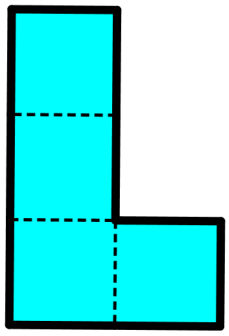
Collège Don Bosco, Woluwe-Saint-Lambert
Année scolaire 2016-2017

Article scientifique MATh.en.JEANS

N carrés



Lucas Prieels, Sam Vanhamme

Enseignantes : Mmes Céline Serta et Ingrid Demulder-t'Kindt

Chercheurs : Mme Émilie Clette (chercheuse à l'ULg) et Dr Yvik Swan
(chargé de cours à l'ULg et à l'ULB)

Table des matières

Table des figures

1 Introduction

1.1 Problème

Le problème que nous avons choisi est le suivant : « Combien peut-on former d'objets avec n carrés ? ». Certaines des notions utilisées comme le symbole-somme sigma ($\sum$) ou la récursivité seront expliquées en annexe pour ne pas atténuer la lisibilité des preuves. Les listes complètes de formes seront aussi proposées dans les annexes.

1.2 Conventions

Nous avons adopté quelques conventions que nous allons expliciter ici : premièrement, assez logiquement, $n \in \mathbb{N}^*$ [1]. En effet, nous ne pouvons pas créer d'objets avec un nombre non-naturel de carrés. Deuxièmement, tous les carrés composant les objets seront de même taille. Troisièmement, l'objet ainsi obtenu sera en un seul tenant. Quatrièmement, les carrés devront se toucher par un côté entier. Ensuite, nous manipulerons les objets ainsi définis avec des rotations et des symétries. Nous ne considérerons que des rotations d'amplitudes multiples de 90°.
Toutes les figures sont faites avec le logicel libre GeoGebra.

2 Dénombrement

2.1 Rotations et symétrie

Nous pouvons nous demander si deux formes équivalentes [2] par une rotation et/ou une symétrie sont considérées comme identiques ou non.
Par exemple, les deux formes suivantes sont-elles identiques ?

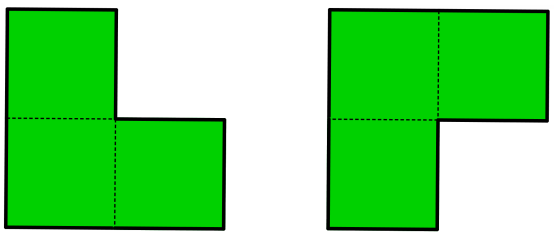


FIGURE 1 – Deux des formes de taille 3

Nous avons répondu à cette question en nous disant que la meilleure manière d'aborder le dénombrement de ces formes est de traiter les deux cas possibles : celui on nous faisons intervenir les rotations et les symétries et celui ou nous ne le faisons pas.

1. $\mathbb{N}^*$ et $\mathbb{N}_0$ sont deux écritures équivalentes pour représenter l'ensemble des entiers naturels non nuls.

2. Deux formes seront considérées comme "équivalentes" si appliquer une rotation et/ou une symétrie sur la première donne la deuxième.

2.2 À la main

Ainsi, nous avons commencé par essayer toutes les configurations de carrés une à une à la main pour calculer les premiers nombres de deux suites, : une suite A avec les formes basiques, et une suite B avec les rotations et symétries. Notons A_n le $n$ème élément de la suite A et B_n le $n$ème élément de la suite B.

Voici les premiers nombres formant les suites A et B.

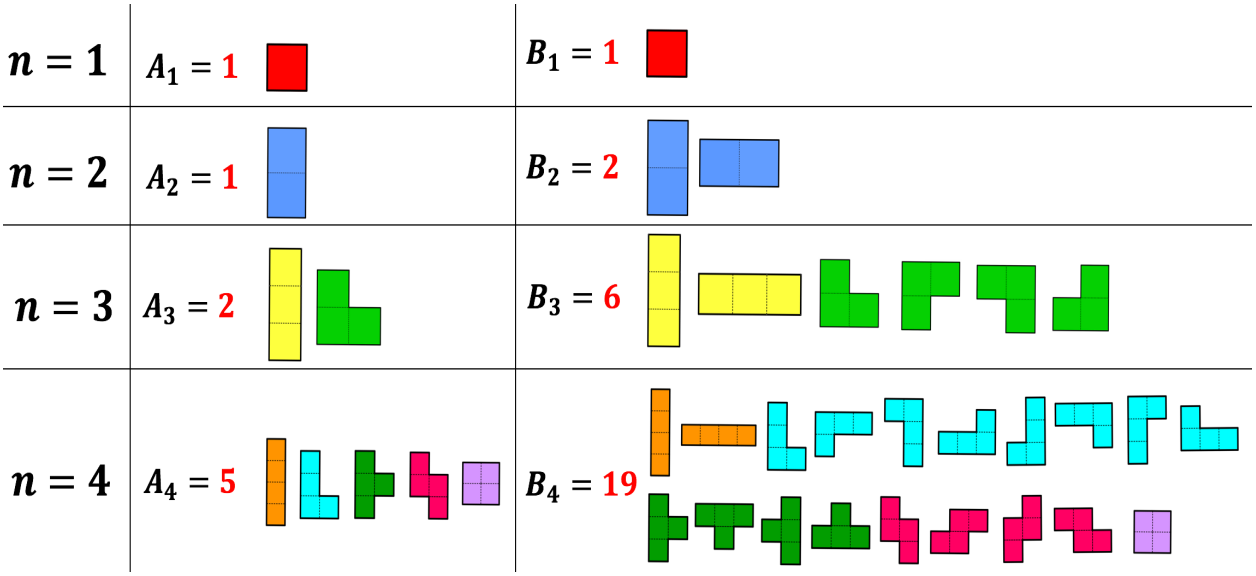


FIGURE 2 – Liste des premières formes

Nous espérions trouver une suite arithmétique, géométrique ou périodique, mais ce ne fut pas le cas.

2.3 À l'ordinateur

Voyant que nous ne pouvions pas continuer à la main indéfiniment car trop fastidieux, nous avons décidé d'écrire un programme informatique de près de 700 lignes en C++ disponible à l'adresse suivante : `https://github.com/Lucas31415/MeJ/blob/master/main.cpp`

Cela nous a aidé à calculer plus de nombres pour les deux suites :

Suite A : 1, 1, 2, 5, 12, 35, 108, 369

Suite B : 1, 2, 6, 19, 63, 216, 760, 2725

2.4 Algorithme

Un algorithme est une suite d'instructions qui sont, dans notre cas, réalisées par un ordinateur.

L'algorithme que nous avons écrit est relativement simple. Il faut donner à l'ordinateur en *input* toutes les formes de taille n. Le programme va alors regarder tous les endroits où il pourrait ajouter un carré de façon à ce que la forme obtenue soit en un seul tenant. Il va à chaque fois stocker en mémoire cette nouvelle forme. L'ordinateur va ensuite comparer toutes les formes stockées pour retirer celles qui sont en doubles. Dans le cas de la suite A, il va faire une étape supplémentaire qui consiste à comparer également les formes équivalentes par rotations et/ou symétries. Cette étape supplémentaire n'est pas une étape si simple et même pour l'ordinateur, elle est très longue. C'est pourquoi nous avons pu calculer un nombre en plus dans la suite B. Le programme va finalement imprimer en *output* le nombre de formes de taille $n+1$, suivi par toutes ces formes. Nous pourrons ensuite les redonner à l'ordinateur en *input* afin de récupérer les formes de taille $n+2$, etc. Mais notre algorithme ayant une complexité beaucoup trop grande, nous ne sommes pas arrivés à calculer plus de nombres des suites A et B. Nous n'avons malheureusement pas trouvé de formule, ni explicite, ni de récurrence.

2.5 Constatations

2.5.1 Axiome

Notre première constatation a été que, pour tout n, $A_n \leq B_n$. En effet, toutes les formes de A se retrouvent évidemment dans B. Autrement dit, $\forall n \in \mathbb{N}^*, A_n \leq B_n$.

2.5.2 Conjectures

La première conjecture que nous avons trouvée est la suivante :

$$\forall n \in \mathbb{N}^*, n \neq 3, \frac{A_{n+2}}{A_{n+1}} > \frac{A_{n+1}}{A_n}$$

Autrement dit, nous devons multiplier A_n par un nombre de plus en plus grand pour obtenir A_{n+1}. La seule exception, sans que nous sachions pourquoi, est que si $n = 3$, l'inégalité n'est pas vérifiée : $\frac{12}{5} \not> \frac{5}{2}$.

Notre deuxième conjecture est très similaire à la première mais avec B :

$$\forall n \in \mathbb{N}^*, \frac{B_{n+2}}{B_{n+1}} > \frac{B_{n+1}}{B_n}$$

Cela veut dire la même chose que la première, mais pour la suite B. En revanche, il n'y a cette fois pas d'exception.

2.5.3 Résultat

Proposition 1. $\forall n \in \mathbb{N}^*, B_n \leq 8 \times A_n$. Cela signifie que chacune des formes de la suite A peut se retrouver au maximum 8 fois dans la suite B.

Démonstration. Nous remarquons que chaque forme de base de la suite A peut donner 4 images par rotation (de 0°, de 90°, de 180° et de 270°). Nous pouvons aussi prendre une symétrie orthogonale de la forme de A et lui appliquer 4 nouvelles rotations de même amplitudes. Cela nous donne donc $4+4=8$ images différentes d'une seule forme de base, et il n'existe pas d'autres transformations du plan qui donnerait de nouvelles images. Chacune des formes de A peut donc donner au maximum 8 formes dans B. Ce ne sera pourtant pas toujours le cas ! Nous pouvons par exemple citer le cas du carré, dont toutes les images que nous venons de citer sont identiques.

3 Borne inférieure

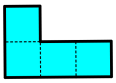
Nous avons cherché ensuite une manière de calculer une estimation de A_n et B_n. Pour cela, nous avons pensé à un concept de "borne inférieure". Le principe est simple : il faut trouver une suite C, qui sera plus facile à calculer, de façon à ce que $\forall n \in \mathbb{N}^*, C_n \leq B_n$

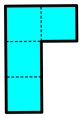
3.1 Décompositions

Nous avons pensé à faire des décompositions de nombre. Définissons C_n comme le nombre de sommes dont le total vaut n à commutativité près, c'est à dire de façon à ce que les nombres forment une suite décroissante. Par exemple, pour le nombre 4, nous pouvons l'écrire sous les formes suivantes :

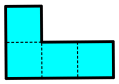
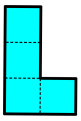
$$4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1$$

Le nombre de sommes dont le total vaut 4 à commutativité près est 5. Autrement dit, $C_4 = 5$

Pour faire le lien avec nos formes composées de carrés, il faut se rendre compte que nous pouvons remplacer chaque terme de la somme par une colonne de carrés. Par exemple, la somme $2+1+1$ peut s'écrire  et la somme $2+2$ peut s'écrire  .

Cela signifie qu'en utilisant ce principe, nous aurons quelques-unes des formes de B_n, mais pas toutes. En effet, nous aurons seulement les formes qui "subissent la gravité" en quelque sorte. Nous n'aurons donc pas de formes de ce type  . Cela signifie que nous obtiendrons dans la suite C certaines de formes de la suite B mais pas toutes. Autrement dit,

$$C_n \leq B_n$$

En revanche, il suffit de citer par exemple les formes obtenues par $2+1+1$  et $3+1$  qui sont images l'une de l'autre pour se convaincre que $C_n \not\leq A_n$.

3.2 Fonction *dec*

Nous avons voulu à nouveau écrire un programme informatique qui calculerait C_n. Pour cela nous avons dû définir la fonction *dec* qui prend deux arguments : la somme des termes et le nombre de termes. Autrement dit $dec(a, b)$ est le nombre de sommes de b termes dont le total est a, à commutativité près.
Par exemple, $dec(7, 4) = 3$ car $7 = 4+1+1+1 = 3+2+1+1 = 2+2+2+1$.

Plus généralement, $dec(a, b)$ est le nombre d'ensembles $\{x_1, x_2, ..., x_b\} \subset \mathbb{N}^*$ tels que $x_1 + x_2 + ... + x_b = a$, ou encore $\sum_{i=1}^{b} x_1 = a$

3.3 Algorithme

Nous avons alors écrit un algorithme récursif [3], dont le principe est expliqué en annexe, assez simple, mais dont la preuve est compliquée.

Proposition 2. $\forall a, b \in \mathbb{N}^*, dec(a, b) = dec(a-1, b-1) + dec(a-b, b)$

Démonstration. Par définition, $dec(a, b)$ est le nombre d'ensembles $\{x_1, x_2, ..., x_b\} \subset \mathbb{N}^*$ tels que $x_1 + x_2 + ... + x_b = a$. Pour tout ensemble répondant à ces conditions, nous avons deux possibilités. Soit $1 \in \{x_1, x_2, ..., x_b\}$, soit $1 \notin \{x_1, x_2, ..., x_b\}$.
Notons $dec_{avec1}(a, b)$ le nombre d'ensembles $\{x_1, x_2, ..., x_b\} \subset \mathbb{N}^*$ tels que $x_1 + x_2 + ... + x_b = a$ et qui contiennent 1. Notons également $dec_{sans1}(a, b)$ le nombre d'ensembles $\{x_1, x_2, ..., x_b\} \subset \mathbb{N}^*$ tels que $x_1 + x_2 + ... + x_b = a$ et qui ne contiennent pas de 1. Nous avons donc $dec(a, b) = dec_{avec1}(a, b) + dec_{sans1}(a, b)$.
Montrons que $dec_{avec1}(a, b) = dec(a-1, b-1)$: S'il y'a un 1 dans la décomposition, nous retirons 1 de la somme. Ainsi, nous avons $x_1+x_2+...+x_{b-1}+x_b-1 = \sum_{i=1}^{b} x_i - 1 = a-1$. Or, puisque nous savons qu'il y a un 1 dans la décomposition et que la suite des termes est décroissante, cela implique que $x_b = 1$. Par conséquent, nous pouvons remplacer le x_b par 1. Cela donne $x_1+x_2+...+x_{b-1}+1-1 = x_1+x_2+...+x_{b-1} = \sum_{i=1}^{b-1} x_i + x_b - 1 = \sum_{i=1}^{b-1} x_i = a-1$.
Nous avons alors une somme de $b-1$ termes qui a un total de $a-1$, autrement dit $dec(a-1, b-1)$.

3. Un algorithme récursif est un algorithme qui utilise une ou plusieurs fonctions récursives. La récursivité sera expliquée plus en détails en annexes.

Montrons que $dec_{sans1}(a,b) = dec(a-b,b)$: dans ce cas-là, retirons 1 à chacun des termes, c'est à dire retirer 1 à b termes, donc b au total. Nous obtenons $x_1 - 1 + x_2 - 1 + ... + x_{b-1} - 1 + x_b - 1 = \sum_{i=1}^{b} x_i - 1 = a - 1 \times b = a - b$. Cela correspond à $dec(a-b,b)$. En résumé, $dec_{avec1}(a,b) = dec(a-1,b-1)$ et $dec_{sans1}(a,b) = dec(a-b,b)$. Comme $dec(a,b) = dec_{avec1} + dec_{sans1}$, $dec(a,b) = dec(a-1,b-1) + dec(a-b,b)$.

En utilisant cette formule récursive, nous avons écrit un programme en C++ disponible à l'adresse suivante : `https://github.com/Lucas31415/MeJ/blob/master/décompositions`.

3.4 Résultat de l'algorithme

Les premiers nombres que nous avons pu calculer grâce à cette fonction récursive sont les suivants :

Suite C : 1, 2, 3, 5, 7, 11, 15, 22, 30, 42

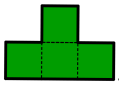
Cette fois, l'algorithme est assez rapide pour calculer beaucoup plus de nombres, 416 plus précisément.

4 Un autre type de décomposition

Nous avons cherché un autre type de décomposition qui se rapprocherait plus de la suite B. Nous avons finalement trouvé une manière plus efficace et plus simple, mais malheureusement après avoir fait notre présentation au congrès de MATh.en.JEANS à Liège. Nous avons quand même décidé de le détailler lors de cet article.

Nous avons en fait simplement décidé de construire une suite D qui utiliserait le même principe que la suite C, mais sans que ce soit à commutativité près. D_n correspond au nombre de sommes différentes dont le total est n. Par exemple $D_4 = 8$ car 8 peut être écrit sous les formes suivantes :

$$4 = 3+1 = 2+2 = 2+1+1 = 1+3 = 1+2+1 = 1+1+2 = 1+1+1+1$$

Il y a donc bien 8 sommes différentes dont le total vaut 4. Ces sommes peuvent être assimilées aux formes de la même manière que C. Par exemple, $1+2+1$ peut s'écrire avec des carrés :  . Nous aurons également une partie des formes, mais pas toutes.

4.1 Dénombrement

Pour dénombrer cette suite D, remarquons déjà que le premier nombre de la décomposition de D_n sera un naturel compris entre 1 et n. Cela signifie que le reste de la décomposition aura comme somme $n-1$ si le premier terme est 1, $n-2$, si c'est 2, ..., 1 si c'est $n-1$ et 0 si c'est n. Donc, si le premier terme est 1, le total du reste de la décomposition devra être $n-1$. Nous aurons ainsi D_{n-1} possibilités. Nous pouvons continuer de la sorte pour les autres premiers termes.

Cela implique que $D_n = D_{n-1} + D_{n-2} + ... + D_2 + D_1 + D_0 = \sum_{i=0}^{n-1} D_i$. Nous pouvons définir $D_0 = 1$ car il n'y a qu'une manière de décomposer 0.

Par conséquent, $D_1 = D_0 = 1$, $D_2 = D_1 + D_0 = 1 + 1 = 2$, $D_3 = D_2 + D_1 + D_0 = 2+1+1 = 4$, ... Nous constatons qu'il s'agit en fait des puissances de 2. Plus précisément, $D_n = 2^{n-1}$. En conclusion,

$$\forall n \in \mathbb{N}^*, B_n \geq 2^{n-1}$$

car, par définition, $B_n \geq D_n$.

Les premiers nombres de la suite D sont ainsi les suivants :

suite D : 1, 2, 4, 8, 16, 32, 64, 128, 256, 512

4.2 Preuve

Finalement, il nous reste à prouver qu'il s'agissait bien de la suite de puissances de 2. Nous avons utilisé pour cela une preuve par récurrence dont le principe est expliqué en annexe.

Initialisation : nous avons déjà trouvé que $D_1 = 1 = 2^0$. Nous n'utilisons pas D_0, car c'est une exception qui a été définie, nous arriverions à des nombres qui ne sont pas entiers.

Hérédité : si D_k est une puissance de 2, prouvons que D_{k+1} est une puissance de 2.

Pour cela, remarquons que $D_{k+1} = \sum_{i=0}^{k} D_i = \sum_{i=0}^{k-1} D_i + D_k$.

Or, par définition, $D_k = \sum_{i=0}^{k-1} D_i$. Donc, $D_{k+1} = D_k + D_k = 2 \times D_k$.

$D_k = 2^{k-1}$ implique que $D_{k+1} = 2 \times 2^{k-1} = 2^k$, qui est bien la puissance de 2 suivante.

5 Pour aller plus loin

Nous pourrions étendre notre problème en passant de nos petits carrés dans le plan à des petits cubes dans l'espace. Nous pourrions également essayer de paver des carrés ou des rectangles à l'aide de ces formes. Finalement, nous pourrions essayer d'améliorer nos algorithmes afin de calculer plus de nombres dans les suites.

6 Conclusion

Nous souhaitons remercier particulièrement les chercheurs et enseignants qui nous ont accompagnés pendant toute l'année : le professeur Yvik Swan, docteur en mathématiques (en statistique et probabilités) et chargé de cours à l'ULg et à l'ULB, madame Émilie Clette, assistante à l'ULg, mesdames Céline Serta et Ingrid Demulder-t'Kindt, organisatrices de Math.en.JEANS dans notre école, au collège Don Bosco. Elles ont aussi sacrifié un midi par semaine ! Enfin, nous souhaitons aussi remercier le département de mathématiques de l'ULg pour avoir organisé le congrès Math.en.JEANS fin avril.

Si vous pensez avoir compris une logique dans les suites, ou si vous avez des questions, des suggestions ou des remarques, nous vous encourageons à nous écrire un email à l'adresse prieelslucas@gmail.com.

Nous avons appris à la fin de l'année qu'un site internet, `oeis.org`, avait beaucoup d'informations sur toutes les suites de nombres que nous avons pu calculer. Cependant, nous avons préféré présenter seulement notre travail personnel dans cet article et pas des informations que n'importe qui aurait pu trouver en cherchant sur internet.

7 Annexes

7.1 $\sum$

Le symbole somme ($\sum$, la lettre majuscule sigma en grec), aussi appelé symbole de sommation, est une notation mathématique très utilisée. Il sert à noter une somme de plusieurs termes sans devoir utiliser les points de suspension. Pour cela, nous utilisons un indice, la plupart du temps noté i. Nous allons faire varier cet indice dans une intervalle de nombres. Par exemple [4],

$$\sum_{i=1}^{n} x_i$$

est la notation simplifiée de

$$x_1 + x_2 + x_3 + ...x_{n-1} + x_n$$

Nous pouvons remarquer qu'en dessous du symbole-somme, nous allons initialiser l'indice i, il sera ensuite incrémenté de 1 à chaque itération, jusqu'à ce que $i = n$.

4. La lecture de ce symbole est "la somme pour i variant de 1 à n de x_i"

7.2 Algorithme récursif

Un algorithme récursif est un algorithme qui utilise au moins une fonction récursive. Nous allons maintenant détailler le principe d'une fonction récursive.

Basiquement, une fonction récursive est une fonction informatique qui s'appelle elle-même. Pour mieux comprendre, nous allons donner un exemple : la fonction factorielle.

Tout d'abord, la factorielle [5] de n est définie comme $n \times (n-1) \times (n-2) \times ... \times 2 \times 1$ et est notée $n!$. Par convention, $0! = 1$ Nous pouvons ainsi faire une fonction récursive pour la factorielle. Nous pouvons appeler une fonction récursive $fact(n)$ qui calcule $n!$ en posant $fact(n) = n \times fact(n-1)$ et en utilisant le fait que $fact(0) = 1$.
Par exemple, $fact(5) = 5 \times fact(4)$, mais là, nous pouvons recommencer l'opération avec $fact(4) = 4 \times fact(3)$. Ainsi,

$$fact(5) = 5 \times fact(4) = 5 \times 4 \times fact(3) = 5 \times 4 \times 3 \times fact(2) = 5 \times 4 \times 3 \times 2 \times fact(1)$$

$$= 5 \times 4 \times 3 \times 2 \times 1 \times fact(0) = 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 5!$$

7.3 Preuve par récurrence

Nous allons expliquer le principe de la preuve par récurrence. C'est une preuve qui peut seulement être utilisée pour prouver une propriété sur les entiers naturels qui se déroule en deux étapes.

D'abord, l'initialisation : nous allons prouver qu'une propriété est vraie pour le plus petit entier qui nous intéresse (le plus souvent 0). Ensuite, c'est l'hérédité. Nous allons prouver que si cette propriété est vraie pour un entier appelé k, alors elle est vraie pour $k+1$.

Cela signifie que, d'après l'initialisation, la propriété est vraie pour 0. Or si c'est vrai pour 0, c'est vrai pour 1, d'après l'hérédite. Mais si c'est vrai pour 1, c'est vrai pour 2. Nous pouvons continuer ainsi en avançant d'un entier à la fois et cela veut dire que ça sera vrai pour tous les entiers naturels.

5. La factorielle de n est aussi appelée factorielle n ou n factorielle.

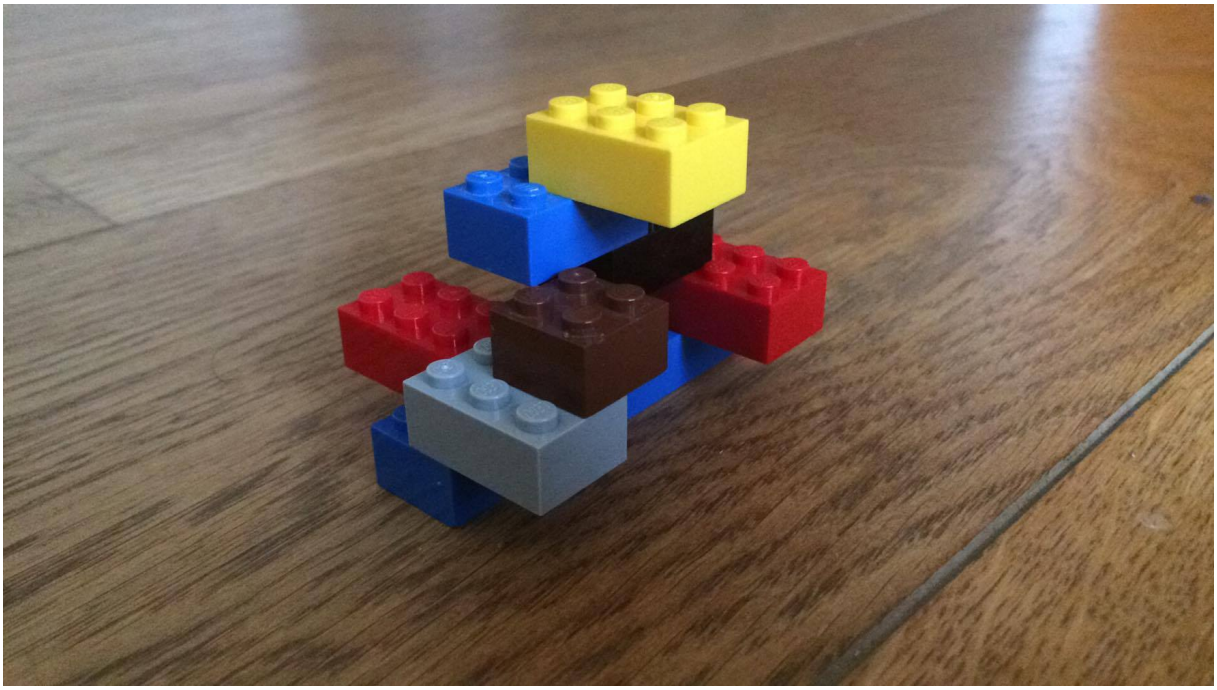


FIGURE 3 – Et après...