\,(a+b)$. Pour développer ce produit de trois facteurs, on utilise la distributivité de la multiplication par rapport à l'addition, en multipliant un terme de la première parenthèse avec un terme de la deuxième et avec un terme de la troisième, cela de toutes les façons possibles.



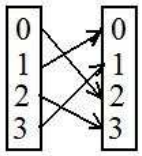
D'où viennent par exemple les termes en a^2b, soit $3a^2b$? Il s'agit de prendre *a* dans deux des parenthèses et *b* dans la parenthèse restante, ce qui donne a^2b, et cela de toutes les façons possibles. Cela revient à choisit une parenthèse parmi 3 (celle où est *b*), les *a* étant pris dans les deux parenthèses restantes, soit $C_3^1 = 3$ cas. D'où la présence de $3a^2b$. Et cela se généralise.

10. Retour aux permutations

On a *n* objets, par exemple un jeu de *n* cartes. Rien n'empêche de leur donner à chacune un numéro, de 0 à *n*-1. Comme on l'a vu, le fait de ranger ces objets numérotés dans un certain ordre est appelé une permutation. Par exemple une permutation de $n = 4$ objets est 2031, une autre est 0123, qui correspond à l'ordre naturel. Il s'agit là d'une vision statique : si l'on a un jeu de 4 cartes en main, on constate par exemple que l'on a les cartes 2031 devant soi.

On peut aussi voir cette permutation d'une façon dynamique, en l'écrivant $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 3 & 1 \end{pmatrix}$, ce qui correspond exactement à l'écriture informatique dans un tableau :

0	1	2	3
2	0	3	1



Cela peut maintenant se lire par colonne : 0 va en 2, 1 va en 0, 2 va en 3, et 3 va en 1 et l'on retrouve la notion d'application, et même de bijection (chaque élément du départ admet un correspondant unique à l'arrivée, et chaque élément de l'arrivée admet un prédécesseur unique).

Mieux encore, il n'est plus utile de numéroter les objets, nos cartes par exemple. On se contente de noter les changements de position des objets. On lit maintenant cette permutation ainsi : l'élément qui était en position 0 se retrouve en position 2, celui en position 1 va en position 0, etc.

Faisons maintenant une lecture à l'envers, en échangeant le départ et l'arrivée, ainsi que le sens des flèches. Dans le tableau lu colonne par colonne de bas en haut, cela donne : 2 va en 0, 0 va en 1, 3 va en 2, 1 va en 3. On obtient ce que l'on appelle la permutation inverse de la précédente, ct il s'agit de :

0	1	2	3
1	3	0	2

On en déduit le programme qui permet de passer d'une permutation à son inverse. La permutation initiale est placée dans un tableau $a[N]$. La permutation inverse va être construite en remplissant progressivement un tableau $b[N]$. Pour l'obtenir on parcourt le tableau $a[]$:

```
for(i=0 ;i<N ;i++)
b[a[i]] = i ;
```

11. Décomposition d'une permutation en cycles

Reprenons notre permutation 2031 qu'il vaut mieux écrire dorénavant $\begin{pmatrix}0&1&2&3\\2&0&3&1\end{pmatrix}$ si l'on veut comprendre comment elle agit. Puis faisons une lecture par enchaînement : 0 va en 2, à son tour 2 va en 3, puis 3 va en 1, puis 1 va en 0, et l'on est retourné au point de départ. On peut écrire cette permutation sous forme d'un cycle unique :

Prenons d'autres exemples, où l'on obtient plusieurs cycles, avec la présence possible de cycles de longueur un. Ainsi :

$\begin{pmatrix}0&1&2&3&4&5\\2&4&1&5&0&3\end{pmatrix}$ s'écrit sous forme de deux cycles (0 2 1 4) (3 5)

$\begin{pmatrix}0&1&2&3&4&5\\2&0&1&5&4&3\end{pmatrix}$ s'écrit (0 2 1) (3 5) (4) , avec 4 qui reste en 4 (c'est un point fixe).

Cela se généralise : toute permutation peut s'écrire sous forme de cycles disjoints (ils n'ont aucun élément en commun). Et cette écriture est unique, à l'ordre des cycles près.[3]

[3] Démonstration : Prenons une permutation de n éléments. L'élément 0 est transformé en un certain élément (parmi les n), puis celui-ci est transformé en l'un des n éléments, etc. Comme le nombre d'objets est le nombre fini n, il est sûr que l'on finisse par repasser là où l'on est déjà passé, d'où une trajectoire de l'élément 0 de la forme :

Mais ici l'élément e possède deux antécédents, ce qui n'est pas possible puisqu'une permutation est une bijection. La seule possibilité est :

Programmons cette décomposition d'une permutation en cycles. Comme d'habitude, on ne va pas traiter ce problème d'un seul coup. Il convient d'abord de faire une fonction qui ramène le cycle d'un élément quelconque. Puis on utilisera cette fonction pour chaque élément. En faisant cela, l'ennui c'est de répéter les mêmes cycles plusieurs fois, car on a le même cycle pour chacun des éléments de ce cycle. Il conviendra de rectifier.

Supposons que notre permutation soit enregistrée dans un tableau *a*[*N*], et commençons par la fonction donnant le cycle d'un élément *i* , sachant que le successeur de *i* est *a*[*i*]:

```
void cycle(int idebut)
{   printf(«(») ; i= idebut;
    do { printf(« %d »,i) ; i=a[i] ; }   while (i!= idebut);
    printf(«)»);
}
```

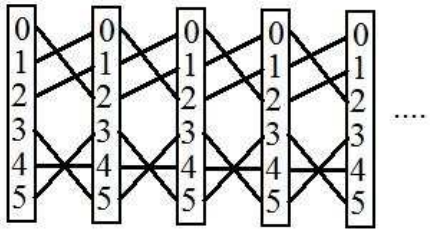
Faisons maintenant le programme ramenant tous les cycles, sans les répéter. Pour éviter ces répétitions, on gère un tableau *fini*[*N*] qui contient OUI ou NON selon que l'élément *i* a déjà été mis dans un cycle ou non. On met progressivement à *fini*=1 les éléments qui sont placés dans un cycle, en ajoutant cela à la fonction précédente. Le tableau *fini*[] est déclaré en global, et ainsi mis à 0 (NON) au départ.

```
remplir le tableau a[N]  (et  fini[N] est à NON)
for(i=0 ; i<N ; i++)  if (fini[i] == NON)  cycle (i);

void cycle(int idebut)
{   printf(«(») ; i= idebut;  /* i va décrire tous les éléments du cycle */
    do { printf(« %d »,i) ; fini[i]=OUI;   i=a[i] ; }
    while (i!= idebut);
    printf(“)”);
}
```

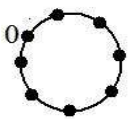
12. Période d'une permutation

Enchaînons plusieurs fois la même permutation, ce qui revient à répéter à plusieurs reprises la même opération de mélange. Par exemple, voici ce que cela donne pour la permutation $\begin{pmatrix} 0&1&2&3&4&5 \\ 2&0&1&5&4&3 \end{pmatrix}$:



Suivons le mouvement de 0 : 0 va en 2, puis 2 va en 1 puis 1 va en 0, et l'on vient de retrouver le cycle de 0. Ainsi tous les trois coups -la longueur du cycle de 0-, 0 revient en 0. Il en est de même pour

---



La trajectoire de 0, avec son successeur, son successeur de successeur, etc., est bien un cycle. Si ce cycle ne contient pas les *n* éléments, on prend le premier élément qui n'est pas dans le cycle, et l'on prend ses successeurs, ce qui donne encore un cycle. Et ainsi de suite. Avec le nombre fini d'éléments, on obtient des cycles qui finissent par inclure tous les éléments. Et comme chaque élément n'a qu'un successeur, cette écriture sous forme de cycles est unique.

les autres éléments de ce cycle, à savoir pour 2 et pour 1. A leur tour, les éléments 3 et 5 reprennent leur place tous les deux coups, et 4 garde constamment la sienne. Ainsi pour les multiples communs des longueurs des cycles de la permutation, chaque élément redevient lui-même. Cela arrive pour la première fois en prenant le *ppmc* des longueurs des cycles, soit 6 dans notre exemple. Un tel nombre de coups, permettant de revenir au point de départ, s'appelle la période de la permutation. Concrètement, cela signifie qu'une personne capable de mélanger un jeu de cartes toujours de la même façon (avec la même permutation sous-jacente) est assurée de revenir à l'ordre initial des cartes au bout d'un nombre de coups qui est le *ppmc* des longueurs des cycles de la permutation. De nombreux tours de cartes utilisent cette propriété.