2019

Projet Euler

RAPPORT DE PROJET

KILLIAN DERRIEN - CORENTIN MOULY

Tables des matières

Introduction

Le projet a eu pour but de résoudre des problèmes d'Euler (disponible sur https://projecteuler.net/archives) pour cela nous avons abordé différents problèmes avec une difficulté croissante.

Les problèmes ont été traités selon différents angles, dans un premier temps on a appliqué différentes connaissances mathématiques afin de résoudre le plus simplement possible le problème. Celui-ci a été traduit en un algorithme structuré, puis mis sous forme de programme informatique à l'aide du langage C#.

Nous avons également considéré l'ensemble des opérations permettant l'optimisation du premier algorithme. La dernière étape fut destinée à l'analyse du résultat obtenu ainsi qu'au temps d'exécution et à la complexité de l'algorithme.

I/ Under The Rainbow (Problème 493)

1.1 Enoncé

70 coloured balls are placed in an urn, 10 for each of the seven rainbow colours.

What is the expected number of distinct colours in 20 randomly picked balls?

Give your answer with nine digits after the decimal point (a.bcdefghij).

Figure 1 - Enoncé problème 493

Difficulté du problème : 10 %

1.2 Approche mathématique

On a comme données pour le problème :

- 7 couleurs différentes et 10 balles pour chaque couleur
- 70 boules au total
- On effectue 20 tirages

Pour répondre à ce problème on a décidé de poser une autre question : Quelle est la probabilité de ne tirer aucune balle d'une certaine couleur à la fin des tirages ?
Pour ne pas tirer de balle de la couleur **Y** on ne doit donc pas tirer de balle de la couleur **Y** sur les 20 tirages, or il y a 10 balles de cette couleur, ce qui nous laissent 60 balles possible. On obtient ainsi :

Au numérateur : 20 C 60

Et au dénominateur : 20 C 70 le nombre de tirages total possible.

Ce résultat est la probabilité de ne tirer aucune balle de la couleur **Y** à la fin des 20 tirages.

Donc pour connaître la probabilité d'avoir tiré au moins une balle de la couleur **Y**, il suffit de prendre l'inverse de ce résultat.

1 - (20 C 60) / (20 C 70)

Et enfin pour répondre à la question du problème, il suffit de prendre en compte toutes les couleurs, c'est à dire multiplier le résultat précédent par le nombre de couleurs.

7 x (1 - (20 C 60) / (20 C 70))

1.3 Cas général

Maintenant si l'on s'intéresse au cas général :

On doit connaître le nombre de balles par couleur, le nombre de couleurs et le nombre de tirages.

On peut ainsi savoir le nombre de balles totales.
On doit ensuite calculer le nombre de balles sans les balles de la couleur **Y** (Total de balles – Balle de la couleur **X**), on note le résultat **V**.

On obtient comme formule générale :
Nombre de couleurs x (1 - (Nombre de tirages C **V**) / (Nombre de tirages C Total de balles)

Attention cette formule marche uniquement si le nombre de balle est le même pour chaque couleur.

1.4 Approche informatique

L'ensemble du problème a été codé en C#.
Le calcul de combinaisons a été simplifié afin de ne calculer que les factorielles nécessaire dans le cas demandé. On définit alors :
T le nombre de tirages
V le nombre de balles sans la couleur **Y**
X le nombre de balles total

(T C V) / (T C X) = (T ! / (V ! * (T – V) !)) / (T ! / (X ! * (T – X) !))

Par simplification, on obtient :

(T C V) / (T C X) = (X ! * (T – X) !)) / (V ! * (T – V) !))

On a ainsi 4 factoriels au lieu de 6.

Afin d'éviter des duplications de code, le calcul de factoriels a été mis sous forme de fonction générique.

1.5 Complexité

La complexité de la fonction factorielle est de O(n), cette fonction est appelée 4 fois donc la complexité est de l'ordre de O(4n) soit une complexité linéaire.

1.6 Résultat

Pour conclure ce problème on obtient comme résultat **6.818741802** en **1 milliseconde**.

II/Squarefree Binomial Coefficients (Problème 203)

2.1 Enoncé

The binomial coefficients ${}^{n}C_{k}$ can be arranged in triangular form, Pascal's triangle, like this:

```
                    1
                1       1
            1       2       1
        1       3       3       1
    1       4       6       4       1
1       5       10      10      5       1
1   6       15      20      15      6   1
1   7   21      35      35      21  7   1
.........
```

It can be seen that the first eight rows of Pascal's triangle contain twelve distinct numbers: 1, 2, 3, 4, 5, 6, 7, 10, 15, 20, 21 and 35.

A positive integer n is called squarefree if no square of a prime divides n. Of the twelve distinct numbers in the first eight rows of Pascal's triangle, all except 4 and 20 are squarefree. The sum of the distinct squarefree numbers in the first eight rows is 105.

Find the sum of the distinct squarefree numbers in the first 51 rows of Pascal's triangle.

Figure 2- Enoncé problème 203

2.2 Approche algorithmique

La construction d'un triangle peut se faire en calculant chaque position n C k ou également peut se faire en utilisant la ligne précédente.

La deuxième méthode sera utilisée pour construire le triangle de Pascal car la méthode avec les combinaisons est plus complexe, de plus on ne calculera que la moitié du triangle de Pascal car celui-ci est symétrique et on s'intéresse uniquement au nombre présent et pas à leur quantité.

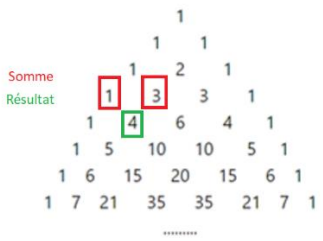


Figure 3 - Méthode 2 de construction du triangle de Pascal

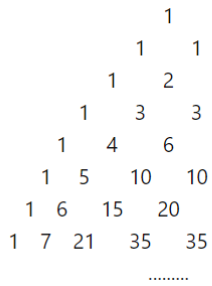


Figure 4 - Triangle de Pascal construit pour l'algorithme

Pour chaque nombre généré, on testera si celui-ci est un entier sans facteur carré (squarefree).

Afin de tester si le nombre est un entier sans facteur carré, on ne testera que le carré des nombres premiers car un nombre est une composition de nombre premiers par conséquent le carré d'un nombre est forcément composé de nombres premiers au carrés.

Par exemple : 35 est égal à 5*7 par conséquent $35^2 = 1225 = 5*7 * 5 * 7 = 5^2 * 7^2$

Si le nombre est un entier sans facteur carré alors on le met de côté.

En accord avec la définition de Wikipédia, un nombre premier peut être considéré comme tel s'il n'existe pas un entier plus grand que 1 et plus petit ou égal à la racine de ce nombre dont le résultat de la division est égal à un entier. (En effet, si n = a*b est une composition (avec a et b =/= 1) alors un des facteurs est forcément au plus la racine carré de n).

On effectuera la somme de l'ensemble des nombres mis de côté.

2.3 Approche informatique

Le programme sera réalisé à l'aide du langage C#.

Afin d'obtenir des temps plus rapides lors de l'insertion ainsi que la possibilité d'ignorer les doublons, on utilisera ici une **HashSet.** Cette structure de données générique ne stocke que des éléments d'un seul type et par conséquent permet de générer la mémoire afin de permettre des allocations dynamiques et plus optimisés (car correspondante au type). Lors de l'insertion la **HashSet** génère un code appelé hash qui est un code obtenu par découpage de l'ensemble des données de l'élément à insérer. Cet identifiant permet à la fois de récupérer plus rapidement la valeur mais également d'éviter l'insertion de doublon dans la structure. En effet, lors de l'insertion celle-ci cherche tout d'abord si l'élément est déjà présent sinon elle insère celle-ci.

Afin d'éviter de générer à chaque fois l'ensemble des nombres premiers nécessaire pour trouver un nombre « squarefree ». On décidera au préalable de générer l'ensemble des nombres premiers nécessaires, pour cela on va calculer le coefficient central de la dernière ligne du triangle de Pascal et on va faire la racine carrée de ce nombre (car le carré des nombres premiers doit être au maximum égal au nombre dans le triangle de Pascal). L'ensemble de ces nombres sera stocké dans une **ArrayList**.

Le type utilisé pour stocker les nombres du triangle sera des **long** car le coefficient le plus grand pour un triangle de Pascal avec 51 lignes est de $20C51$ soit 77 535 155 627 160 car la taille limite d'un int est de 2 147 483 647 alors qu'un **long** permet de stocker jusqu'à 9 223 372 036 854 775 807.

2.4 Complexité

La complexité du triangle de Pascal est de : O (N(N-1) / 2)

La complexité du calcul permettant de savoir si un nombre est premier : O ($\sqrt{N}/2$)

La complexité du calcul permettant de savoir si un nombre est un entier sans facteur carré : O ($\sqrt{N}$)

Le temps de création des nombres premiers est de 10,368 s.

Le temps de calcul afin de trouver la somme des entiers sans facteur carré est de 130 ms.

2.5 Résultat

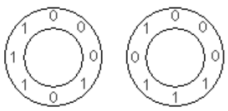
Le résultat obtenu lors des calculs est de **34029210557338** en **130 ms**.

III/Binary circle (Problème 265)

3.1 Enoncé

2^N binary digits can be placed in a circle so that all the N-digit clockwise subsequences are distinct.

For N=3, two such circular arrangements are possible, ignoring rotations:



For the first arrangement, the 3-digit subsequences, in clockwise order, are:
000, 001, 010, 101, 011, 111, 110 and 100.

Each circular arrangement can be encoded as a number by concatenating the binary digits starting with the subsequence of all zeros as the most significant bits and proceeding clockwise. The two arrangements for N=3 are thus represented as 23 and 29:

$$00010111_2 = 23$$
$$00011101_2 = 29$$

Calling S(N) the sum of the unique numeric representations, we can see that S(3) = 23 + 29 = 52.

Find S(5).

Figure 5 - Enoncé problème 265

Difficulté du problème : 40 %

Dans le cas de cet exercice, un cercle binaire est un cercle qui se compose de 2^N nombres binaire. Un cercle est considéré unique par les valeurs qu'il contient et leurs ordres mais cependant une rotation de l'ensemble du disque ne permet pas de créer un disque différent.

<u>Par exemple :</u>

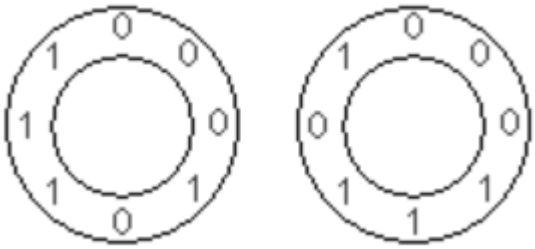


Figure 6 - Deux cercles binaires

Les disques ci-dessus sont deux disques différents.

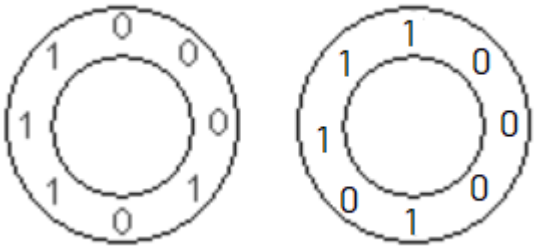


Figure 7- Un cercle binaire et sa rotation

Les disques ci-dessus représentent le même disque.

Afin de savoir quelle « version » du disque est correcte, on choisira de prendre la valeur plus petite en binaire que peut prendre le disque.

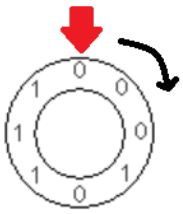


Figure 8 - Sens de lecture d'un cercle

Afin de lire la valeur d'un disque, on lira la série de 0 et de 1 à partir de la flèche rouge (ici « 00010111 ») et on traduira la suite sous forme de nombre en base 10 (ici 23) et on prendra la valeur la plus petite parmi les différentes rotations.

Pour qu'un cercle binaire de taille N soit valide il doit permettre de recomposer l'ensemble des nombres binaires avec une taille de N bits (l'ensemble des valeurs allant de 0 à (2^N) - 1).

Prenons exemple avec le cercle précédent, celui-ci est un cercle de taille 3.

Il y a 2^N nombres possibles sachant que chaque nombre est composé de N bits (0 ou 1), par conséquent la suite recherchée aura une taille de 2^N * N (dans notre cas 2^3 * 3 = 24).

Pour obtenir une suite de cette taille, on répétera la valeur binaire du cercle N fois.

La suite du cercle ci-dessus est 00010111 00010111 00010111.

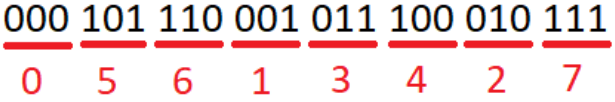


Figure 9 - La décomposition d'un cercle binaire

Donc ce cercle binaire est validé.

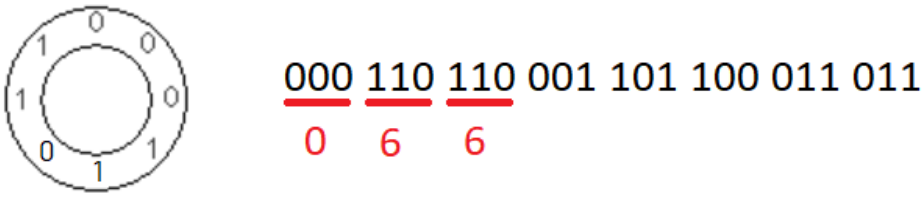


Figure 10 - La décomposition d'un cercle non valide

Par conséquent, ce cercle n'est pas validé.

Lorsque l'ensemble des cercles binaires de taille N validés sont trouvés alors on convertit la valeur du cercle binaire sous forme de nombre en base 10 et on effectue la somme de l'ensemble des valeurs, le résultat trouvé prendra la forme de S(N).

Le but du problème est de trouver le résultat de S (5).

3.2 Approche algorithmique

La méthode « Force Brute » :

Le but de la méthode est de générer l'ensemble des disques possibles afin de calculer S (N).

Un cercle est composé de 2^N chiffres binaires, par conséquent la valeur du disque peut aller de 0 à $2^{2^N} - 1$ soit 2^{2^N} possibilités.

Pour chaque possibilité, on va créer une chaine composée de la suite répétée **n** fois. Lorsque cette chaine est construite, on va lire la chaine N caractères par N caractères.

Pour chaque lecture dans la chaine, on va récupérer la valeur puis la convertir en base 10.

On utilisera un tableau de taille 2^N pour gérer la lecture des valeurs, pour chaque case la valeur initiale est de 0. Si la valeur est rencontrée une première fois alors la valeur dans le tableau passe de 0 à 1. Cependant, si la valeur est rencontrée une deuxième fois alors le cercle n'est pas validé et on passe au cercle suivant.

Si toutes les valeurs sont passés à 1 alors le disque est validé et mis de côté.

Lorsque toutes les possibilités ont été testées, alors on retire les doublons en effectuant les rotations de bits sur la suite et en supprimant les valeurs égales à cette rotation.

On convertit l'ensemble de valeurs en base 10 et on effectue la somme des valeurs obtenues.

3.3 Approche informatique

L'ensemble du problème sera traité dans le langage **C#** (celui-ci sera mis en avant dans les optimisations).

À la suite des contraintes algorithmiques, on utilisera les différentes structures de données afin de transcrire de la manière la plus optimale l'algorithme sous forme de programme informatique :

- On utilisera des doubles afin de traiter la valeur des différents disques

En effet un **int** (nombre entier) ne permet d'aller au maximum que jusqu'à la valeur 2,147,483,647 or 2^{2^5} = 4 294 967 296 on a donc utilisé un **double** dont la capacité peut aller à $\pm 1{,}7 \times 10^{308}$

- On utilisera des chaines de caractères pour représenter l'ensemble des nombres binaires
- On utiliser un tableau classique pour la lecture des données
- On utilisera des **ArrayList** afin de stocker l'ensemble des disques valides

Une **ArrayList** est une structure propre au **C#** (également présente en Java) qui permet de stocker un ensemble d'objets de manière dynamique (la gestion de la mémoire est faite par la structure). Elle permet une gestion de l'ensemble des éléments plus développés et plus pratique que l'usage d'un tableau.

Par soucis de sécurité dans le langage, on utilisera des listes intermédiaires afin de traiter la suppression des doublons. En effet il est impossible de supprimer des éléments d'une liste lors de son parcours, la suppression pouvant amener à des problèmes d'indices par exemple.

En programmation orientée objet lors de l'appel d'une fonction, il existe deux types de passages de données en paramètres : le passage par copie et le passage par référence. Le premier (passage par copie) est le plus classique, utilisé par défaut il est totalement transparent pour le programmeur et permet de créer une copie dans la fonction afin de pouvoir modifier la valeur de cette copie uniquement dans le contexte de la fonction. A l'opposée, le passage par référence utilise la référence (adresse de l'objet) et par conséquent toute modification de la valeur sera prise en compte sans tenir compte du contexte. Pour cela, nous utiliserons le mot clé **ref** en **C#** qui permettra le passage par référence.

3.4 Optimisations

Dans un premier temps, on a pu remarquer une totale symétrie dans l'ensemble des caractères qui composent les différents nombres l'ensemble des valeurs allant de 0 à $2^N - 1$. On en déduit qu'il y a autant de 0 que 1 donc on ne cherche que dans les disques composés de $\frac{(2^N)}{2}$ uns.

Après un temps de calcul bien trop élevé pour calculer S (5) nous avons décidé d'optimiser l'algorithme afin de réduire le temps d'exécution de celui-ci, dans un premier temps nous avons décidé d'observer les différents résultats obtenus à l'aide des calculs effectués précédemment.

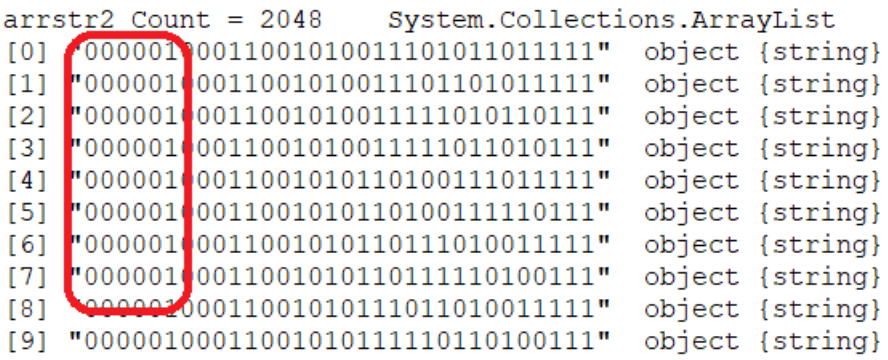


Figure 11 - Extrait des résultats de S (5)

On peut remarquer que l'ensemble des disques obtenus commencent **n** zéros par conséquent on a pu limiter les différentes possibilités allant de $(2^{(2^N)-N-1})$ à $\left(2^{(2^N)-N}\right)$ à la place de 2^{2^N} possibilités. Le temps de calcul passe alors de 36 heures à 1 heure 50 mins.

Un dernier cas d'optimisation est la gestion des 1 dans la suite pour trouver le plus rapidement les disques validés, il y a deux méthodes possibles :

- La méthode « Force Brute » en filtrant les cas ne contenant pas le bon nombre de 1
- Mettre en place l'ensemble des combinaisons $\frac{2^{2^N}}{2}C2^{2^N}$ de 1 dans la suite

Malgré un plus faible nombre d'opérations en utilisant la méthode combinaisons, la méthode « Force brute » est beaucoup plus linéaire et permet d'être découpée afin d'être parallélisée à l'aide de **Thread,** l'utilisation de ces threads est intégrée nativement dans le langage **C#**.

Actuellement, un PC est composé d'un processeur quad-core (voir octo-core) ce qui signifie que celui-ci possède 4 cœurs physiques (réciproquement 8 cœurs) par conséquent il possède 8 threads (ou 16 réciproquement) afin d'effectuer les différents calculs/opérations nécessaires pour le fonctionnement des applications.

Le but de cette opération est de paralléliser la recherche des différents disques cependant cette opération ne peut pas être découpée en plus d'étapes parallèles que de threads possédés par le processeur sinon l'efficacité de la mise en parallèle est diminuée.

Temps avec un thread = 110 mins

Temps avec 2 threads = 56 mins

Temps avec 6 threads = 35 secondes

Temps avec 8 threads = 105 mins

3.5 Complexité

Pour la méthode « Force brute » :

Recherche pour chaque cercle si celui-ci est valide : O (2^{2^N})

- Construction de la chaine : O (n)
- Recherche de nombre en double dans la chaine : O (2^N)

Suppression des différentes rotations : O (m)

Total de la complexité : O (2^{2^N} * (n + 2^N) + m)

Pour la méthode optimisée :

Recherche pour chaque cercle si celui-ci est valide : O (($2^{(2^N)-N}$) − ($2^{(2^N)-N-1}$))

- Construction de la chaine : O (n)
- Recherche de nombre en double dans la chaine : O (2^N)

Total de la complexité : O ((($2^{(2^N)-N}$) − ($2^{(2^N)-N-1}$)) * (n + 2^N))

3.6 Résultat

Le résultat obtenu est de **209110240768** en **36 heures** pour la version initiale et de **35 secondes** pour la version optimisée.

IV/ Exploring the number of different ways a number can be expressed as a sum of powers of 2 (Problème 169)

4.1 Enoncé

Define f(0)=1 and f(n) to be the number of different ways n can be expressed as a sum of integer powers of 2 using each power no more than twice.

For example, f(10)=5 since there are five different ways to express 10:

1 + 1 + 8
1 + 1 + 4 + 4
1 + 1 + 2 + 2 + 4
2 + 4 + 4
2 + 8

What is f(10^{25})?

Figure 12 - Enoncé problème 169

Difficulté du problème : 50 %

4.2 Approche mathématique

On a deux conditions :
- Base 2
- Utiliser au maximum 2 fois le même chiffre pour une addition

Pour ce problème on a décidé de passer par le binaire afin de pouvoir plus facilement décomposer les nombres en base 2. On peut ainsi plus facilement décomposer les additions :

Prenons un exemple avec 8 :

Base 10 :	Binaire :
8	1000
4 + 4	0100 + 0100
4 + 2 + 2	0100 + 0010 + 0010
4 + 2 + 1 + 1	0100 + 0010 + 0001 + 0001

On peut remarquer entre la première ligne et la seconde ligne qu'il suffit de décaler le 1 et on obtient ainsi une autre décomposition.

Mais si on prend 7 (111) :

Base 10 :	Binaire :
4+2+1	100 + 10 + 1

Il n'y a qu'une seule décomposition possible pour ce nombre car on ne peut pas séparer celui-ci en plusieurs décompositions sinon on ne respecte pas la deuxième condition, dans le cas contraire on obtiendrait un chiffre en triple.

Avec ce qu'on a obtenu on a un début d'algorithme, on sépare les « 1 » tant qu'il y a des « 0 » et on s'arrête au prochain « 1 ».

Prenons maintenant 10 qui possède plusieurs 1 en binaire, 1010 :

Base 10 :	Binaire :
8 + 2	1000 + 10
8 + 1 +1	1000 + 1 + 1
4 + 4 + 2	100 + 100 + 10
4 + 4 + 1 + 1	100 + 100 + 1 + 1
4 + 2 + 2 + 1 + 1	100 + 10 + 10 + 1 + 1

Là on peut voir qu'il y a plusieurs décompositions à faire, maintenant la question c'est de savoir par où commencer la décomposition. C'est à dire par 8 (1000) ou par 2 (0010).

En réalité les deux sont possible mais il est plus simple pour notre approche de commencer par le plus petit et compter le nombre de possibilités de décompositions possibles, dans un second temps il faut compter le nombre de décomposition possible pour le chiffre suivant sans oublier de respecter la deuxième condition.

En effet si on reprend 10 :

On peut voir que « 2 » possède deux décompositions : 2 et 1+1

On peut voir que « 8 » possède quatre décompositions : 8 ; 4+4 ; 4+2+2 et 4+2+1+1

Mais les deux dernières décompositions de « 8 » posent un problème avec la deuxième condition :

L'avant dernière décomposition de « 8 » n'est pas compatible avec la première décomposition de « 2 » (4+2+2+2 n'est pas valable) mais l'autre décomposition de « 2 » ne pose pas ce problème, on a alors 4+2+2+1+1. Pour ce qui est de la dernière décomposition de « 8 » elle ne nous apporte aucun cas valable, en effet elle est soit redondante (4+2+2+1+1) soit incompatible (4+2+1+1+1+1). On remarque également que les autres décompositions de « 8 » ne posent pas de problème et sont donc chacune réalisable avec toutes les décompositions de « 2 », c'est-à-dire deux fois.

Au vu des résultats observés sur plusieurs exemples plus complexes on admet que le nombre de possibilités pour le cas particulier (c'est à dire le nombre de décomposition du dernier cas) est équivalent au nombre des décompositions possibles du précédent (x_n) moins le nombre de décompositions du précédent précédent (x_{n-1}).

Illustration avec un exemple : 164 en décimal

10 100 100 en binaire

On va appliquer ce qui a été vu précédemment.

Cela va se faire en trois tours (le nombre de « 1 ») :

- 1er tour :

On commence par lire le nombre de droite à gauche jusqu'au premier « 1 ». On lit ainsi 100 en binaire, on a donc 3 décompositions possibles : 100

010+010

010+001+001

(x_1 (nombre de décompositions possibles) = 3)

- 2ème tour :

On recommence en allant au second « 1 ». On lit 100 à nouveau et on a donc 3 décompositions possibles chacune compatible avec toutes les décompositions de 100 vues précédemment. Cependant il nous reste le « cas particulier », si l'on rajoute un 0 à 100 donnant 1000 une autre décomposition devient possible : 0100+0010+0001+0001, compatible avec deux (par ailleurs on a bien $x_1 – x_0 = 2$) décompositions de 100 qui sont 010+010 et 010+001+001.

Rajouter davantage de 0 est inutile, on obtient alors des décompositions redondantes ou incompatibles.

Au total on a 3 * 3+2 décompositions possible c'est à dire 11

($x_2 = 11$)

- 3ème tour :

On recommence une dernière fois. On obtient 10 donc deux décompositions toutes compatibles avec les 11 précédentes. On passe au « cas particulier », on rajoute donc à nouveau un 0 pour obtenir 100 et donc une autre décomposition devient possible, compatible avec, selon la formule vu précédemment $x_2 – x_1 = 11 – 3 = 8$ décompositions possibles.

On a ainsi 11 * 2 + 8 = 30 décompositions possibles pour 10 100 100.

C'est à dire 30 manières possibles d'écrire 164 selon les critères du problème.

Autre exemple : 183 en décimal
Ce qui donne en binaire 10110111

- 1^{er} tour :

On trouve 1 en binaire qui possède 1 décomposition.

- 2eme tour :

On trouve 1 en binaire qui possède 1 décomposition, on ne peut la décomposer sinon elle n'est pas compatible avec aucune décomposition précédente.

On a 1 x 1 =1 donc 1 chemin au total.

- 3eme tour :

On trouve 1 en binaire qui possède 1 décomposition.

On a 1 x 1 =1 donc 1 chemin au total.

- 4eme tour :

On trouve 10 en binaire qui possède 2 décompositions.

On a 1 x 1 + 1 x 1 = 2 donc 2 chemins au total.

- 5eme tour :

On trouve 1 en binaire qui possède 1 décomposition mais on peut le considérer comme 10 car il possède une décomposition compatible avec une décomposition précédente, on peut la décomposer en « 01 + 01 »

On a 1 x 2 + (1 x 2 - 1) = 3 donc 3 chemins au total.

- 6eme tour :

On trouve 10 en binaire qui possède 2 décompositions mais on peut le considérer comme 100 car il possède une décomposition compatible avec une décomposition précédente, on peut la décomposer en « 010 + 001 + 001 »

On a 2 x 3 + (1 x 3 -2) = 7 donc 7 chemins au total.

4.3 Algorithme

Dans un premier temps on transforme un entier positif en binaire.

On passe à la partie où l'on traite le problème, on commence par le plus petit bit :

Tant que l'on a un 0

On incrémente un compteur

Quand on a un 1

On incrémente la variable '**somme**' qui est la somme du nombre d'expression possible

On remet le compteur à 1 et on stocke la valeur du précédent précédent **ppsomme** qui est le cas particulier vu dans le dernier exemple (la décomposition est compatible avec certaines décompositions des précédents mais pas toutes) et le nombre de décompositions avec lesquelles elle est compatible est **somme** moins **ppsomme**.

On fait cela jusqu'à ce qu'on ait traité tout le nombre binaire. Et on retourne la variable **somme**.

4.4 Approche informatique

Dans un premier temps, l'algorithme a été traduit sous forme de programme en langage C. Cependant, le plus grand type en C étant le **long long** (il en existe d'autres plus grand mais qui ne sont pas compatibles avec des compilateurs et ne peuvent pas effectuer l'ensemble des opérations de bases). Celui-ci à une capacité de 9,223,372,036,854,775,807 soit bien inférieure au nombre recherché qui est 10^{25}.

Dans un second temps, nous avons abordé le problème en C# en utilisant les **BigInteger** ce type est inclus dans une bibliothèque du **C#** (System.Numerics) et permet manipuler des entiers sans aucune limite de taille et gère automatiquement l'allocation de manière dynamique.

L'inconvénient de l'utilisation de ce type de données est que celui-ci comporte très peu de méthode de base par conséquent la conversion en binaire n'est pas implémentée et on a réalisé une fonction qui permet de convertir un **BigInteger** en un tableau contenant la valeur du nombre en binaire.

On a également utilisé le mot clé **ref** (cf. 3.3) afin d'effectuer un passage par référence de la taille du nombre en binaire.

4.5 Résultat

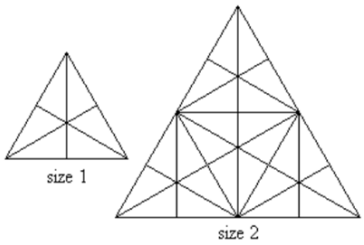
Pour conclure ce problème on obtient comme résultat **178653872807** en **11 millisecondes**.

V/Cross-hatched triangles (Problème 163)

5.1 Énoncé

Difficulté : 70 %

Consider an equilateral triangle in which straight lines are drawn from each vertex to the middle of the opposite side, such as in the *size 1* triangle in the sketch below.



Sixteen triangles of either different shape or size or orientation or location can now be observed in that triangle. Using *size 1* triangles as building blocks, larger triangles can be formed, such as the *size 2* triangle in the above sketch. One-hundred and four triangles of either different shape or size or orientation or location can now be observed in that *size 2* triangle.

It can be observed that the *size 2* triangle contains 4 *size 1* triangle building blocks. A *size 3* triangle would contain 9 *size 1* triangle building blocks and a *size n* triangle would thus contain n^2 *size 1* triangle building blocks.

If we denote T(*n*) as the number of triangles present in a triangle of *size n*, then

T(1) = 16
T(2) = 104

Find T(36).

Figure 13 - Enoncé problème 163

Pour ce problème nous n'avons pas réussi à aboutir aux résultats.

5.2 Pistes explorées

Nous avons commencé par repérer les différents triangles pour T (1).

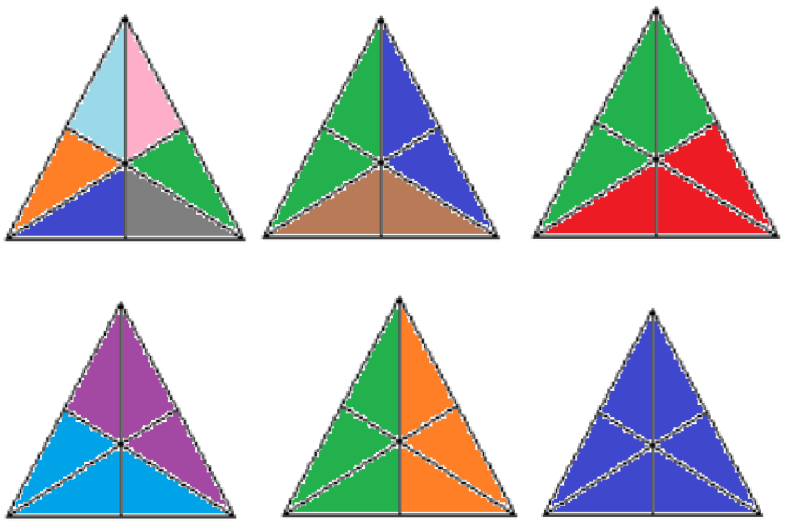


Figure 14 - Ensembles des triangles pour T (1)

On retrouve bien les 16 triangles indiqués dans l'énoncé.

On s'est ensuite intéressé au T (2). On remarque qu'il y a 4 triangles T (1) à l'intérieur et également que chacun des triangles de T (1) sont répétés dans ces 4 triangles comme le montre les images ci-dessous :

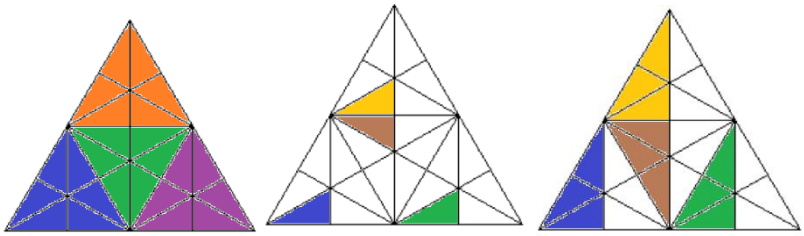


Figure 15 - Différents triangles T (1) qui compose un triangle T (2)

On a ensuite compté les triangles de T (2) mais ceux de taille 2, c'est-à-dire des triangles qu'on ne retrouve pas dans T (1) et qui utilise au moins deux triangles T (1) différents.

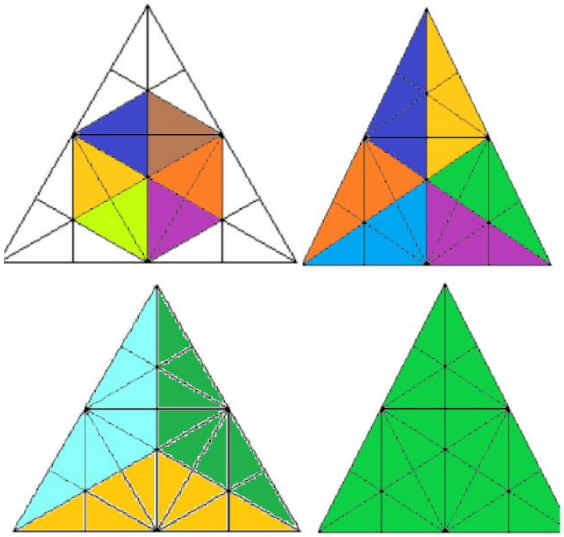


Figure 16- Différents triangles composés de deux triangles T (1)

Les triangles qui suivent sont représentés que deux fois mais en réalité il en existe six de chaque (il suffit de prend la figure dans un autre sens.

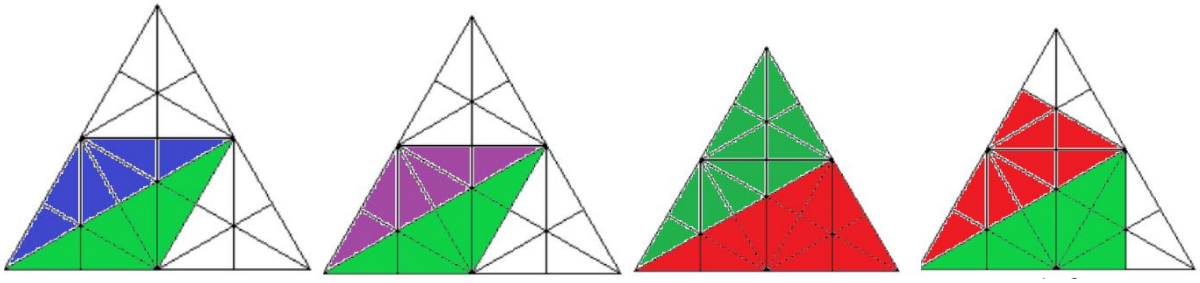


Figure 17 - Différents triangles T (2)

On trouve 40 triangles de taille 2 pour T (2).
Si on compte les triangles de taille 1 pour T (2) : 16 * 4 = 64
Au total on trouve 64 + 40 = 104 on retrouve bien les triangles indiqués dans la consigne.

On a ensuite regardé le triangle de T (3) :

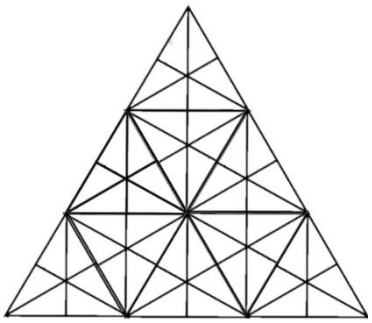


Figure 18 - Triangle de taille T (3)

On remarque qu'il y a 9 triangles de T (1) et 3 triangles de T (2).
On a ensuite compté le nombre de triangle de taille 3 dans T (3) et on trouve 39.
Au total on a 9*16 + 3 * 40 + 39 = 303 triangles pour T (3)

On commence à trouver un début de formule pour calculer le nombre de triangle :
n^2*16 + le nombre de triangle T(2) * 40 + le nombre de triangle T(3) * 39 + … + le nombre de triangle T(n) * X

On arrive maintenant au problème de déterminer « X » qui est le nombre de triangles formés à la taille n.
C'est cette étape qui nous pose un problème.

On a remarqué que certains triangles reviennent selon la taille T et que certains apparaissent ou disparaissent à partir d'une certaine taille.

Ces types de triangles apparaissent peu importe la taille :

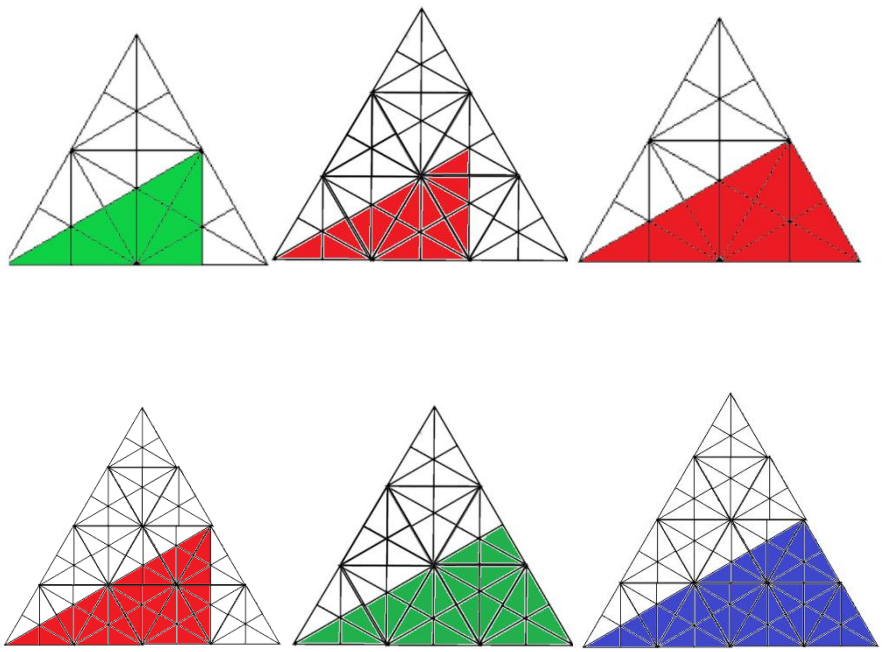


Figure 19 - Triangles indépendants de la taille T

Ces triangles n'apparaissent que pour certaines tailles :

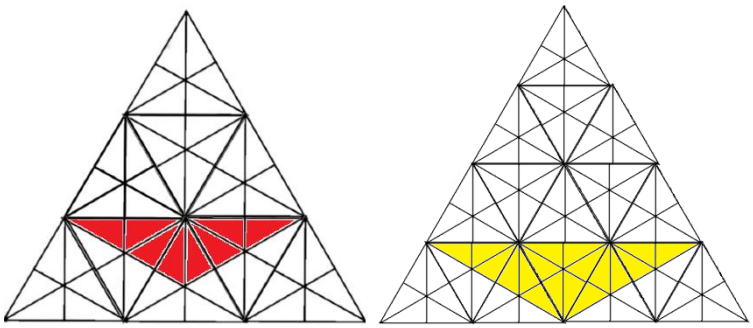


Figure 20 - Triangles existants seulement pour certaines tailles T

Cette forme de triangle n’apparaît pas à la taille 5 mais réapparaît à certaine taille.

D’autres cas à prendre en compte sont les triangles à l’envers supérieur à T (1) :

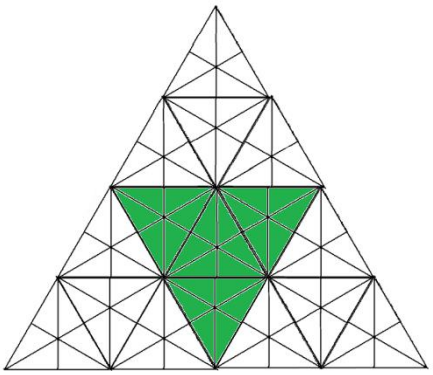


Figure 21 - Triangle à l'envers de taille T (2)

On peut assez facilement les compter, le problème est les triangles qu’il génère car certains sont déjà pris en compte dans le calcul et d’autres non.

Exemple :

Ce triangle n’est pas pris en compte dans le calcul précédent pour calculer T (4)

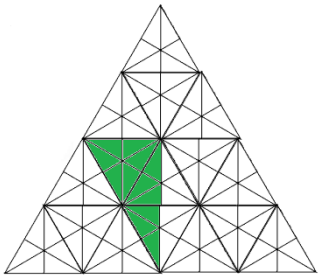


Figure 22 - Triangle à l’envers ignoré lors du calcul de T (4)

Alors que ce triangle est pris en compte pour le calcul de T (3)

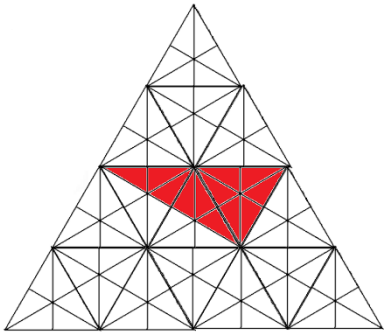


Figure 23 - Triangle à l'envers dans T (3)

Nous n'avons pas réussi à faire l'inventaire de tous les cas possible, ce qui nous empêche de déterminer X.

<u>Une autre piste explorée :</u>

Nous avons également essayé d'explorer d'autres pistes mais qui ont moins abouti que celle expliquée précédemment.

En effet on a constaté que l'on peut déterminer le nombre de ligne suivant T(n) :

On trouve **9n-6**, on pensait ainsi pouvoir déterminer le nombre de triangles générés à partir du nombre d'intersections mais cela n'a pas été concluant car nous n'avons pas trouvé de récurrence dans cette voie.