

PROJET EULER  
Problème 117

## Table des matières


<b>I</b>	<b>Résolution du problème</b>	<b>2</b>
<b>1</b>	<b>Présentation</b>	<b>2</b>
<b>2</b>	<b>Méthodes de réflexion</b>	<b>3</b>
2.1	Première méthode (méthode récursive) . . . . .	3
2.1.1	Algorithme de principe . . . . .	3
2.1.2	Développement . . . . .	3
2.2	Seconde méthode (méthode tetranacci) . . . . .	4
2.2.1	Algorithme de principe . . . . .	4
2.2.2	Développement . . . . .	4
<b>II</b>	<b>Optimisation</b>	<b>5</b>
<b>3</b>	<b>Objectifs</b>	<b>5</b>
<b>4</b>	<b>Méthode finale</b>	<b>5</b>
4.1	Algorithme de principe . . . . .	5
4.2	Développement . . . . .	5
<b>III</b>	<b>Comparaison des trois méthodes</b>	<b>6</b>
<b>5</b>	<b>Comparaison en temps</b>	<b>6</b>
<b>6</b>	<b>Complexité</b>	<b>6</b>
<b>IV</b>	<b>Nouvelle méthode récursive</b>	<b>7</b>
<b>V</b>	<b>Annexes</b>	<b>8</b>

## Première partie

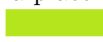









# Résolution du problème





## 1 Présentation

Le principe de ce problème est de remplir un certain nombre de cases avec des rectangles plus ou moins long. Les bleus mesurent 4 de long, les vert 3 et les rouge 2. Une dernière contrainte est imposée : l'ordre compte.

Prenons 4 cases vides.  On peut choisir de les remplir par "rien". Déjà une possibilité.

On peut maintenant choisir de mettre un bleu : . Il mesure 4, il prend ainsi toute la place. On veut maintenant placer des verts. On peut les placer de deux manières différentes :

  ou  . Enfin, il faut placer les rouges. Il y a bien évidemment ces dispositions :    et   .

On remarque que l'on peut également mettre 2 rouges à la suite :  . Mais étant donné que l'ordre des rectangles compte, la possibilité suivante est aussi bonne  . On arrive donc à un bilan de 4 possibilités pour les rouges, 2 pour les verts, 1 pour les bleus et la possibilité vide. Il y a 8 façons de remplir 4 cases, en suivant les contraintes de ce problème.

Pour donner un autre exemple, il y a 15 manières de remplir 5 cases :



Le but du problème est de déterminer combien il y a de possibilités de remplir 50 cases.

## 2 Méthodes de réflexion

### 2.1 Première méthode (méthode récursive)

Cet algorithme prend en entrée un entier  $n$  (représentant le nombre de cases à remplir), et retourne le nombre de possibilités de remplir les  $n$  cases, en respectant les contraintes du problème. Il s'effectue de manière récursive. Par exemple, en rentrant 5, le programme renvoie 15.

#### 2.1.1 Algorithme de principe

```
1  Algorithme : Projet Euler, Problème 117, Méthode récursive
2  Paramètres : Nombre n -> le nombre de cases à remplir.
3  Type de Retour : Nombre -> le nombre de façon de remplir les n cases.
4
5  si n = 5:
6      retourner 15
7  sinon si n = 4:
8      retourner 8
9  sinon si n = 3:
10     retourner 4
11 sinon si n = 2:
12     retourner 2
13
14 retourner Méthode récursive(n - 1) + Méthode récursive(n - 2) + Méthode récursive(n - 3) + Méthode récursive(n - 4)
```

#### 2.1.2 Développement

Dans cette méthode récursive, nous considérons que pour remplir les 50 cases il faut déjà commencer par choisir comment remplir la première case. Il y a 4 réponses possibles à cette question : soit on ne la remplit pas, soit on pose un rectangle de 2 cases, soit de 3 cases ou soit de 4 cases. Dans le premier cas, il ne restera plus que 49 cases à remplir. Dans le deuxième cas, il n'en restera que 48, 47 dans le troisième et 46 dans le dernier. C'est pourquoi nous appelons la même fonction de façon récursive pour calculer le nombre de possibilités de remplir 49, 48, 47 et 46 cases. Ensuite on répète ce processus d'appel récursif sur les 4 précédents jusqu'à trouver un cas d'arrêt. Les cas d'arrêts sont au nombre de 4 avec cette méthode. En effet, on connaît grâce à l'exemple donné précédemment, le nombre de possibilités pour remplir 5 cases, donc 5 est un cas d'arrêt et on retourne 15 dans ce cas-là. Mais il est possible que l'on n'appelle jamais la fonction avec  $n = 5$ . En effet, lorsque  $n = 6$ , on appelle *recursive*(5), *recursive*(4), *recursive*(3) et *recursive*(2). Ainsi, 4, 3 et 2 sont des cas d'arrêts également.

Ceci correspond ainsi à une modélisation de la suite de Tetranacci, une dérivée de la suite de Fibonacci. En effet, si on appelle  $F$  la suite de Fibonacci, on sait que  $F(n) = F(n-1) + F(n-2)$ . Ici, on remarque que  $recursive(n) = recursive(n-1) + recursive(n-2) + recursive(n-3) + recursive(n-4)$ . Ceci correspond à une suite où chaque terme est la somme des 4 précédents, la suite de tetranacci. Malheureusement, cette méthode est beaucoup trop longue, comme pour Fibonacci. En prenant 50 comme entrée, elle doit faire un nombre incalculable d'appels récursifs. Il faut donc la modifier pour ne plus faire appels à la fonction plusieurs fois avec la même entrée. En effet, ici, on appelle plusieurs fois la même fonction avec le même argument. Par exemple, en partant de 50, on l'appelle avec 49, 48, 47 et 46. Mais avec 49, on l'appelle encore avec 48, 47 et 46. Ceci n'est que le début, au total, le programme appellera *recursive*(6) 1 965 381 541 064 de

fois, pour donner un exemple. Il faut donc stocker le résultat de chaque appel pour ne plus avoir à le calculer.

## 2.2 Seconde méthode (méthode tetranacci)

Cet algorithme prend en entrée un entier  $n$  (représentant le nombre de cases à remplir), et retourne le nombre de possibilités de remplir les  $n$  cases, en respectant les contraintes du problème. Ici on stocke dans une liste les 4 précédents termes.

### 2.2.1 Algorithme de principe

```
1 Algorithme : Projet Euler, Problème 117, Méthode tetranacci
2 Paramètres : Nombre n -> le nombre de cases à remplir.
3 Type de Retour : Nombre -> le nombre de façon de remplir les n cases.
4
5 liste := [0, 0, 0, 1]
6 Pour i allant de 4 à n + 4:
7     ajouter (liste[i-4] + liste[i-3] + liste[i-2] + liste[i-1]) à liste
8
9 retourner liste[n+3]
```

### 2.2.2 Développement

Ici, on remarque que la suite de Tetranacci (que l'on appellera  $T$ ) commence, comme Fibonacci, à 0. Ainsi,  $T(0) = 1, T(1) = T(0), T(2) = T(0) + T(1)$  et  $T(3) = T(0) + T(1) + T(2)$ . On sait que chaque terme est la somme des 4 précédents. Il faut donc stocker ces 4 termes pour ne pas avoir à les recalculer. On stocke donc dans une liste 4 valeurs. A l'initialisation, il ne doit y avoir que  $T(0)$ . Or, pour calculer  $T(1)$ , on doit avoir accès à 4 valeurs pour pouvoir calculer une somme. C'est pourquoi on initialise cette liste à  $[0, 0, 0, T(0)]$ . Ensuite, tant que nous n'avons pas  $n + 3$  valeurs dans la liste, on y ajoute le prochain terme en faisant la somme des 4 derniers termes.  $T(n)$  sera le  $n + 3^{\text{ème}}$  terme puisque les 3 premiers ne font pas à proprement parler de la suite. Ainsi, on calcule  $T(n)$  très rapidement, en ne calculant que  $n$  sommes.  $T(50) = 100808458960497$ , il y a donc 100808458960497 possibilités de remplir les 50 cases avec des rectangles de longueur 4, 3 et 2.

## Deuxième partie

# Optimisation

### 3 Objectifs

Après avoir résolu, le problème nous nous sommes fixés de nouveaux objectifs :

- Essayer d'obtenir un algorithme plus simple, qui pourrait également prendre moins de place mémoire.
- Améliorer la lisibilité et la compréhension de nos méthodes, et si nécessaire en refaire une autre.

### 4 Méthode finale

Cet algorithme prend en entrée un entier  $n$  (représentant le nombre de cases à remplir), et retourne le nombre de possibilités de remplir les  $n$  cases, en respectant les contraintes du problème. On reprend le même principe que l'algorithme précédent, mais on stocke les termes précédents dans des variables.

#### 4.1 Algorithme de principe

```
1 Algorithme : Projet Euler, Problème 117, Méthode finale
2 Paramètres : Nombre n -> le nombre de cases à remplir.
3 Type de Retour : Nombre -> le nombre de façon de remplir les n cases.
4
5 a := 0, b := 0, c := 0, d := 1
6 Pour i allant de 0 à n:
7     d := a + b + c + d, a := b, b := c, c := d
8
9 retourner d
```

#### 4.2 Développement

Cette méthode est identique à la précédente à une différence près. Cette différence permet de réduire l'utilisation de l'espace mémoire puisque plutôt que de réserver  $n + 3$  cases mémoires, on n'en réserve que 4 dans tous les cas. En effet, pour calculer  $T(n)$ , on n'a besoin que des 4 termes précédents. On peut donc n'utiliser que 4 variables qui correspondent aux 4 derniers termes calculés. Comme précédemment, 3 variables sont initialisées à 0, et la dernière correspond à  $T(0)$ , elle est donc initialisée à 1. Ensuite, on réalise  $n$  fois la somme des 4 variables afin de calculer le prochain terme. Ce résultat est stocké dans la dernière variable et les 3 autres prennent la valeur du terme qui les suit. Après  $n$  sommes, la 4ème variable est donc égale à  $T(n)$ , on peut donc la renvoyer.

## Troisième partie

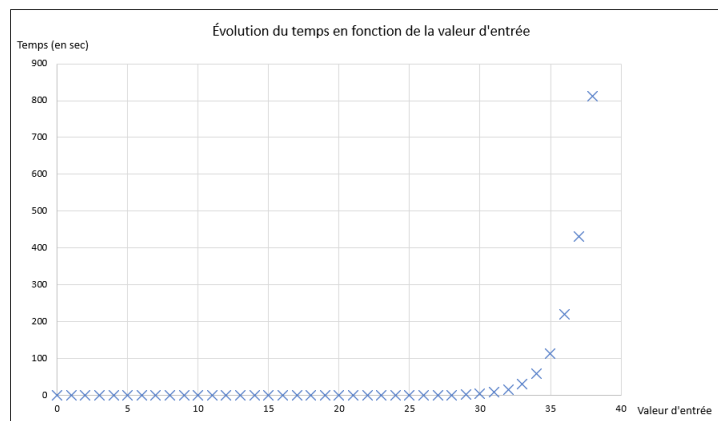
# Comparaison des trois méthodes

## 5 Comparaison en temps

Afin de mettre en évidence les différences d'efficacité entre nos méthodes, nous avons réalisé des tests. La première méthode est très limitée. En effet, comme expliqué précédemment, elle nécessite beaucoup d'appels. Quand on cherche à résoudre le problème avec comme entrée 36, le programme met 219 secondes à retourner le résultat. Elle n'est pas du tout efficace. Les 2 autres méthodes quant-à-elles sont réellement plus efficaces. En effet, pour tout nombre entré inférieur à 100 000, elles mettent moins d'une secondes à retourner le bon résultat. On privilégiera donc ces deux méthodes.

## 6 Compléxité

Afin d'établir la compléxité des méthodes, nous avons établi un graphique des résultats obtenu grâce à la première méthode.



On observe ainsi que la complexité de cette méthode est exponentielle,  $O(e^n)$ . Cela s'explique par le nombre d'appel récursif effectué. En effet, après une démonstration difficile et qui sort du cadre de ce problème<sup>1</sup>, nous obtenons bien une complexité exponentielle. On comprend ainsi pourquoi elle n'est pas du tout efficace comme méthode de résolution.

La complexité des deux autres méthodes est linéaire,  $O(n)$ , car on effectue  $n$  fois les mêmes opérations.

On peut conclure qu'il est impensable d'utiliser la première méthode. Le choix d'utiliser la seconde ou la dernière méthode est en grande partie esthétique, même si la dernière utilise moins de place mémoire.

1. Nb : pour plus d'infos sur le calcul de la complexité, voir <https://www.supinfo.com/cours/2ADS/chapitres/02-complexite-algorithmes-recursifs> et se renseigner sur les suites récurrentes linéaires homogènes d'ordre quatre

## Quatrième partie

# Nouvelle méthode récursive

Suite à notre dernier entretien, nous avons essayé de refaire une nouvelle méthode récursive. Le problème de notre précédente méthode récursive, était le trop grand nombre d'appels. Pour résoudre ce problème, il faudrait pouvoir stocker nos termes d'un appel à un autre. Pour cela, on retourne une liste contenant les 4 derniers termes calculés. Cela va nous permettre de réduire la complexité. Voici le code en Python :

```
1 def recursive2(n):
2     if n == 0:
3         return [0, 0, 0, 1]
4     else:
5         temp = recursive2(n-1)
6         temp3_next = temp[0] + temp[1] + temp[2] + temp[3]
7         return [temp[1], temp[2], temp[3], temp3_next]
8
9
10 nbTeste = int(input("Rentrez le nombre de case disponible : "))
11
12 result = recursive2(nbTeste)[3]
13 # car le résultat final se trouve dans la 4ème "case" de la liste
14
15 print("Résultat au problème 117 avec", nbTeste, "comme entrée :", result)
```

On voit ici que le principe est similaire à la seconde méthode. Étant donné que l'on part de  $n$ , que l'on décrémente de 1 à chaque fois jusqu'à 0, on fait  $n$  appels. La complexité est donc  $O(n)$ . On obtient ainsi une méthode récursive, au moins aussi efficace que les méthodes itératives, dont la complexité est linéaire (et non exponentielle comme la précédente méthode récursive). Cependant on se heurte à un autre problème : les limites de récursions. En effet, en Python comme dans d'autres langages, il existe une limite de récursions possibles. C'est à dire que l'on ne peut pas faire plus de récursions que cette limite. Cette limite est en fait due à la place en mémoire que prend les récursions. Quand la fonction s'appelle elle-même, son état est mis en pause et stocké en mémoire. Dans notre cas, la limite était de 1000 récursions. On ne peut donc pas conclure sur l'efficacité de cette méthode à long terme.

## Cinquième partie

# Annexes

## Méthode récursive

```
1 def recursive(n):
2     """
3     :param n: nombre de cases à remplir
4     :return: nombre de possibilités pour remplir n-cases
5     """
6     if n == 5:
7         return 15
8     elif n == 4:
9         return 8
10    elif n == 3:
11        return 4
12    elif n == 2:
13        return 2
14    return recursive(n - 1) + recursive(n - 2) + recursive(n - 3) + recursive(n - 4)
15
16 nbTeste = int(input("Rentrez le nombre de case disponible : "))
17 print("Résultat au problème 117 avec", nbTeste, "comme entrée :", recursive(nbTeste))
```

## Méthode tetranacci

```
1 def tetranacci(n):
2     """
3     :param n: nombre de cases à remplir
4     :return: nombre de possibilités pour remplir n-cases
5     """
6     l = [0,0,0,1]
7     for i in range(4,n + 4):
8         l.append(l[i - 4] + l[i - 3] + l[i - 2] + l[i - 1])
9     return l[n + 3]
10
11 nbTeste = int(input("Rentrez le nombre de case disponible : "))
12 print("Résultat au problème 117 avec", nbTeste, "comme entrée :", tetranacci(nbTeste))
```



## Méthode finale

```
1 def finale(n):  
2     '''  
3     :param n: nombre de cases à remplir  
4     :return: nombre de possibilités pour remplir n-cases  
5     '''  
6     a, b, c, d = 0, 0, 0, 1  
7     for i in range(n):  
8         a, b, c, d = b, c, d, (a+b+c+d)  
9     return d  
10  
11 nbTeste = int(input("Rentrez le nombre de case disponible : "))  
12 print("Résultat au problème 117 avec", nbTeste, "comme entrée :", finale(nbTeste))
```