Partie n°4 : Récursivité et efficacité

a) Écrire un programme itératif (utilisant uniquement des boucles *for* et *while*) qui calcule le PGCD de deux nombres entiers *a* et *b* en traduisant l'algorithme d'Euclide. Écrire ce programme en mode récursif (utilisant une fonction qui s'appelle elle-même).
Le mode itératif (ou séquentiel) est le mode habituel, celui que l'on apprend au début : il utilise des boucles *for* et/ou des boucles *while* et c'est tout. La fonction *pgcd*() que nous avons écrit (à gauche) est une traduction de l'algorithme d'Euclide dans ce mode : on effectue des divisions euclidiennes sans se préoccuper du quotient, en utilisant juste le reste *reste=a%b*. Dans la boucle *while*, on remplace (*a,b*) par (*b,reste*) jusqu'à ce que *reste*==0 soit *True* (tant que le reste est différent de zéro). Le PGCD cherché est *b*, le dernier diviseur.
La récursivité simplifie souvent l'écriture d'une fonction, surtout quand celle-ci est, par nature, récursive et c'est le cas de l'algorithme d'Euclide avec la propriété PGCD(*a,b*)=PGCD(*b,reste*) tant que *reste*≠0. La traduction en mode récursif (à droite) de cet algorithme utilise cette propriété dans le cas général avec un test d'arrêt qui examine la valeur du reste et arrête le processus quand ce reste est nul. Ainsi pgcd(12,8) calcule 12%8=4 et comme 4≠0 retourne pgcd(8,4) qui calcule 8%4=0 et comme 0=0 retourne 4 qui se substitue à pgcd(8,4) puis à pgcd(12,8) comme valeur de retour.
Ces deux algorithmes donnent exactement le même résultat (heureusement!) et le programme principal fonctionne aussi bien avec l'un qu'avec l'autre. La boucle récursive est un peu plus courte à écrire mais si on mesure très précisément les temps d'exécution, elle est un peu plus longue à s'exécuter. Il faut tout de même prendre des valeurs assez grandes pour que cette différence soit sensible : un peu plus d'un dix millièmes de seconde d'écart pour déterminer que
PGCD(573147844013817084101,354224848179261915075)=1 (nous avons pris le 101ème et le 100ème terme de la suite de Fibonacci car, à nombres de tailles égales, c'est pour les termes consécutifs de cette suite qu'il y a le plus de boucles à effectuer ; ici, il y a 99 divisions à effectuer pour les deux programmes).
On peut dire que cette différence d'efficacité n'est pas significative. Elle ne pénalise pas le programme le plus lent, même si, lorsque les nombres augmentent, le retard s'accroit légèrement aussi.

b) Parfois l'écriture récursive est moins efficace que son équivalent itératif, car les appels multiples à la fonction créent autant d'environnements d'exécution qu'il en faut. Ces environnements s'ignorant les uns les autres peuvent être redondants. Pour mieux comprendre cela, programmer le calcul d'un terme de la célèbre suite de Fibonacci : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. (chacun des termes étant la somme des deux précédents) dans les deux modes (itératif et récursif). Mesurer les temps d'exécution avec la fonction *clock*() du module *time* : cette fonction renvoie un temps très précis de l'horloge interne en secondes ; pour mesurer une durée, il faut faire la différence entres le temps de fin et de début d'exécution.
La programmation en est simple dans les deux modes, mais le récursif est le plus simple des deux car il colle mieux à la définition du terme de rang *n* comme la somme des termes de rang *n*−1 et *n*−2. Par contre, l'exécution de la fonction récursive (*fibo2*() dans notre illustration) prend de plus en plus de retard par rapport à la fonction itérative quand *n* augmente. Pour le calcul du 120ème terme de la suite, la fonction itérative répond presque instantanément, alors que la fonction récursive est au bord du dépassement de capacité et prend plusieurs dizaines de minutes pour s'exécuter. Pourquoi cela arrive t-il ? Parce qu'à chaque appel récursif, Python crée deux nouveaux environnements d'exécution qui chacun lancent deux nouveaux environnement d'exécution qui chacun… *fibo2*(120) demande le calcul de *fibo2*(119) et *fibo2*(118) ; jusque là tout va bien, sauf que *fibo2*(119) demande le calcul de *fibo2*(118) et *fibo2*(117), en ignorant complètement le fait que *fibo2*(118) a déjà été demandé… Et le processus continue créant des redondances : *fibo2*(118) est demandé 2 fois, *fibo2*(117) est demandé 3 fois, *fibo2*(116) est demandé 5 fois, … (cette progression semble emboîter le pas d'une suite de Fibonacci).
Pour pallier à ce défaut majeur du mode récursif naïf (qui ne se rencontre pas pour toutes les fonctions récursives, il faut le dire), on doit éviter ce genre d'appels multiples, et réaliser des fonctions récursives qui ne s'appellent qu'une seule fois (pour éviter les redondances). Voici le calcul des termes de la suite de Fibonacci par un algorithme récursif de ce genre, qualifié ici de « terminal » (car les calculs s'effectuent directement en descendant dans la pile d'exécution, au lieu de se faire en remontant). Les temps de réponses sont donnés à droite pour comparer avec les deux autres méthodes.
L'astuce, pour ne lancer qu'une seule fonction *fibo3*() à chaque fois (au lieu de deux pour la fonction *fibo2*()) consiste en l'insertion de deux arguments supplémentaires (les deux termes qui précèdent celui

qu'on doit calculer) qui sont calculés avant l'envoi de la fonction : au démarrage, *fibo3*(120) contient des arguments par défaut *a*=0 et *b*=1 (les deux premiers termes de la suite) et demande le calcul de *fibo3*(119,1,2) qui demande *fibo3*(118,2,3), etc. jusqu'à *fibo3*(1, 331164814351698201718008 1, 535835925499096664087184 0) qui renvoie le résultat final 120!=535835925499096664087184 0 car, alors, *n*=1.
La notation (*n*, *a*=0, *b*=1) dans les arguments de la fonction signifie que 0 et 1 seront affectés par défaut aux variables *a* et *b* si ces valeurs ne sont pas renseignées lors de l'appel. Quand nous appelons cette fonction par *fibo3*(*n*) en ne renseignant pas les deux derniers paramètres, ce sont les valeurs par défaut qui seront prises. Par la suite, les appels *fibo3*(*n-1,b,a+b*) renseignent les valeurs de *a* et de *b* ce qui désactive les affectations par défaut.

c) Écrire un programme itératif qui calcule la factorielle $n! = 1\times 2\times ...\times n = \prod_{k=1}^{n} k$ d'un nombre entier *n*.
En déduire un programme qui calcule les coefficients binomiaux $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, aussi appelés (en dénombrement) nombres de combinaisons de *k* éléments pris parmi *n*, ou coefficients du triangle de Pascal.
Écrire ces deux programmes en mode récursif (utilisant une fonction qui s'appelle elle-même).
Disons le tout de suite : dans le module *math* de Python, la fonction factorielle existe ! Il suffit de l'appeler avec l'instruction *factorial*(3) (après avoir importé les fonctions de ce module par *from math import* *). Ce n'est donc qu'à titre d'exercice (comme souvent, comme pour le PGCD qui existe aussi, bien sûr, dans les bibliothèques de Python) que nous chercherons à programmer par nous-même cette fonction.
En mode itératif, il suffit de faire varier *i* de 1 à *n* dans l'affectation *fact=fact*i* qui s'écrit aussi *fact*=i.* On ne doit pas non plus oublier d'initialiser *fact* à 1. Le programme ci-contre suffit pour calculer 10!
Pour un affichage plus propre qui nous rappelle la valeur de *n*, la variable *n* est nommée (affectée en mémoire) car elle est utilisée deux fois, par exemple dans une instruction qui demande à l'opérateur d'entrer cette variable. On peut étoffer la dernière ligne avec du texte : *print*('{}!={}'.*format*(n,fact)) ou *print*(*str*(n)+'!='+*str*(fact)). Pour une réutilisation ultérieure, on écrit une fonction *factorielle*() qui se charge du calcul, tandis que le programme principal gère les entrées-sorties.
Pour le calcul du coefficient binomial en mode itératif $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, c'est très facile : on utilise trois fois la fonction *factorielle*() précédente sans se demander si la fonction existe déjà dans un des modules de Python. Mais on peut faire légèrement mieux pour calculer $\frac{n!}{k!(n-k)!} = \frac{n\times(n-1)\times(n-2)...\times(n-k+1)}{k\times(k-1)\times(k-2)...\times(1)}$ car il y a autant de facteurs au numérateur qu'au dénominateur (il y en a *k*), par exemple $\frac{6!}{4!(6-4)!} = \frac{6\times5\times4\times3}{4\times3\times2\times1}$ (il y a 4 facteurs). On peut donc exécuter une seule boucle en faisant varier *i* de 1 à *k* dans l'affectation *coeff=coeff*(n-i*+1)/*i* qui s'écrit aussi *coeff*=(n-i*+1)/*i*. Cette nouvelle version itérative de calcul du coefficient binomial est plus efficace car elle se passe de la fonction *factorielle*() (dans la formule, les trois factorielles servent davantage à simplifier l'expression qu'à effectuer concrètement les calculs).

La traduction de la fonction *factorielle*() en une fonction récursive (qui s'appelle elle-même) ne pose pas de problème. C'est l'exemple typique (avec l'algorithme d'Euclide) qui est donné pour illustrer ce qu'est une fonction récursive. Le principe est que l'on doit mettre la condition d'arrêt au cœur de la fonction, généralement c'est une simple instruction conditionnelle qui réalise cela.
Si la récursivité est une nouveauté pour vous, prenez le temps d'analyser ce que fait cette fonction : prenons un petit exemple, pour *n*=3. Le programme demande *factorielle*(3), au test *nb*==1, la réponse est *False* donc la fonction retourne 3×*factorielle*(2), mais le fait d'écrire *factorielle*(2) lance la fonction qui, au test *nb*==1 répond *False*, retourne 2×*factorielle*(1), mais le fait d'écrire *factorielle*(1) lance la fonction qui, au test *nb*==1 répond ***True*** et retourne 1. Ce 1 renseigne le calcul 2×*factorielle*(1) qui vaut maintenant 2, ce qui renseigne le calcul 3×*factorielle*(2) qui vaut maintenant 6, ce qui est retourné au programme principal.

n/k	0	1	2	3	4	5	6	7	8	9	10
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Pour calculer les coefficients binomiaux en mode récursif, on va se servir d'une de leurs propriétés : $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Cette propriété est souvent illustrée par la disposition du « triangle de Pascal » qui dispose les coefficients binomiaux en un tableau triangulaire. On y calcule un des nombres en faisant la somme des deux nombres situés à la ligne d'avant, même colonne et colonne de gauche.

Par exemple, $\binom{7}{2}=\binom{6}{1}+\binom{6}{2}=6+15=21$ (21 est au croisement de la ligne 7 colonne 2, il est égal à la somme de 6 qui se trouve au croisement de la ligne 6 colonne 1 et de 15 qui se trouve au croisement de la ligne 6 colonne 2. Bien sûr, pour remplir le tableau ainsi, il faut savoir que sur la colonne 0 il n'y a que des 1comme, de même, sur la diagonale du tableau où le n° de ligne égale le n° de colonne.

Bien sûr, on pourrait utiliser cette propriété sur un mode itératif (un peu compliqué à mettre en place avec les deux indices de ligne et de colonne), mais ici, nous allons l'employer en mode récursif car la traduction en est immédiate (on n'a pas besoin de gérer des indices). Cela s'écrit comme la propriété l'énonce car il s'agit d'un relation de récurrence qui définit un terme d'une suite en fonction de termes voisins qui ont eux mêmes été calculés de la même façon, à partir des premiers termes qui sont connus.

L'inconvénient de cette définition récursive se fait sentir si on veut calculer des termes d'indices trop élevés. Regardez les temps d'exécution de cette méthode récursive *coeffBin3*() pour le calcul de $\binom{25}{13}$: 10,5 secondes ! Les méthodes 1 et 2 qui utilisent les définitions itératives et récursives de la factorielle mettent respectivement 0,00005 et 0,0002 secondes approximativement pour faire ce calcul. L'explication de ce temps très long pour la méthode récursive est la même que pour le calcul des termes de la suite de Fibonacci : des appels redondants qui ralentissent la fonction de plus en plus quand *n* augmente jusqu'à l'empêcher (si les nombres en Python ne sont pas limités, la pile d'exécution, qui se chargent des calculs effectués dans tous les environnements cloisonnés d'exécution, a une taille limitée qui provoque l'arrêt des calculs en cas de débordement). Il faut écrire une fonction récursive qui ne s'appelle qu'une fois. Nous avons écrit cette fonction « terminale », *coeffBin4*(), en se servant de la relation déjà employée pour améliorer l'algorithme itératif entre deux coefficients voisins d'une même colonne : $\binom{n}{k}=\binom{n}{k-1}\times\frac{n-k+1}{k}$.

La condition d'arrêt est l'arrivée dans les colonnes 1 ou 0 dans lesquelles on trouve toujours *n* (pour la colonne 1) car $\binom{n}{1}=\frac{n!}{1!(n-1)!}=n$ ou 1 (pour la colonne 0).

Notons au passage que nos fonctions *factorielle*() ne donnent pas une valeur correcte pour 0 ! (l'itérative *factorielle1*() donne -1 alors que la récursive *factorielle2*() conduit à une belle boucle infinie). La valeur correcte de 0 ! est 1. Avec cette valeur, la formule du coefficient binomial donne toujours une valeur correcte.

d) Le mode récursif est réputé plus efficace que le mode itératif dans certains cas. Pour trier une liste de nombres initialement dans le désordre, on peut procéder ainsi : choisir un des nombres comme pivot, comparer les nombres de la liste au pivot et les ranger dans deux listes distinctes (les nombres inférieurs dans *listInf* et les nombres supérieurs dans *listSup*), appliquer la même méthode de tri sur les deux listes. Écrire le programme récursif qui effectue le tri de cette façon. Tester ce programme sur une liste de *n* nombres tirés au hasard. Écrire un programme itératif qui réalise le même travail et comparer les deux sur leurs performances respectives.

L'algorithme de tri rapide proposé (*quick sort* en anglais) demande de choisir un pivot qui peut être choisi au hasard (pivot=*random.choice*(liste)) ; on peut aussi bien choisir le pivot toujours au début (pivot=liste[0]), ou bien au milieu approximativement de la liste (pivot=liste[*len*(liste)//2]). On pourra tester ces trois options. La procédure suivie est par nature récursive : on compare les éléments de la liste au pivot ; s'ils y sont inférieurs, on les range dans une *listeInf*, sinon dans une *listeSup*. S'ils sont égaux au pivot on les range dans une 3ème liste (*listeMid*). Ensuite, on renvoie le résultat de la concaténation des 3 listes triées (la *listeMid* n'a pas besoin d'être triée). Donc on relance la fonction de tri d'une liste, mais avec des listes plus petites. La condition d'arrêt du tri porte sur la taille de la liste à trier qui ne peut être égale à 0 ou 1 (il n'y aurait alors rien à trier) : dans ces deux derniers cas, on renvoie la liste reçue. En remontant dans la pile d'exécution, les listes triées se concatènent pour former la liste triée finale.

Le mode itératif copie le fonctionnement de ce tri par pivot. Mais il est difficile, voire impossible, de faire exactement la même chose qu'en mode récursif. Nous avons opté pour une solution qui compare les éléments de la liste au pivot et les insère dans la liste qui convient (*listeInf, listeSup* ou *listeMid*), au bon endroit, créant ainsi trois listes triées qu'il suffit de concaténer.

Le programme récursif est le plus efficace, et cette efficacité augmente quand la taille de la liste augmente.

Cela vient du nombre de comparaisons à faire qui devient énorme en mode itératif comparativement au mode récursif 42 089 comparaisons en moyenne pour 500 nombres à trier en mode itératif alors qu'en mode récursif, il faut 7136 comparaisons en moyenne seulement (6 fois moins). Avec 100 nombres à trier, le rapport n'est que de 1,9 mais pour 2500 nombres, le rapport est de 22 (récursif : 47 561 comparaisons - itératif : 1 038 642 comparaisons). Ces nombres moyens de comparaisons sont à rapprocher des durées moyennes qui suivent globalement la même progression. Voici les chiffres obtenus lorsque le pivot est choisi au milieu de la liste (tableau ci-dessous).

1: choix du pivot au milieu				
Longueur de la liste	20	100	500	2500
Mode itératif				
nombre moyen de comparaisons	90	1841	42089	1038642
Durée moyenne (s)	0,0001067	0,001471	0,02953	0,7336
Mode récursif				
nombre moyen de comparaisons	112	976	7136	47561
Durée moyenne (s)	0,0001834	0,001229	0,00807	0,05072
Rapport itératif/récursif				
a=nombres moy. de comparaisons	0,80	1,89	5,90	21,84
b=durées moyennes	0,58	1,20	3,66	14,46
a/b	1,38	1,58	1,61	1,51

Nous pouvons recommencer ces mesures pour un autre choix du pivot. Est-ce qu'il y a une place optimale pour le pivot ou bien est-ce approximativement toujours la même efficacité moyenne (le nombre de comparaisons et la durée du tri dépendent évidemment de la liste initiale : si elle est déjà triée, il y aura forcément beaucoup moins de comparaisons). Il semblerait que les deux méthodes ont des efficacités plus proches lorsque le pivot est fixé au rang 0 (comparer les lignes a/b pour les trois options de choix du pivot) et plus éloignées lorsque le pivot est choisi de façon aléatoire. L'option 1 (choix au milieu de la liste) donne des résultats intermédiaires, mais c'est la meilleure option pour la méthode itérative (du moins pour des listes longues). La meilleure option pour la méthode récursive est moins flagrante : en durée c'est plus rapide avec l'option 3 mais en nombre de comparaisons, ce serait plutôt l'option 1 qui l'emporterait ; disons que les trois options sont sensiblement à égalité.
Nous n'avons envisagé qu'un seul algorithme de tri mais il en existe de nombreux, la situation de tri étant très fréquente dans les traitements informatiques. Python a, bien sûr, implémenté un algorithme de tri pour les listes. Nous avons souvent utilisé la méthode *sorted*(). Par exemple *sorted*([2,6,8,1,6,7,2,9,88,1]) renvoie la liste triée : [1, 1, 2, 2, 6, 6, 7, 8, 9, 88]. Mesurons la performance de l'algorithme utilisé par Python dans cette implémentation du tri : les chiffres parlent d'eux-mêmes, pour une liste de 2500 nombres à trier, la méthode *sorted*() est 25 fois plus rapide que notre meilleur choix en mode récursif ! Nous devons mentionner un handicap pour nos méthodes : afin de mesurer le nombre de comparaisons effectuées, nous avons incrémenté un compteur *nbComp* à chaque comparaison. Cela peut-il suffire à expliquer le fossé qu'il y a entre ces deux méthodes ? Pour répondre, il n'y a qu'à supprimer ce compteur.

Tri de Python avec sorted()				
Longueur de la liste	20	100	500	2500
Durée moyenne (s)	0,00001227	0,000064	0,0004	0,0026

Au lieu de 0,0500 seconde par liste de 2500 nombres, notre méthode récursive traite maintenant une liste en 0,0321 seconde quand *sorted*() de Python ne prend que 0,0025 seconde. Il y a eu un gain d'efficacité en enlevant ce compteur mais la méthode native de Python est tout de même près de 13 fois plus rapide...

e) Pour prolonger la réflexion sur les méthodes récursives de programmation, nous avons vu qu'il faut éviter les redondances qui sont la plaie des programmes récursifs (du moins pour des traitements de grande ampleur). Comment alors programmer la situation suivante où on a calculé qu'une probabilité p_n vaut $p_n = p_{n-1} + \frac{1-p_{n-6}}{26^6}$? Il s'agit de la probabilité d'écrire un mot spécifique de six lettres comme « COGITO » (*je pense*, en latin) lorsqu'on tape n lettres au hasard[1]. La situation est aggravée par le fait que n est nécessairement grand (n=2000 est un minimum envisageable) sinon p_n va être ridiculement petit.
La définition qui nous est donnée de p_n est une définition par récurrence, comme B. Rittaud le souligne dans son livre. Elle permet, connaissant les premiers termes (ici on doit connaître p_1, p_2, p_3, p_4, p_5 et p_6), de

1 Cette situation d'*un singe dactylographe*, imaginée par Émile Borel au début du XX^e, est mise en scène dans le livre de Benoît Rittaud : *L'assassin des échecs et autres fictions mathématiques* (coll. Plumes de Sciences, ed. Le Pommier, 1992), pp.165-179.

déterminer les suivants. Ici, le 5 premiers termes de cette suite sont nuls (on ne peut pas écrire COGITO qui contient 6 lettres en n'écrivant que 5, 4, 3, 2 ou 1 lettre). Par contre p_6 vaut $p_6=\frac{1}{26^6}\approx 3{,}237\times 10^{-9}$ puisqu'il y a $26^6=308915776$ mots de 6 lettres choisies au hasard parmi 26 (on suppose ici qu'on ne tape que des lettres, le clavier de la machine à écrire de notre singe dactylographe ne doit pas contenir de chiffres, ni d'autres symboles typographiques).
Cette définition par récurrence conduit naturellement à l'écriture d'une fonction récursive qui va générer de nombreuses redondances empêchant son exécution pour des valeurs trop élevées de n. On peut écrire cette fonction récursive *proba1*() pour évaluer jusqu'à quelle valeur de n on peut aller (sans parler du temps d'exécution qui va s'allonger, c'est la pile d'exécution qui va déborder, arrêtant le processus). Les premières valeurs sont obtenues en des temps raisonnables (inférieurs à 1 seconde tant que n ne dépasse pas 55). Mais ce n'est pas exactement ce que l'on souhaite car une ligne de 80 caractères c'est trop court (les lignes de ce document contiennent une centaines de caractères), la probabilité de l'évènement considéré est trop faible : environ 0,0000002478 soit une chance sur quatre millions.
Dans la nouvelle de B. Rittaud, c'est au moment où le lecteur inconnu du journal commence son exposé sur la façon d'obtenir une expression de p_n en fonction de n, que le rédacteur en chef de ce journal décide de mettre sa lettre au panier. Ce faisant, il nous prive de l'étude de cette suite récurrente d'ordre 6 que le lecteur inconnu avait semble t-il effectuée. Nous ne nous lancerons pas dans ce genre d'étude ici, mais pour évaluer cette probabilité, une formule approchée, un peu moins rigoureuse sans doute que la formule de récurrence (voir les explications données p.185-186), mais beaucoup plus simple à employer est donnée. Cette formule est $p_n=6\times\left(1-\left(1-\frac{1}{26^6}\right)^{\frac{n}{6}}\right)$. Elle ne justifie même pas un programme en Python tellement elle est aisée à mettre en œuvre. Avec une calculatrice ou un tableur, on trouve des valeurs sensiblement identiques aux résultats calculés avec notre programme récursif jusqu'à n=80, mais on peut aller bien au-delà.

n	6	10	20	50	60	70	80	100	1000	10000	100000	1000000	10000000	100000000	1000000000
p_n	1,94E-08	3,24E-08	6,47E-08	1,62E-07	1,94E-07	2,27E-07	2,59E-07	3,24E-07	3,24E-06	3,24E-05	3,24E-04	3,24E-03	3,23E-02	3,15E-01	2,50E+00

La valeur obtenue pour un milliard de caractères est assez étrange tout de même puisqu'elle dépasse 1 ! Le tracé de la courbe donné par Geogebra nous confirme ce dépassement de 1 à partir de $n\approx 300\,000\,000$ (voir la courbe). Le nombre $1-\frac{1}{26^6}$ est très proche de 1 (il vaut approximativement 0,9999999967628717), élevé à la puissance $\frac{10^9}{6}\approx 166667$, ce nombre vaut environ 0,5830272319, ôté de 1 cela fait à peu près 0,416972768, et en multipliant cela par 6, on trouve bien davantage que 1. Que doit-on conclure ? La probabilité est-elle tellement élevée que l'évènement est plus que certain ? Certainement pas. La formule est-elle fausse ? Pour cette tranche de valeurs, sans doute, ce qui ne l'empêche pas de donner des valeurs acceptables dans un domaine restreint.

En l'absence d'une formule satisfaisante, alors que l'écriture récursive immédiate ne donne rien de bien satisfaisant, une autre écriture récursive, plus efficace car ne s'appelant qu'une seule fois, semble assez tentante. Il suffit d'entrer les six premières valeurs de p_n et la formule de récurrence donnée plus haut donne l'idée de ce mécanisme récursif : au lieu de l'appel *proba1*(n-1)+(1-*proba1*(n-6))/(26**6) qui conduit à l'explosion rapide de l'espace alloué à la pile d'exécution (aux alentours de n=80), nous avons écrit la fonction *proba2*() qui appelle *proba2*(n-1,b,c,d,e,f,f+(1-a)/(26**6)). La formule $p_n=p_{n-1}+\frac{1-p_{n-6}}{26^6}$ peut, en effet, conduire à calculer directement un terme dès lors que l'on conserve les six termes précédents : $p_7=p_6+\frac{1-p_0}{26^6}$, dès que l'on connait p_7, on peut se passer de p_0, et on calcule $p_8=p_7+\frac{1-p_1}{26^6}$, etc.
La programmation récursive est plus astucieuse car elle ne crée pas de redondance, cependant elle encombre rapidement l'espace alloué à l'exécution car chaque environnement doit conserver 7 nombres en mémoire (la valeur de n et les six valeurs de p_n). Aux alentours de n=1000, cette fois, on assiste au débordement de la pile. Il faut donc renoncer à obtenir les valeurs recherchées (on s'était fixé un minimum à n=2000).
Une autre approche a été étudiée en classe de seconde. Ce n'est pas le calcul de la probabilité qui est visé mais son estimation grâce à l'intervalle de confiance dans lequel elle se situe. Nous allons procéder à la simulation de m échantillons de taille n et au calcul de la fréquence expérimentale de l'évènement : « au moins une occurrence de la séquence « COGITO » (les six lettres tapées par le singe dactylographe) » a été décelée dans l'échantillon. La fréquence f_e obtenue pour nos m échantillons de taille n permet d'affirmer, avec moins de 5% de chance de se tromper, que la probabilité p_n se situe dans l'intervalle $\left[f_e-\frac{1}{\sqrt{m}}\ ;\ f_e+\frac{1}{\sqrt{m}}\right]$. La mise en œuvre de cette méthode ne pose pas de problèmes d'écriture (le

programme de simulation est simple), mais les durées nécessaires pour générer des échantillons de grande taille sont prohibitives : 8 secondes pour générer un texte de un million de caractères choisis au hasard dans l'alphabet et examen de ce texte pour tenter d'y trouver la séquence « COGITO ». Nous allons cependant le lancer m=400 fois (pour avoir une amplitude de l'intervalle de confiance égal à 0,1) en allant jusqu'à n=2000 caractères (et davantage si c'est réalisable). Pour vérifier les valeurs obtenues par cette méthode statistique, nous pourrons lancer également notre programme *simul*() pour les valeurs n=250, 500 et 1000 pour lesquelles nous avons un calcul exact de probabilité.
L'inconvénient majeur de cette méthode est qu'elle ne fonctionne bien que lorsque la fréquence expérimentale n'est ni trop grande (supérieure à 0,8), ni trop petite (inférieure à 0,2). Pour notre situation, la fréquence est bien trop petite, ridiculement petite. Nous ne pourrons donc pas enrichir notre connaissance de cette probabilité qui est infime par cette méthode.
Cette situation mérite t-elle un tel acharnement de méthodes, toutes plus infructueuses les unes que les autres ? Qui peut le dire... Pour le plaisir, ou à titre d'exercice, j'ai envie d'essayer une autre méthode de simulation : générer des échantillons de taille variable contenant au moins une séquence recherchée. Je m'explique, il s'agit de tirer des lettres au hasard jusqu'à obtenir les six lettres de « COGITO » (ce pourrait être n'importe quelle séquence). Les lettres sont oubliées au fur et à mesure des tirages, on n'en garde à chaque fois que cinq pour tester la présence de la séquence lorsqu'on tire la 6ème lettre. Pour pouvoir calculer une moyenne acceptable, nous devons recommencer de nombreuses fois ce dispositif qui n'est pas destiné à évaluer, même indirectement, p_n pour une valeur particulière de n. On arrivera plutôt à un nombre que l'on peut évaluer par cette approche intuitive : le « C » arrive 1 fois sur 26 en moyenne, après le « C » on aura un « O » dans 1 cas sur 26 en moyenne (soit la séquence « CO » dans 1 cas sur 262=676 en moyenne), etc. Cela nous conduit à penser que dans 1 cas sur 26^6, soit dans 1 cas sur 308 915 776, on trouvera le mot « COGITO » écrit. Il faudra donc attendre trois millions de caractères en moyenne pour, enfin, obtenir la séquence tant attendue... Non, finalement je renonce à ce projet qui ne répond plus du tout à la problématique d'origine.

f) Pour finir notre incursion au pays des fonctions récursives, et pour montrer que leur domaine d'application n'est absolument pas limité aux nombres, laissons nous aller à dessiner la *courbe du dragon*. Cette courbe fractale a de nombreuses propriétés comme celle, inattendue, de paver le plan avec des répliques d'elle-même. Pour la construire, on part d'un segment à la 1ère étape, que l'on remplace par un chevron (si le segment initial est la diagonale d'un carré, le chevron est constitué par un des demi-carrés que découpe cette diagonale), et on fait ensuite de même avec les deux nouveaux segments. Les chevrons nouvellement créés sont alternativement tournés de part et d'autre de la courbe dont ils sont issus. Pour mieux comprendre cela, on peut observer le début de la construction sur l'illustration ci-dessous. Si D symbolise un virage à Droite et G un virage à Gauche : à la 2ème étape on a G, à la 3ème on a GGD (nous indiquons avec la couleur verte la séquence précédente, et avec la couleur rose la séquence doublement inversée qui s'ajoute, après un G médian). La séquence GGD devient, à la 4ème étape, GGDGGDD. La 1ère inversion (le début devenant la fin) de GGD conduit à DGG, et il s'ajoute une 2de inversion (les D se changeant en G) qui donne finalement le GDD final. Le processus continue ainsi jusqu'à l'infini, du moins en théorie, mais nous voulons obtenir l'étape n.
Voici le programme récursif qui a réalisé ces courbes pour les valeurs n=1, 2, 3, 4 et 5. Il est très largement inspiré du programme proposé par Didier Müller sur son site *nymphomath.ch*. La fonction récursive *dragon*() prend deux arguments : le 1er est l'indice d'étape et le 2d est un nombre qui oriente les angles de rotation (1 oriente dans un sens et −1 oriente dans l'autre). La longueur du segment initial étant divisée par le coefficient $\sqrt{2}$ à chaque étape, nous calculons dès le départ la longueur du segment final ($\frac{500}{\sqrt{2^n}}$) puisque, de toutes les façons, ce n'est qu'à la fin du processus récursif que la courbe est tracée. Pour que le dragon soit toujours orienté de la même façon (sur ses pattes), nous déterminons l'angle initial selon la valeur de n.
Cette construction récursive crée des redondances en lançant deux fois la courbe *dragon*(n−1,±1) car, si on lance *dragon*(10,1), cela renvoie *dragon*(9,1) et *dragon*(9,−1) qui, respectivement, lancent à leur tour *dragon*(8,1) et *dragon*(8,−1) soit deux *dragon*(8,1) et deux *dragon*(8,−1). Il y aura ensuite quatre *dragon*(7,1) et quatre *dragon*(7,−1), etc.
Nous voudrions réécrire une fonction qui traduise la propriété notée dans l'énoncé : à une séquence de segments donnée (l'étape n−1), on ajoute la même séquence doublement inversée (après avoir inséré une rotation supplémentaire de même nature que la 1ère rotation). La fonction *inverse*() réalise cela : elle prend en argument une liste d'éléments pris dans l'ensemble {0;1} qu'elle inverse doublement selon le schéma

prescrit. Les angles sont juste stockés sous la forme d'un indicateur qui est converti en degré par l'opération (a×2−1)×90 Ainsi 0 donne −90° alors que 1 donne +90°. Cette méthode ne crée pas de redondance (une seule fonction appelée), mais son inconvénient est d'encombrer la mémoire avec une liste d'indicateurs d'angles de plus en plus longue. L'angle qui est ajouté au début de la liste (sous la forme de l'indicateur assez mystérieux (3-*n*)/4 est destiné à donner l'orientation du 1er segment tracé afin que, comme précédemment, le dragon repose sur ses pieds.