Cours et Exercices de Python

niveau 1ère et Tale S

Énoncés des questions

Les questions qui suivent n'ont d'autre but que de vous donner quelques sujets de réflexion pour écrire des programmes Python. Les situations à programmer sont potentiellement infinies. Celles-ci ont le mérite d'avoir été corrigées : vous trouverez mes « propositions de correction » avec les commentaires les accompagnant ainsi que les programmes dans la même page de mathadomicile que ce document.

Partie 1: Primalité

a) Écrire un programme qui détermine si un nombre entier *n* est premier (si *n* n'a que deux diviseurs) ou s'il est composé. Dans ce dernier cas, donner la décomposition en facteurs premiers de *n*.

b) Utiliser ce programme pour montrer qu'à partir de *k*=5 les nombres de Fermat $F_k = 2^{2^k} + 1$ ne sont pas premiers.

c) Montrer, de même, que pour *k*=2, 3, 5, 7 les nombres de Mersenne, notés $M_k = 2^k - 1$, avec *k* premier, sont premiers (on les note alors *M1, M2, M3* et *M4*), alors que pour *k*=11, M_k n'est pas premier. Déterminer la valeur de *k* pour le 5ème nombre de Mersenne premier, noté *M5*.

d) Il est possible de se focaliser sur la primalité des nombres de Mersenne car un théorème dû au mathématicien français Lucas (1842-1891), permet d'accroître notablement l'efficacité du test : étant donnée la suite (s_n) définie par $s_1=4$ et, pour tout $n>1$, $s_n = (s_{n-1})^2 - 2$, le nombre $M_n = 2^n - 1$ est premier si et seulement si il divise s_{n-1}. La difficulté ici va être la croissance extrêmement rapide du nombre de chiffres des termes de la suite (s_n). Par exemple, pour tester si $M_{11} = 2^{11} - 1 = 2047$ est premier, il suffit de tester si ce nombre divise s_{10}. Le problème est que s_{10} s'écrit avec près de 300 chiffres ! Essayer de voir jusqu'où, avec Python sur une machine ordinaire et dans des temps raisonnables, cette simple suite permet d'examiner la primalité des nombres de Mersenne.

Partie 2: Liste de nombres premiers

a) Écrire un programme qui détermine la liste des nombres premiers inférieurs ou égaux à un entier *n* donné. On pourra afficher proprement cette liste (tableau) et donner des informations sur chaque nombre premier (son rang, sa valeur, son résidu modulo 4, modulo 6 ou modulo 10, etc.) ainsi qu'un récapitulatif statistique (effectif total des nombres premiers, pourcentages par résidu dans les différents modulo, etc.). Pour ce faire, on peut partir de rien, et utiliser la méthode du crible d'Ératosthène (un algorithme qui part de la liste des nombres entiers compris entre 2 et un entier *n* donné, et qui raye successivement tous les multiples du premier nombre non rayé, puis du 2ème nombre non rayé, etc.).

b) On peut aussi partir d'une première liste des premiers nombres premiers fournie en argument, et déterminer ceux qui manquent par une adaptation de la méthode précédente.

c) Application n°1 : Une représentation graphique amusante et instructive des nombres premiers consiste à enrouler ceux-ci à la manière d'un escargot autour d'un premier nombre (le germe). L'image ci-contre nous

donne une idée de ce que l'on veut obtenir (le germe y est égal à 7). Pour bien identifier les nombres premiers des autres, nous allons dessiner un gros point de couleur pour les nombres premiers et un plus petit point d'une couleur différente pour les autres. Pour se donner quelques repères, on peut écrire les valeurs des nombres premiers ou seulement celles des nombres de la forme $10k+1$ (un quart des nombres premiers) par dessus le point correspondant.

d) Application n°2 : Lorsqu'on additionne tous les diviseurs stricts (inférieurs au nombre) d'un nombre n, on fabrique un nouveau nombre n' qui peut lui-même suivre le même sort : on additionne tous ses diviseurs pour fabriquer un 3ème nombre n'', etc. Si n=10, les diviseurs stricts de 10 étant 1, 2 et 5, on fabrique le nombre n'=1+2+5=8, puis, comme les diviseurs stricts de 8 sont 1, 2 et 4 on fabrique le nombre n''=1+2+4=7. Comme 7 est premier, on se retrouve au nombre n'''=1 et on s'arrête là car 1 n'a pas de diviseur strict. Il se trouve qu'en essayant avec plusieurs valeurs n de départ, on s'arrête presque toujours sur 1. Prouver cela en écrivant un programme qui affiche cette suite de nombres partant d'un nombre n quelconque. Explorer le comportement des premiers entiers : longueur de la suite et sens de variation. Essayer de repérer les exceptions à la règle énoncée.

Partie 3: Le petit théorème de Fermat

Ce théorème a été souvent utilisé pour montrer qu'un nombre n'est pas premier, voici comment. Ce théorème s'énonce en disant que, si p est un nombre premier et $a \geq 2$ un nombre entier non divisible par p, alors $a^{p-1}-1$ est un multiple de p.

a) Le nombre 7 est premier donc quelque soit l'entier $a \geq 2$ non divisible par 7, on doit avoir a^6-1 divisible par 7. Vérifier cela, en testant la divisibilité de a^6-1 par 7 pour toutes les valeurs inférieures à 14 qui ne soient pas dans la table de 7. Recommencer le même travail pour d'autres nombres premiers inférieurs à 100.

b) Pour prouver qu'un nombre n n'est pas premier, il suffit de montrer que $a^{n-1}-1$ n'est pas divisible par n (c'est la forme contraposée du théorème) pour au moins une valeur de $a \geq 2$ non divisible par n. Par exemple, 91 n'est pas premier car $2^{90}-1=1237940039285380274899124223$ n'est pas divisible par 91 (le reste est 63). Et, en effet, $91=7\times13$, 91 est un nombre composé (on aurait pu le savoir bien plus facilement, sans utiliser ce théorème).

Mettre au point un algorithme qui teste, avec cette méthode la non-primalité d'un entier n. On pourra par exemple tenter de prendre a=2 et montrer que $b=2^{p-1}-1$ n'est pas divisible par n (utiliser la fonction % : reste de la division euclidienne), si ce n'est pas le cas, on tente la division par le nombre premier suivant, etc. Tester, par exemple, la non-primalité de $10^3+1=1001$ puis de $10^7+1=10000001$.

c) Par contre, on ne peut utiliser directement la réciproque qui est fausse généralement : ce n'est pas parce qu'il existe un entier $a \geq 2$ premier avec l'entier impair n tel que $a^{n-1}-1$ soit un multiple de n (on écrit cela aussi $a^{n-1} \equiv 1 (mod\ n)$, que n est un nombre premier... De tels nombres impairs n sont appelés *nombres pseudo-premiers de base a* (on précise parfois que ce sont des nombres pseudo-premiers de Fermat car il existe d'autres sortes de nombres pseudo-premiers). Si a est égal à 2, et si a est quelconque (pas nécessairement premier avec p) de tels nombres pseudo-premiers sont appelés *nombres de Poulet*. Si la propriété est vérifiée pour tout entier a compris entre 2 et n, le nombre est appelé *nombre de Carmichael*.

Retrouver, à l'aide d'un algorithme, les dix premiers nombres de Poulet (la liste commence par 341, 561, 645, 1105 et 1387) et les dix premiers nombres de Carmichael (la liste commence par 561, 1105 et 1729).

Quels sont les premiers nombres pseudo-premiers de base 2, 3, 4, 5, etc. qui soient supérieurs à leur base ? (pour 2, 3 et 4 la réponse est 341, 91 et 15).

Partie 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).

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.

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.

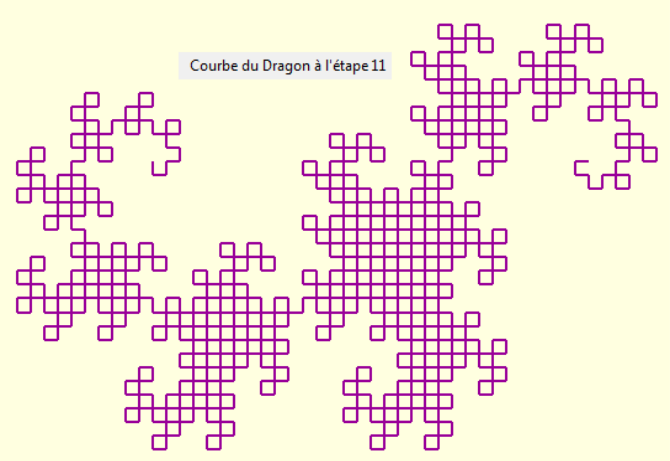
En déduire un programme qui calcule les coefficients binomiaux $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, aussi appelés (en dénombrement) nombres de combinaisons de k éléments pris parmi n, ou coefficients du triangle de Pascal.

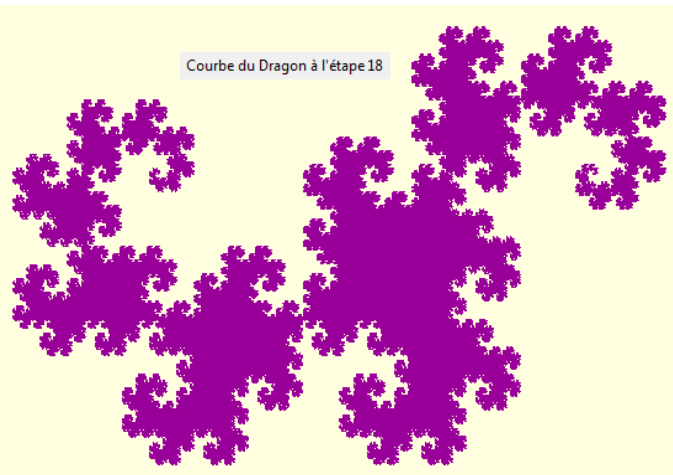
Écrire ces deux programmes en mode récursif (utilisant une fonction qui s'appelle elle-même).

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.

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.

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





1 Cette situation d'*un singe dactylographe*, imaginée par Émile Borel au début du XXe, 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.

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*.

Partie 5: Modules et classes

Les fractions permettent d'écrire les nombres rationnels à partir de deux nombres entiers, le numérateur et le dénominateur, le second devant être non nul. Parmi toutes les fractions égales, il existe une unique fraction plus simple que toutes les autres : la *fraction irréductible*. Les nombres rationnels ont une autre propriété : tous ont une écriture décimale qui est périodique à partir d'un certain rang. Pour $\frac{2}{3}=0{,}66666...$, c'est le chiffre 6 qui se répète à partir du rang des dixièmes (−1), précédé par le nombre 0 ; pour $\frac{13}{11}=1{,}181818...$ c'est la suite de chiffres 18 qui se répète à partir du rang des dixièmes (−1), précédée par le nombre 1 ; pour $\frac{15}{14}=1{,}071428571428571...$, c'est la suite de chiffres 714285 qui se répète à partir du rang des centièmes (−2), précédée par le nombre 1,0 (le 0 est important ici).

a) Écrire les fonctions nécessaires à l'obtention de l'écriture irréductible d'un nombre rationnel donné par deux nombres *a* et *b* (le numérateur et le dénominateur de la fraction) ainsi que celles donnant la suite de chiffres se répétant, le rang à partir duquel cette suite apparaît dans la partie décimale et la partie précédent cette suite (appelé partie non périodique du quotient). Toutes ces informations doivent pouvoir être obtenues à partir d'une classe que vous appellerez « Frac » (on ne se sert pas encore de la classe « Fraction » qui existe déjà dans Python).

b) Notre classe *Frac* peut s'étoffer encore un peu : nous voulons pouvoir obtenir l'écriture de notre fraction sous sa forme de fraction continue. Par exemple $\frac{105}{8}=13+\frac{1}{8}$ ou $\frac{51}{7}=7+\frac{1}{3+\frac{1}{2}}$. On notera ces fractions avec une notation linéaire [13,8] et [7,3,2]. Ainsi, toutes les fractions peuvent s'écrire sous la forme d'une liste finie [a_0, a_1, a_2, …, a_n], cette notation ayant un intérêt qui dépasse largement ce niveau anecdotique. Comment obtient-on les coefficients de cette notations : avec l'algorithme d'Euclide (encore), les quotients partiels de chaque étape de la recherche du PGCD nous les fournit : 51=7×7+2 (donc a_0=7) puis 7=2×3+1 (donc a_1=3) et enfin 2=1×2+0 (donc a_2=2). Une fois définie cette fonction *fracContinue*(), nous voudrions pouvoir instancier un objet de la classe *Frac* avec une liste d'entiers. Par exemple, nous voudrions pouvoir faire *a=Frac*([7,3,2]) et que dans *a.num* on trouve 51.

c) Notre classe *Frac* peut sans doute accepter encore un autre type de déclaration : nous voulons pouvoir définir une fraction par la partie non-périodique de son développement décimal et la suite de chiffres qui se répète (deux chaînes de caractères). Par exemple *a=Frac*("2.1","6") doit être interprété comme le nombre 2,16666... qui est égal à la fraction $\frac{13}{6}$. Pour réaliser cela, on calcule $10^1 a-a=9a$ (1 car il n'y a qu'un seul chiffre qui se répète) qui vaut autant que 21,6−2,1 (les chiffres d'après sont identiques dans les écritures décimales) soit 19,5. Il ne reste plus qu'à simplifier $\frac{19{,}5}{9}=\frac{195}{90}=\frac{13\times15}{6\times15}=\frac{13}{6}$. Une fois que la classe *Frac* est au point, la transposer comme une classe héritée de la classe *Fraction* du module *fractions* de Python (*from fractions import Fraction*).

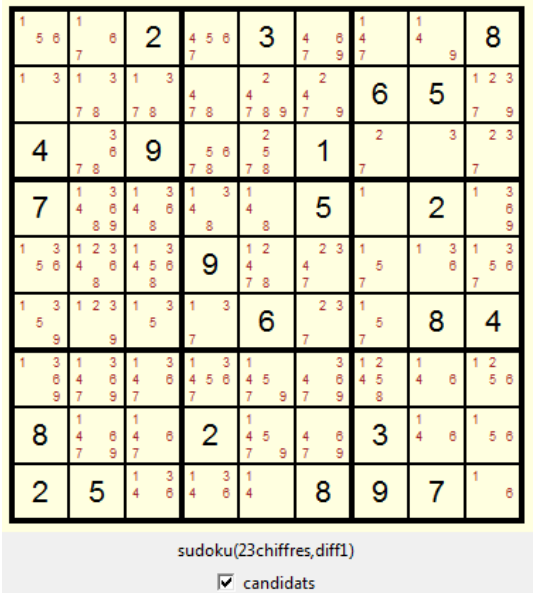
d) Pour compléter notre étude des classes Python et enrichir le domaine de nos investigations (il n'y a pas que les nombres!), intéressons nous à un jeu. Il est assez simple de résoudre un sudoku quand on envisage toutes les possibilités, les unes après les autres. L'algorithme de *backtracking* réalise cela : on essaie méthodiquement le remplissage de la grille (sans réfléchir), chiffre après chiffre, et si cela conduit à une impasse, on revient au point précédent où l'on augmente le chiffre, jusqu'à ce qu'on ai essayé toutes les possibilités, dans ce cas on revient encore plus en arrière, etc. En principe une bonne grille de sudoku ne doit avoir qu'une seule solution, nous la cherchons. L'objectif fixé ici : la grille initiale (une liste de 81 éléments donnée par un fichier) devient une instance de la classe *Grille* qui contient les méthodes nécessaires à la recherche de la solution et à son affichage.

e) Vous noterez que l'algorithme employé ici pour résoudre les sudokus n'est pas du tout intelligent : il essaie bêtement toutes les possibilités jusqu'à trouver une solution. S'il n'y a en a pas, il va produire une erreur (rien n'est prévu pour ce cas) et s'il y en a deux ou plus, il ne nous le dira pas (et c'est pourtant disqualificatoire pour un sudoku). Nous voulons améliorer notre programme et compter les solutions (aller,

tout aussi bêtement, jusqu'au bout du processus). Pour prolonger cette étude, nous pouvons essayer de générer une grille de sudoku valide (a une solution unique) et dont le nombre de chiffres de la grille initiale n'est pas trop grand (entre 17 qui est la valeur minimum et 25, une valeur raisonnable arbitraire) .

Partie 6: Interactivité et widgets graphiques

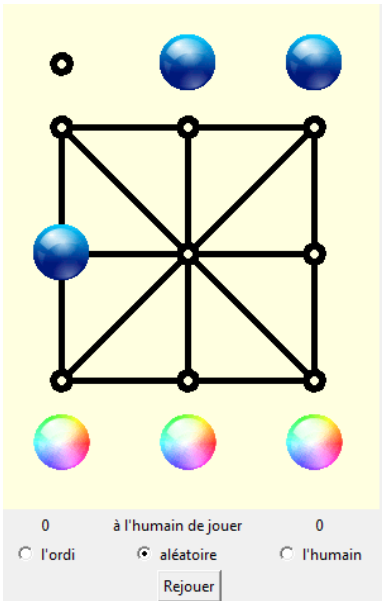
Nous avons déjà rencontré quelques situations interactives – où l'utilisateur peut converser avec le programme – quand nous avons utilisé le module *tkinter* et découvert quelques uns de ses nombreux dispositifs graphiques. Mais l'interactivité est également présente dans un simple programme qui demanderait à l'opérateur d'entrer son nom jusqu'à ce que le nom entré ne contienne aucun chiffre. Cela pour dire que la notion d'interactivité recouvre des fonctionnalités très diverses, pas forcément compliquées, qui permettent de moduler les actions du programme.



a) Pour continuer avec les sudokus, nous aimerions disposer d'un programme qui affiche une grille initiale et offre la possibilité de la compléter avec des chiffres entrés au clavier. La fenêtre d'affichage doit être rendue sensible (la méthode *bind*() de la classe *Tk* réalise cela) et réagir aux entrées du clavier en lançant un évènement (*KeyPress*) qui permet à la commande *ajoute*() de se servir du chiffre entré. Le programme doit reconnaître une grille complète et valide en affichant le message « Bravo ! La solution a été trouvée en … secondes »). On peut prévoir aussi une case à cocher *Checkbutton*() qui, lorsqu'elle est cochée, déclenche l'affichage des chiffres « candidats » dans les cases vides.

b) Changeons de sujet et faisons un pas de plus dans le monde merveilleux des fractales ! Les ensembles de Julia sont des images fractales construites selon un procédé assez simple. À chaque pixel P de coordonnées $(x;y)$ nous allons associer une couleur $coul(r,b,v)$ selon la 1ère valeur de l'entier n pour laquelle l'image P_n du point P s'écarte de l'origine $O(0;0)$ du repère d'une différence supérieure ou égale à 2. Pour trouver les coordonnées $(x';y')$ de l'image P_n, à partir de celles $(x;y)$ de P_{n-1} on applique les relations : $x'=x^2-y^2+x_I$ et $y'=2xy+y_I$ où $(x_I\,;y_I)$ sont les coordonnées d'un point I du plan. Lorsque le point P_n ne sort pas (en essayant les valeurs de n jusqu'à une certaine valeur maximum appelée *prof* pour profondeur) du disque de rayon 2 centré sur O, la couleur du point est noire, sinon la couleur est définie par un choix arbitraire, par exemple le dégradé de vert $coul(0,\frac{(prof-n)\times 255}{prof},0)$. En procédant ainsi, on obtient l'image de l'ensemble de Julia associé au point I. Programmer l'affichage d'une telle image pour x et y compris entre -2 et 2, avec la possibilité de modifier les coordonnées de I et la possibilité de zoomer sur une partie de cette image (par exemple en cliquant dans le cadre deux extrémités de la nouvelle fenêtre en diagonale).

c) Donnons-nous un objectif un peu plus ludique que la simple exploration d'une image statique, fusse t-elle magnifique, et un peu plus ambitieux que la programmation du sudoku où les contraintes ne laissent aucune possibilité de variation. Un jeu simple à deux joueurs dont l'un des joueurs est l'ordinateur et qui nécessite une forme d'intelligence de sa part : le jeu des neuf trous appelé aussi *tapatan*. Le plateau de jeu est une grille carrée de 3 sur 3 et chaque joueur dispose de trois pions qu'il doit aligner sur le plateau. Au début, chacun à tour de rôle pose un pion, n'importe où sur le plateau. Une fois les six pions posés, chacun peut déplacer un de ses pions d'une position, selon un des tracés figurés sur le plateau (le long des six lignes ou des deux diagonales), si le nouvel emplacement est libre. Programmer ce jeu avec un compteur des parties gagnées/perdues et un dispositif permettant de choisir si c'est le joueur qui commence (plus facile), si c'est l'ordinateur (plus difficile) ou bien si ce choix est laissé au hasard (équilibré). Dans un 1er temps, on peut se consacrer à la réalisation du jeu, sans chercher une réponse intelligente de la part de l'ordinateur (rendre ses choix aléatoires).



d) Pour finir en beauté, envisageons une application graphique qui utilise un menu. Fixons un but à ce programme : réaliser des rosaces, des figures ayant *n* axes de symétrie concourants (*n* est variable, supérieur ou égal à 2). Les « pinceaux » ont une couleur et une taille réglables. On trace des lignes de plusieurs types (des segments, des arcs de cercles, des lignes libres, etc.) et on doit pouvoir gommer (supprimer des éléments). Dans un 2ème temps, on peut prévoir un enregistrement de l'image ainsi crée au format *png* ou une exportation de cette image dans le « presse-papier ».