Épreuve orale de « *Mathématiques et algorithmique* » de la Banque PT – Rapport 2019

Les futurs candidats trouveront dans ce rapport des remarques et des conseils qui pourraient leur être utiles pour leur futur passage. Ce rapport n'est pas exhaustif et ne met l'accent que sur quelques points jugés importants par l'équipe d'interrogateurs de cet oral. Nous suggérons aux futurs candidats de consulter le **site de la Banque PT**, où ils pourront trouver le mémento *Python* fourni lors de l'oral, les exercices types d'informatique, ainsi que les rapports des années antérieures comportant à la fois des informations complémentaires en regard du présent rapport et des exercices qui ont été posés lors de sessions antérieures, à titre d'exemples.

1 – Objectifs

Le but d'une telle épreuve est d'abord de contrôler l'assimilation des connaissances des programmes de mathématiques et d'informatique (items 2, 3 et 5) de toute la filière (première et deuxième années), sans oublier celle des connaissances de base du programme des classes du lycée (seconde, première, terminale).

Cette épreuve permet aussi d'examiner :

- l'aptitude du candidat à lire attentivement un sujet et à répondre précisément à la question posée ;
- son aisance à exposer clairement ses idées avec un vocabulaire précis ;
- sa capacité d'initiative et son autonomie et, en même temps, son aptitude à écouter l'interrogateur, à prendre en compte ses indications, à lui demander des précisions si besoin ;
- son aptitude à mettre en œuvre ses connaissances et son savoir-faire pour résoudre un problème (par la réflexion et non par la mémorisation de solutions toutes faites) ;
- sa maîtrise des algorithmes et manipulations de base, des calculs sur des nombres entiers, décimaux ou complexes, et du langage de programmation pour mettre en œuvre une solution informatique ;
- sa faculté à critiquer, éventuellement, les résultats obtenus et à changer de méthode en cas de besoin.

2 – Modalités de cette épreuve

La durée de cet oral de « *Mathématiques et algorithmique* » est de 1 heure, préparation incluse.

Il comporte deux exercices de durées comparables :

- l'un porte sur le programme de mathématiques des deux années de la filière PTSI/PT (algèbre, analyse, géométrie et probabilités) et se déroule au tableau ;
- l'autre exercice porte sur les items 2, 3 et 5 du programme d'informatique et se déroule sur ordinateur. Pour ce deuxième exercice, les candidats disposent d'un ordinateur (Windows 10, clavier français Azerty) dans lequel sont installés *Python* 3.6 et ses principales bibliothèques (dont `numpy`, `scipy`, `matplotlib`, `random`, aides incluses) [1] [2], d'un mémento plastifié en couleurs au format A3, et de feuilles de brouillon, qu'il ne faut pas hésiter à utiliser. **L'environnement de développement** est *IDLE*, comme annoncé depuis 2014, muni de l'extension ***IDLEX*** qui permet notamment d'afficher plus clairement les numéros de ligne, de faire exécuter une partie d'un programme seulement (F9 au lieu de F5), ou de rappeler dans la console une commande déjà saisie (flèches montante et descendante). Quelques candidats ont avoué avoir préparé l'oral avec *Spyder*, *Pyzo* ou autre, ce qui est un peu surprenant. Nous ne pouvons que conseiller de se placer dans les conditions de passage de l'oral tout au long des deux années de préparation.

1. Distribution Anaconda (voir par exemple **Formations Python 3 Arts et Métiers**).
2. *Scilab* 5.5 est également installé mais, depuis 2015, aucun candidat n'a demandé à programmer en *Scilab*.

3 – Organisation

Cette dernière session s'est déroulée dans des conditions identiques aux sessions précédentes. Comme les autres années, elle a eu lieu dans les locaux de « *Arts et Métiers ParisTech* », 155 boulevard de l'Hôpital à Paris (13^{e}). En raison du plan Vigipirate, il n'a pas été possible cette année d'accueillir de futurs candidats ; les enseignants de classes préparatoires peuvent assister à un oral, après en avoir fait la demande au Service Concours ; cette année, personne n'a fait cette démarche.

4 – Conseils généraux

Lors d'une épreuve orale, le candidat doit être extrêmement vigilant :

- Lire attentivement le sujet et bien écouter une question permet de répondre à la question effectivement posée ; lire une phrase dans son intégralité (du premier mot au point final) peut s'avérer extrêmement profitable ; trop souvent, c'est pour n'avoir pas vu un mot, un seul, que l'on passe à côté d'une question.
- Écouter les consignes de l'interrogateur est en général utile ; il vaut mieux attendre qu'il ait terminé avant de répondre.
- Lorsqu'une indication est donnée pour aider le candidat, il faut savoir l'écouter et réagir à celle-ci, par exemple en la reformulant pour vérifier qu'on l'a bien comprise.
- La capacité du candidat à s'exprimer clairement avec un vocabulaire précis est évidemment un critère important d'évaluation.

Ces capacités d'attention, d'écoute et de réaction sont des éléments d'évaluation. De manière générale, la passivité, l'attentisme, le mutisme, ou l'obstination dans une voie infructueuse sont déconseillés lors de l'oral.

Les exercices posés sont tous issus de banques d'exercices sur lesquelles l'équipe d'interrogateurs travaille tout au long de l'année, notamment en faisant le bilan de chaque session d'oraux. Ces exercices sont de longueurs variables et assez souvent trop longs. Il est donc important de rappeler que l'objectif poursuivi est l'évaluation par l'interrogateur des capacités de chaque candidat grâce à l'exercice proposé, et non pas que le candidat termine nécessairement l'exercice.

L'oral, contrairement à une « *colle* », ne sert qu'à évaluer les capacités du candidat et non plus à participer à sa formation ; des indications seront en général données par l'interrogateur si le candidat reste bloqué trop longtemps, ou si celui-ci demande de l'aide par des questions dont il reconnaît implicitement ignorer la réponse (exemples : « *Est-ce que je peux utiliser tel théorème ?* », ou « *Pourquoi la figure ne s'affiche-t-elle pas ?* »).

Il est évidemment préférable, lorsqu'on sollicite de l'aide, d'expliquer les pistes envisagées et les raisons pour lesquelles elles ne semblent pas déboucher, plutôt que de se contenter de dire « *Je ne vois pas.* » ou « *Ça ne marche pas.* ».

Quelques détails utiles en mathématiques comme en informatique :

- Une bonne maîtrise des nombres complexes, de leurs différentes représentations (tant mathématique qu'informatique) et de leur manipulation est requise ; leur utilisation et leur manipulation en tant qu'affixes de points du plan, permettant d'éviter de revenir systématiquement aux coordonnées, peut s'avérer très efficace (exemples : affixe du milieu de deux points, distance entre deux points) ; les interprétations géométriques du module, de l'argument, des parties réelles et imaginaires, du conjugué d'un nombre complexe doivent donc être connues.
- En géométrie dans le plan, on doit être capable de construire et/ou de manipuler les coordonnées de points et de vecteurs, de calculer la longueur d'un segment (en repère orthonormé), les coordonnées des sommets d'un polygone usuel – en vue par exemple de faire tracer les côtés de ce polygone à

l'écran –, l'aire de polygones usuels (triangle, trapèze, carré, rectangle, parallélogramme) ; le rôle du déterminant de deux vecteurs $\overrightarrow{AB}$ et $\overrightarrow{BC}$ du plan (et aussi, dans l'espace, de leur produit vectoriel) est trop souvent méconnu pour caractériser l'alignement des 3 points, la colinéarité des vecteurs, la surface du parallélogramme $ABDC$ et, conséquemment, celle du triangle ABC.

5 – Conseils pour l'exercice de mathématiques

5.1 – Généralités

- L'oral n'est pas un écrit sur tableau ; les justifications et commentaires doivent être donnés au moment où l'on est interrogé ; le temps étant limité, il est inutile d'écrire de longues phrases, notamment pour justifier une linéarité ou une continuité triviales, et encore moins de recopier l'énoncé que l'interrogateur et le candidat connaissent tous les deux.
- Le candidat doit être précis dans ses propos, et, en particulier lorsqu'il énonce une définition, une propriété ou un théorème au programme de mathématiques, il doit énoncer l'ensemble des hypothèses sans en oublier ; le jury attend d'un candidat qu'il connaisse les résultats de cours.
- Un exercice de mathématiques ne peut se résumer à l'application d'une recette toute faite ; au lieu de se précipiter vers l'utilisation d'un théorème, d'une règle ou d'une technique, chaque candidat pourra se poser la question : « *L'application de la définition ou un calcul élémentaire ne suffisent-ils pas à fournir une solution ?* » (Exemples : $\phi(\mathbf{v}) = \lambda\,\mathbf{v}$ pour la recherche d'une valeur propre λ et d'un vecteur propre $\mathbf{v}$ pour l'application linéaire ϕ, calcul de la somme partielle pour étudier une série, calcul de l'intégrale dépendant d'un paramètre, dérivation de $x \mapsto \int_a^x f(t)\,\mathrm{d}t$).
- On attend également d'un candidat qu'il maîtrise les techniques de calcul en connaissant les concepts sous-jacents ; par exemple, maîtriser le procédé de calcul puis de recherche des racines du polynôme caractéristique ne dispense pas de connaître les définitions de valeur propre et de sous-espace propre ; lorsque plusieurs procédés de calcul sont possibles, par exemple pour la résolution d'un système linéaire ou la détermination du rang d'une matrice (méthode du pivot, substitution, combinaisons linéaires, etc.), le candidat peut utiliser celui qu'il préfère à condition d'être efficace.
- Parmi les recettes toutes faites, il a été souvent observé l'utilisation de techniques qui ne sont pas explicitement dans les programmes (calcul de l'enveloppe d'une famille de droites par résolution du système linéaire $\{a(t)\,x + b(t)\,y + c(t) = 0, a'(t)\,x + b'(t)\,y + c'(t) = 0\}$, étude de $\alpha^n\,u_n$ pour la convergence d'une série, règle de d'Alembert pour déterminer le rayon de convergence d'une série entière – rappelons au passage que la règle de d'Alembert n'est pas la panacée –, etc.) ; dans ce cas-là, l'interrogateur sera particulièrement pointilleux quant à la justification détaillée des affirmations du candidat.
- Les candidats doivent s'attendre à être interrogés sur la nature des objets qu'ils manipulent ; ils doivent pouvoir dire s'ils manipulent un nombre, une fonction, un vecteur ; par exemple, il n'est pas acceptable à ce niveau de confondre intégrale et primitive.

5.2 – Polynômes à coefficients réels et complexes

- On n'est pas obligé de passer par un calcul de discriminant pour résoudre $x^2 + 4 = 0$ ou $x^2 - 2\,x + 1 = 0$; cela permettra de disposer de plus de temps pour traiter les autres questions.
- Les racines n-ièmes de l'unité et leurs propriétés, notamment leur somme et leur représentation géométrique, doivent être connues.

5.3 – Algèbre linéaire

- En algèbre comme ailleurs, on doit veiller à utiliser un vocabulaire précis et à éviter les confusions ; cette année, assez bizarrement, de trop nombreux candidats devant démontrer la linéarité d'une

application ϕ ont vérifié que $\phi(O_E) = O_F$, montrant ainsi une confusion avec la notion de sous-espace vectoriel.

- Les notions liées aux sous-espaces vectoriels (s.e.v. supplémentaires, s.e.v. engendrés par une famille de vecteurs, etc.) doivent être mieux connues.
- Les liens entre les notions de valeur propre, de rang, de noyau, gagneraient en général à être mieux assimilés ; par exemple, les équivalences entre $\det(A) \neq 0$ et $\ker(A) = \{O_E\}$, entre $\dim(\ker(A)) \geqslant 1$ et « 0 *est valeur propre de* A », entre « le vecteur non nul $\mathbf{u}$ est invariant par l'endomorphisme f » et « $\mathbf{u}$ *est vecteur propre de* f *pour la valeur propre* 1 ».
- Le calcul littéral sur les matrices et les vecteurs doit être maîtrisé, pour caractériser par exemple une matrice symétrique, une matrice orthogonale, un vecteur propre d'une matrice et la valeur propre associée, un produit scalaire associé à une matrice.

5.4 – Analyse

- Les candidats qui pensent à utiliser un développement limité à bon escient, notamment lorsqu'un simple équivalent ne suffit pas, sont en général positivement évalués ; il est par conséquent conseillé de connaître les développements limités usuels (comme celui de $x \mapsto (1+x)^\alpha$ au voisinage de 0, par exemple).
- L'écriture $\lim\limits_{x\to a} f(g(x)) = f\left(\lim\limits_{x\to a} g(x)\right)$ doit être justifiée clairement, même si la fonction f est une fonction usuelle.

5.5 – Intégration

- Lorsqu'on étudie l'intégrabilité d'une fonction sur un intervalle, penser à regarder en premier lieu si celle-ci est continue sur l'intervalle fermé ou, à défaut, sur l'intervalle ouvert, avant de détailler les problèmes éventuels aux bords.
- De trop nombreux candidats mélangent le *Théorème fondamental du calcul intégral* et les théorèmes sur les intégrales dépendant d'un paramètre.

5.6 – Suites et séries

- Les suites récurrentes doivent être maîtrisées, ce qui est heureusement souvent le cas mais pas toujours, avec notamment une confusion trop fréquente entre l'écriture du terme général d'une suite définie par une récurrence multiple et l'écriture de la solution générale d'une équation différentielle ordinaire linéaire homogène à coefficients constants.
- Les séries géométriques doivent être parfaitement maîtrisées, ce qui est heureusement très souvent le cas.
- L'écriture $\lim\limits_{n\to\infty} f(u_n) = f\left(\lim\limits_{n\to\infty} u_n\right)$ doit être justifiée clairement, même si la fonction f est une fonction usuelle.

5.7 – Géométrie dans le plan

- De nombreux sujets de géométrie sont posés, y compris parmi les exercices d'informatique. C'est une particularité de la filière PT. Il est plus que conseillé de faire un dessin lisible ; cela permet de mieux comprendre le sujet, et est très apprécié par les examinateurs.
- Les sujets de géométrie utilisent fréquemment la trigonométrie ; il convient donc de pouvoir donner rapidement les formules utiles à l'exercice, et aussi d'être capable d'étudier des fonctions trigonométriques simples, qui paramètrent souvent les courbes.
- Il faut surtout que les candidats, au lieu de se précipiter sur les calculs, mettent en place une

démarche de résolution et annoncent à l'examinateur la liste des tâches pour arriver à la solution du problème posé.

- Trop peu de candidats ont réussi à mener à bien l'étude d'une courbe paramétrée, vraisemblablement par manque de pratique ; la réduction du domaine d'étude et la mise en évidence de symétries doivent être maîtrisées, ainsi que l'étude des points singuliers, ce qui est fort heureusement assez fréquent.
- Il sera apprécié qu'un candidat sache paramétrer simplement une conique définie par son équation cartésienne réduite.
- Comme indiqué en préambule, il en sera de même pour la signification géométrique du déterminant de deux vecteurs $\overrightarrow{AB}$ et $\overrightarrow{AC}$.

5.8 – Fonctions de plusieurs variables et géométrie des courbes et surfaces

Liées aux notions de champs, de courbes et de surfaces, les fonctions de plusieurs variables sont indispensables, notamment en ingénierie mécanique. En particulier, il est nécessaire de :

- savoir étudier leur continuité (ou plus généralement leur régularité $\mathcal{C}^1$) ;
- connaître la définition de ses dérivées partielles et savoir les calculer ;
- savoir utiliser la *règle de la chaîne* (dans le programme PT : « *Calcul des dérivées partielles d'ordres 1 et 2 de* $(u,v) \mapsto f(x(u,v),y(u,v))$ ») ;
- savoir déterminer les points critiques et leur nature ;
- savoir déterminer la tangente et la normale à une courbe ainsi que le plan tangent à une surface, à partir d'équations cartésiennes ou paramétriques.

Nous avons pu observer en 2019 une amélioration générale dans ce domaine.

En revanche, cela ne dispense pas d'être capable de donner des représentations cartésiennes et paramétriques d'éléments géométriques de base comme les droites, les plans, les cylindres ou les sphères.

5.9 – Équations différentielles linéaires

- La résolution d'équations différentielles linéaires à coefficients constants avec second membre doit être maîtrisée, ce qui est heureusement très souvent le cas ; avec hélas, comme évoqué plus haut, une confusion avec les suites à récurrence multiple.
- Les équations différentielles linéaires du premier ordre sans second membre et à coefficients non constants ont semblé particulièrement maîtrisées lors de la session 2019 ; ce qui constitue une nette amélioration sur ce point soulevé dans le rapport 2018.

5.10 – Probabilités

- Encore plus qu'ailleurs, il faut lire attentivement l'énoncé et être précis dans son vocabulaire ; un minimum de formalisme est attendu.
- On apprécie qu'un candidat justifie naturellement un résultat obtenu (probabilités totales, conditionnelles, etc.) et donne des définitions correctes, notamment celle de l'indépendance de deux évènements, ou de deux variables aléatoires. Savoir prononcer le terme « *système complet d'évènements* » est bien, mais il est nettement mieux d'être en mesure de détailler de quoi il s'agit.

6 – Exercice d'algorithmique/simulation numérique

Chaque année, les candidats sont de mieux en mieux préparés pour cet oral. Le nombre d'excellents candidats est en très forte augmentation en 2019 alors que les candidats incapables d'écrire la moindre ligne de programme informatique ont quasiment disparu.

Cependant, l'effort sur l'aspect « *simulation numérique* » et plus particulièrement sur l'utilisation des tableaux (dont vecteurs et matrices) de la bibliothèque **numpy** doit être poursuivi.

6.1 – Conseils généraux

- Lire attentivement l'énoncé ; il arrive très souvent que plusieurs phrases introductives présentent le contexte de l'exercice ; ne pas hésiter à solliciter l'interrogateur si on a le moindre doute, pour clarifier le problème et éviter tout contresens qui pourrait induire des réponses « *hors sujet* ».
- Sauf indication contraire de l'énoncé, **toutes les fonctionnalités de Python 3 sont autorisées** (fonctions `sum`, `max`, `min`, `sorted`, ..., l'instruction « `x in L` » qui donne un booléen indiquant si l'objet `x` est dans l'itérable `L` ou pas, etc.) ; cela ne dispense pas le candidat d'être capable de répondre s'il est interrogé sur un algorithme de base.
- Si quelques lignes de code sont proposées à la compréhension, il est conseillé au candidat de taper ce code et de le comprendre en modifiant certains paramètres ; expliquer un code n'est pas le lire mot à mot mais décrire globalement ce qu'il fait et à quoi il sert.
- Ne pas hésiter à utiliser le brouillon mis à disposition avant de se jeter trop rapidement dans la programmation ou pour décrire l'ébauche d'un algorithme à l'interrogateur.
- Ne pas négliger les premières questions : elles contiennent le plus souvent des éléments de réponse pour la suite, voire des rappels.
- Ne pas hésiter à utiliser le mémento, surtout si le conseil en est donné par l'interrogateur.
- Ne pas hésiter à utiliser la console (l'interpréteur) pour effectuer des vérifications élémentaires ou tester les fonctions de Python suggérées par l'énoncé.
- Il est indispensable de savoir utiliser les instructions `help` et `numpy.info` : il est normal de ne pas connaître toutes les fonctions apparaissant dans les exercices ; le nom de la fonction à utiliser est très souvent suggéré dans l'énoncé, notamment si cette fonction n'apparaît pas dans le mémento, et il faut donc savoir se renseigner à son sujet et faire des tests élémentaires dans la console.
- Il faut savoir mettre en œuvre une démarche en cas d'erreur : faire des tests élémentaires dans la console, insérer des `print` pour contrôler pas à pas une exécution, lire attentivement et savoir utiliser les messages d'erreurs (lecture de bas en haut, savoir par exemple que « `...index out of range` » est lié à un problème de numérotation dans un objet indexé, que « `...object is not callable` » indique un problème de parenthèses et que « `...object is not subscriptable` » indique un problème de crochets), etc. Il s'agit d'une compétence valorisée par le jury.
- Bien faire la distinction entre les entiers et les flottants, et maîtriser les conséquences induites, dont la différence entre l'opérateur `//` (division entière) et l'opérateur `/` (division flottante).
- La manipulation des entiers est indispensable en informatique et il est essentiel de connaître la numération en bases 10 et 2, ainsi que le passage de l'une à l'autre.
- La manipulation des chaînes de caractères fait aussi partie des capacités exigibles, et en particulier la connaissance des méthodes **split**, **strip**, **replace** qui peuvent être utiles pour la lecture de données structurées dans un fichier ASCII. En 2019, de trop nombreux candidats ne faisaient pas clairement la différence entre le nom `my_name` et la chaîne `'my_name'` (ou `"my_name"`), montrant ainsi une méconnaissance du rôle des apostrophes et des guillemets.

- L'effort doit être poursuivi dans la lecture d'un fichier texte se trouvant dans un sous-répertoire du répertoire courant ; le plus souvent, le candidat aura à extraire des données numériques à partir de ce fichier ; dans le cas où le fichier contient un texte comportant des lettres accentuées, il est systématiquement encodé selon la norme internationale et multiplateforme UTF-8 ; le rajout de l'option « `encoding='UTF8'` » lors de l'ouverture du fichier est alors en général indiqué dans l'énoncé ou, à défaut, par l'interrogateur ; ce détail ne peut entraîner aucune pénalité.
- La numérotation des éléments, le découpage et la concaténation des chaînes de caractères comme des listes doivent être aussi maîtrisés, dont l'utilisation de l'indexation négative qui ne nécessite pas de connaître le nombre d'éléments (`nom[-1]` pour le dernier élément, `nom[-2]` pour l'avant-dernier, `nom[-3:]` pour les trois derniers, etc.).
- Il n'est pas nécessaire de définir systématiquement une fonction pour chaque tâche demandée, et, plus généralement, il n'y a aucun style de programmation imposé ; le candidat est évalué sur la maîtrise des outils mis à sa disposition et non sur le respect dogmatique de telle ou telle règle ou interdiction le plus souvent arbitraire.
- En revanche, **une fonction doit toujours être testée**, soit dans l'éditeur (F5 ou F9), soit dans la console, comme cela est spécifié dans l'en-tête de chaque énoncé.
- Préférer une boucle `for` à un `while` quand le nombre d'itérations est connu à l'avance. Préférer également une boucle non indexée « `for objet in iterable` » à une boucle indexée « `for i in range(len(iterable))` » lorsque la connaissance de l'indice i ne sert à rien.
- Comme on le fait en général en mathématiques, réserver les noms `i`, `j`, `k`, `m`, `n` à des entiers et en particulier à des indices, et par conséquent éviter d'écrire « `for i in L` » si `L` ne désigne pas une séquence d'entiers ; ce dernier point est parfois révélateur d'une **confusion** encore trop souvent observée **entre l'objet** (sa valeur s'il s'agit d'un nombre) **et son indice** (sa position dans la séquence).
- La distinction entre une liste (type `list`) et un vecteur (tableau `numpy.ndarray` à un seul indice) doit être parfaitement comprise ; les avantages et les inconvénients de ces deux types complémentaires doivent être connus, ainsi que les fonctions de conversion pour passer de l'un à l'autre (la fonction `numpy.array` et la méthode `tolist`).
- L'utilisation de variables globales n'est pas conseillée, et encore moins exigée.

6.2 – Gestion du temps

Quelques candidats perdent un temps considérable avec des pratiques peu adaptées pour une épreuve de 30 minutes :

- Il est bon de connaître et de savoir utiliser par exemple les fonctions intrinsèques `min`, `max`, `sum`, `sorted`, les méthodes `append`, `extend`, `sort`, `index` pour les listes, les méthodes `min`, `max`, `argmin`, `argmax`, `sum`, `mean`, `std`, `transpose`, `conjugate`, ... pour les tableaux `numpy.ndarray` (`T.real` et `T.imag` aussi pour un tableau de complexes) ; ces méthodes existent aussi sous forme de fonctions dans le module `numpy`.
- Les techniques de *slicing* peuvent être utilisées :
  - ◇ « `U[debut:fin:pas]` » pour une séquence (liste, chaîne de caractères, vecteur, etc.) ;
  - ◇ « `M[Ldeb:Lfin:dL,Cdeb:Cfin:dC]` » pour une matrice (tableau à 2 indices).

  Ce dernier point particulièrement important fait l'objet d'un encadré spécifique dans le mémento fourni aux candidats.

- Il a encore été observé cette année un abus de la méthode **append** pour créer des séquences très simples. Des exemples caricaturaux observés plusieurs fois :

```
L = []
L.append(a)
return L
```
au lieu de
```
return [a]
```

```
L = []
for i in range(10) :
    L.append(i)
```
au lieu de
```
L = list(range(10))
```

```
L = []
for x in np.linspace(-2.5,2.5,51) :
    L.append(x)
V = np.array(L)
```
au lieu de
```
V = np.linspace(-2.5,2.5,51)
```

  Dans une moindre mesure :

```
L = [ i for i in range(10) ]
```
au lieu de
```
L = list(range(10))
```

- Même si les listes en compréhension ne sont pas exigibles, leur utilisation maîtrisée permet de gagner en efficacité et en lisibilité ; de nombreux candidats les ont utilisées en 2019.
- Ne pas hésiter à réutiliser les fonctions créées dans les questions précédentes, ou même à créer de petites fonctions intermédiaires si cela peut être utile ; les exercices sont très souvent structurés dans cet esprit.
- Dans le rapport 2018, il était indiqué : « *L'écriture systématique de commentaires et d'en-têtes (`"docstring"`) pour les fonctions est déconseillée pour l'oral ; même si elle est légitimement préconisée en génie logiciel, elle fait perdre un temps précieux ; les explications peuvent être données oralement par le candidat.* ». Ce point a été particulièrement suivi en 2019, ce qui a contribué à une meilleure efficacité générale des candidats.
- En revanche, un effort doit être fait pour éviter les écritures redondantes contenant des booléens ; par exemple, si une fonction **test** a été définie précédemment et que **test(a,b)** donne un booléen, on écrira :

```
t = test(a,b)
```
et non pas
```
if test(a,b) == True :
    t = True
else :
    t = False
```

```
while not test(a,b) :
    ...
```
et non pas
```
while test(a,b) == False :
    ...
```

6.3 – Algorithmique

- Les algorithmes du cours et leurs coûts de calcul doivent être connus (algorithmes de tri, méthodes par dichotomie, de Newton, d'Euler, des trapèzes, pivot de Gauss, algorithme d'orthonormalisation de Gram-Schmidt, algorithme d'Euclide, etc.). Leur connaissance est fréquemment évaluée. Lors de la session 2019, une amélioration sur ce point a été perçue.
- En revanche, deux algorithmes simples ont semblé poser des difficultés inattendues :
  - ◇ L'extraction de la liste **D** des éléments deux à deux distincts d'un itérable **L**, qui peut se faire par exemple par :

```
D = []
for e in L :
    if e not in D :
        D.append(e)
```
ou même par
```
D = list(set(L))
```
;

◇ Le calcul de la liste des premiers termes d'une suite itérative du type $u_{n+1} = f(u_n)$, qui peut se faire par exemple par :

```
L = [uo]
for i in range(n) :
    L.append( f(L[-1]) )
```

.

- La distinction claire entre *algorithme récursif* et *algorithme itératif* doit être acquise ; dans l'écriture d'une fonction récursive, un soin particulier doit être porté à la condition d'arrêt.

6.4 – À propos d'intelligence et de programmation

Répondre à une question d'exercice d'informatique ne se résume pas systématiquement à traduire directement un énoncé en code informatique de façon automatique (sans reformulation). Ceci est particulièrement vrai lorsqu'un critère d'arrêt ou de poursuite est donné. Il est très souvent préférable de reformuler intelligemment ce critère pour éviter des calculs inutiles, par une résolution au brouillon.

Exemples, où le nombre d'itérations est connu avant de commencer la boucle :

```
for n in range( int( 1/erreur - a ) + 1 )
    ...
```

est préférable à

```
n = 0
while 1/(n+a) >= erreur :
    ...
    n = n+1
```

;

```
import numpy as np
nmax = int(0.5*np.log(c)/np.log(x))
for n in range(1,nmax+1)
    ...
```

est préférable à

```
n = 1
while x**(2*n) <= c :
    ...
    n += 1
```

.

7 – Analyse des résultats

En 2019, 1500 candidats ont passé l'oral de « *Mathématiques et algorithmique* ». Chacun des 11 jours de l'oral, les 8 ou 9 jurys se sont efforcés de poser des exercices balayant l'ensemble du programme, tant en mathématiques qu'en algorithmique et simulation numérique.

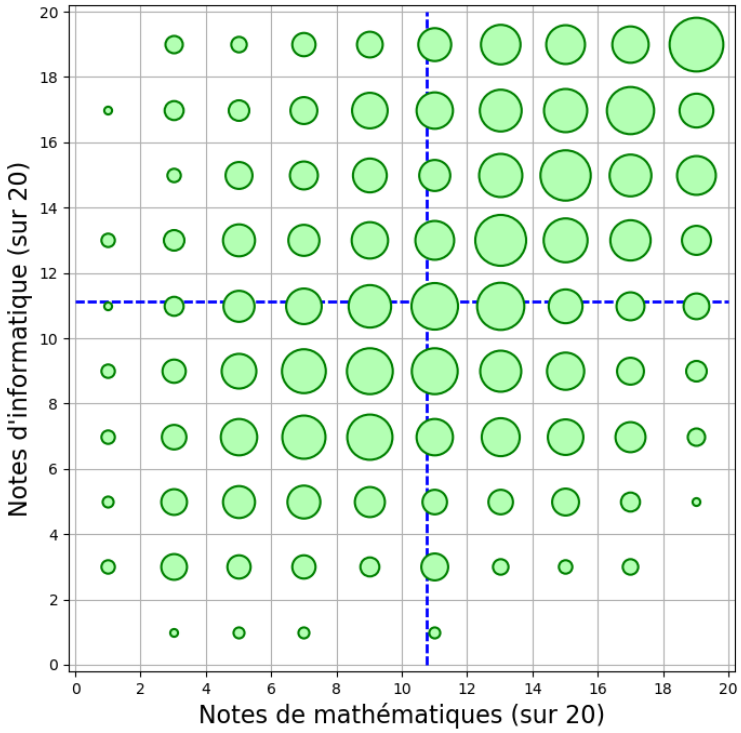
Ainsi, 220 exercices différents d'analyse et de probabilités ont été proposés à 800 candidats contre 191 exercices différents de géométrie et d'algèbre proposés à 700 candidats.

166 exercices différents d'informatique à dominante algorithmique ont été posés à 817 candidats, contre 172 exercices à dominante « *simulation numérique* » pour 683 candidats.

Les statistiques sur les notes sont les suivantes [3] :

Oral 2019	**Note (sur 20)**	Math. (sur 10)	Algo. (sur 10)
Moyenne	**10,94**	5,38	5,56
Écart-type	**3,94**	2,36	2,40
Minimum	1	0	0
Maximum	20	10	10

3. Rappelons que seule la note globale est communiquée au candidat.



Distribution des notes 2019

Éric Ducasse, Coordonnateur de l'épreuve orale de « *Mathématiques et algorithmique* » de la Banque PT, Le 10 juillet 2019.
eric.ducasse@ensam.eu

Annexe 1 – À propos du mémento pour l'oral

La version actuelle du mémento de l'oral date d'août 2018 (voir **site de la Banque PT**). Elle est destinée uniquement au passage de l'oral. Ce mémento est à disposition du candidat sous forme d'une seule feuille plastifiée au format A3 recto-verso.

Un mémento à vocation pédagogique, plus fourni, est disponible sur l'espace numérique de travail *Arts et Métiers* https://savoir.ensam.eu/moodle/course/view.php?id=1428 destiné aux étudiants des Arts et Métiers. Des supports de référence sont également disponibles sur ce site.

Annexe 2 – Exemples d'exercices d'informatique

Ces exercices ayant été posés de nombreuses fois au cours des dernières sessions, ils sont communiqués aux futurs candidats à titre d'exemples. Attention : il ne sont pas forcément représentatifs de tous les exercices se trouvant dans la banque d'exercices.

D'autres exercices publiés sont joints aux rapports 2015, 2016 et 2018.

2019.1 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On considère une pièce de monnaie équilibrée, c'est-à-dire donnant à chaque lancer Pile (1) ou Face (0) avec la même probabilité $p = \frac{1}{2}$ (Loi de Bernoulli).

1. Écrire une fonction **`tirer`** d'argument un entier n et renvoyant la série de résultats (1 ou 0) obtenus en simulant la succession de n lancers de la pièce. On pourra utiliser la fonction **`randint`** du module **`random`**.
2. Soit une liste de zéros et de uns obtenue par n lancers indépendants numérotés de 0 à $(n-1)$.
   On s'intéresse aux « *séries* », c'est-à-dire aux successions de lancers pour lesquels la pièce tombe du même côté. Par exemple la suite de lancers 11101001111 contient successivement les séries : 111, 0, 1, 00, 1111. La longueur d'une série est le nombre de lancers qu'elle contient. Dans notre exemple, les longueurs sont : 3,1,1,2,4.

   Créer une fonction **`longueursSeries`** dont l'argument est une liste L de zéros et de uns et qui renvoie la suite des longueurs des séries obtenues, dans l'ordre. Par exemple, $L =$**`[1,1,1,0,1,0,0,1,1,1,1]`** donnera **`[3,1,1,2,4]`**. Noter que la somme des longueurs des séries vaut n, la longueur de L.
3. En déduire une fonction **`tirerSeries`** d'argument un entier n et renvoyant la liste des longueurs des séries issues d'une simulation de n lancers successifs.
4. On souhaite maintenant compter le nombre de séries au fur et à mesure qu'elles apparaissent lors des lancers. Par exemple :

Numéro i du lancer :	0	1	2	3	4	5	6	7	8	9	10
Valeur du lancer :	1	1	1	0	1	0	0	1	1	1	1
Nombre N_i de séries obtenu :	1	1	1	2	3	4	4	5	5	5	5

   Écrire une fonction **`compter`** d'argument une liste s de longueurs de séries et qui renvoie la liste $[N_1, N_2, \cdots, N_n]$, où N_i est le nombre de séries obtenu après i lancers. Par exemple, **`[3,1,1,2,4]`** donnera **`[1,1,1,2,3,4,4,5,5,5,5]`**.
5. À partir des fonctions construites, mettre en oeuvre une démarche permettant d'émettre une conjecture sur le nombre moyen de séries lors d'une succession de n lancers.

2019.2 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Pour tout entier naturel n non nul, on note E_n la liste $[1, \cdots, n]$ des entiers de 1 à n. On représente une partie A de E_n par la liste $[\alpha_1, \cdots, \alpha_p]$, où $\alpha_1 < \cdots < \alpha_p$. Par convention, la partie vide est représentée par la liste vide [ ].

1. On considère la fonction suivante :

```
1 def A(listes,numero):
2     resultat = []
3     for L in listes :
4         resultat.append( L + [numero] )
5     return resultat
```

   Quel est le type de l'argument **listes** ? Que renvoie la fonction **A** ?

On représente un ensemble de parties de E_n par une liste de listes représentant ces parties. Ainsi l'ensemble $\{\emptyset, \{2\}, \{3, 4\}\}$ est représenté, par exemple, par $[[\,], [2], [3, 4]]$.

2. Donner la liste **L0** représentant l'ensemble des parties de E_5 à 0 élément.
3. Écrire une fonction récursive **parties** de deux arguments n et p renvoyant la liste représentant l'ensemble des parties de E_n ayant au plus p éléments.

   On pourra remarquer que l'ensemble des parties à au plus p éléments de E_n se partitionne en l'ensemble des parties à au plus p éléments de E_n contenant n et l'ensemble des parties à au plus p éléments de E_n ne contenant pas n.

4. Donner la liste **L1** représentant l'ensemble des parties de E_5 à 1 élément, et celle **L5** représentant l'ensemble des parties de E_5 à 5 éléments.
5. Écrire une fonction récursive **parties2** de deux arguments n et p renvoyant la liste représentant l'ensemble des parties de E_n à p éléments.

2019.3 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Pour un entier naturel non nul n, son nombre de diviseurs est le nombre d'entiers naturels inférieurs ou égaux à n qui le divisent. Si cet entier n a strictement plus de diviseurs que chacun des entiers naturels qui le précèdent, on dit que c'est un nombre *riche*. C'est un peu le contraire d'un nombre premier qui n'a que 2 diviseurs.

L'objet de cet exercice est de créer la liste des premiers nombres *riches* puis d'examiner leurs facteurs premiers.

1. Écrire une fonction **`nbdiv`** d'argument un entier naturel n qui renvoie le nombre de diviseurs de n. La tester pour $n = 60$.
2. Écrire une fonction **`riches`** d'argument un entier naturel K qui renvoie les K premiers nombres riches sous forme d'une liste de couples (n, d), où d est le nombre de diviseurs de n. Par exemple, **`riches(3)`** doit donner `[[1,1],[2,2],[4,3]]`. Faire afficher les 16 premiers nombres riches.
3. Écrire une fonction **`facteursPremiers`** d'argument un entier naturel n qui renvoie la liste des facteurs premiers de n, avec répétitions si nécessaire. Par exemple, **`facteursPremiers(360)`** doit renvoyer `[2,2,2,3,3,5]`.
4. Modifier la fonction **`facteursPremiers`** de sorte qu'elle renvoie la décomposition en facteurs premiers de n sous la forme d'une liste de couples (p, i), où i est le nombre de fois où apparaît le facteur premier p dans la décomposition. Par exemple, **`facteursPremiers(360)`** doit maintenant renvoyer `[[2,3],[3,2],[5,1]]`.
5. En utilisant la fonction **`facteursPremiers`** sur les premiers nombres riches, que peut-on conjecturer sur le lien entre la décomposition en facteurs premiers et le nombre de diviseurs ?

2019.4 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Pour tout ensemble non vide, identifié ici à une liste d'éléments deux à deux distincts, $E = [e_0, \ldots, e_{n-1}]$ de taille n, chacune de ses parties A peut être codée sous forme d'une liste `C` à n éléments contenant des zéros et des uns :

`C[i] = 1` si $e_i \in A$, et `C[i] = 0` sinon.

Par exemple, les parties $\emptyset$, $[a]$, $[b]$ et $[a, b]$ de la liste $[a, b]$ sont respectivement codées par les listes $[0, 0]$, $[1, 0]$, $[0, 1]$ et $[1, 1]$.

1. Écrire une fonction **`decoder`** de deux arguments E et C renvoyant la partie de la liste E codée par le code C.

   Ainsi, `decoder([2,3,5,7],[1,0,0,1])` donne `[2,7]` ; de même `decoder([2,3,5,7],[0,0,0,0])` donne `[]`.

2. Écrire une fonction **`coder`** de deux arguments E et A renvoyant le code de la partie A de la liste E.

   Ainsi `coder([2,3,5,7],[2,7])` donne `[1,0,0,1]`.

3. Écrire une fonction **`incrementer`** d'argument une liste C de zéros et de uns de taille n, représentant sur n bits l'écriture en base 2 d'un entier naturel k, et renvoyant la liste représentant sur n bits l'écriture en base 2 de l'entier $(k + 1)$.

   Par exemple, `[0,1,0,1,1,1]` est l'écriture en base 2 sur 6 bits de 23 et `[0,1,1,0,0,0]`, l'écriture en base 2 sur 6 bits de 24.

4. En déduire la fonction **`parties`** d'argument E renvoyant la liste des parties de la liste E.

5. Écrire une fonction **`p_parties`** de deux arguments, un ensemble E non vide de taille n et un entier naturel $p \leqslant n$, qui renvoie la liste des parties de E de taille p.

2019.5 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

L'objet de cet exercice est d'étudier $q(n)$ le nombre de triplets (x, y, z) d'entiers naturels tels que :

$$x + 2\,y + 3\,z = n\,.$$

1. Programmer une fonction p d'argument m renvoyant le nombre de couples (x, y) d'entiers naturels tels que $x + 2\,y = m$. On distinguera les cas « m pair » et « m impair ».
2. En déduire une fonction `q1` d'argument n renvoyant $q(n)$. Calculer $q(n)$ pour n entier de 0 à 10, ainsi que $q(100)$.
3. Au brouillon, exprimer $q(n+3)$ en fonction de $q(n)$ et de $p(n+3)$.

   En déduire que $q(n+6) = q(n) + n + 6$.
4. En déduire une fonction récursive `q2` d'argument n renvoyant $q(n)$.
5. On peut démontrer que $q(j + 6\,k) = q(j) + \sum_{i=1}^{k}(j + 6\,i) = q(j) + j\,k + 3\,k\,(k+1)$.

   En déduire une troisième fonction `q3` d'argument n renvoyant $q(n)$.
6. Quels sont les coûts de calcul des fonctions `q1`, `q2`, `q3` ? Conclure.

2019.6 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit une liste L de longueur paire $2\,n$. L est dite de type $\mathcal{D}$ si et seulement si :
- L ne contient que des 1 et des -1 ;
- L contient autant de 1 que de -1 ;
- chaque sous-liste d'éléments consécutifs de L en partant du début contient au moins autant de 1 que de -1.

Par convention, la liste vide est de type $\mathcal{D}$.

1. Quel est nécessairement le premier élément d'une liste de type $\mathcal{D}$? Au brouillon, donner toutes les listes de type $\mathcal{D}$ de longueurs 2 et 4.
2. Écrire une fonction `D2par` d'argument une liste L qui ne contient que des 1 et des -1, qui renvoie une chaîne de caractères obtenue en convertissant chaque 1 de L en parenthèse ouvrante `"("` et chaque -1 en parenthèse fermante `")"`, et en mettant bout-à-bout les caractères ainsi obtenus.
   Tester cette fonction sur les listes de type $\mathcal{D}$ de longueurs 2 et 4.
3. Écrire une fonction `DQ` d'argument une liste L qui renvoie un booléen indiquant si L est de type $\mathcal{D}$ ou pas.
4. On admet que toute liste de type $\mathcal{D}$ non vide s'écrit de façon unique sous la forme `[1]+u+[-1]+v`, où `u` et `v` sont deux listes de type $\mathcal{D}$, éventuellement vides.
   Écrire une procédure récursive `LD` d'argument un entier naturel N qui renvoie la liste de toutes les listes de type $\mathcal{D}$ de longueur $2N$.

2019.7 – **Exercice à dominante algorithmique**

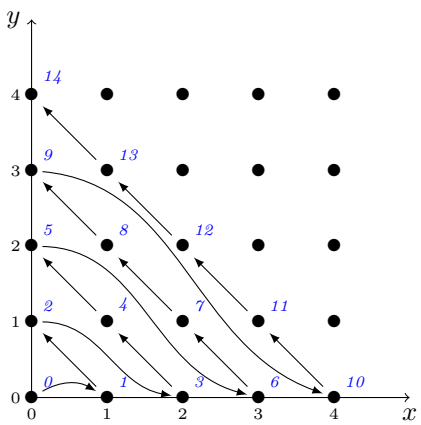
Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit une matrice carrée M d'ordre 9 dont chaque coefficient est repéré par un numéro de ligne i et un numéro de colonne j, compris entre 0 et 8. On attribue à chacun de ces coefficients un seul numéro $n(i,j)$ défini par le code *Python* suivant :

```
def n(i,j):
    if j == 0 :
        if i == 0 :
            return 0
        else :
            return n(i-1,8)+1
    else :
        return n(i,j-1)+1
```

1. Expliquer concrètement comment la fonction n numérote les coefficients de M.
2. Déterminer l'expression de $n(i,j)$ en fonction de i et de j.
3. Écrire une fonction **`ij`** d'argument m qui renvoie le couple (i,j) tel que $n(i,j)=m$.

On se place maintenant dans un plan muni d'un repère orthonormé, et on numérote chaque point de coordonnées (x,y) où x et y sont des entiers naturels par le procédé décrit sur la figure ci-dessous :



4. Écrire une fonction récursive **`num1`** d'arguments x et y ayant pour résultat le numéro du point de coordonnées (x,y).
5. Écrire une fonction non récursive **`num2`** faisant la même chose.
6. Déterminer la fonction réciproque de **`num1`** (ou de **`num2`**).

2019.8 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On considère un jeu de 32 cartes. Il est formé de couples de 8 valeurs ordonnées, `valeurs=["7", "8", "9", "10", "V", "D", "R", "A"]`, et de 4 « *couleurs* », `couleurs=["T", "K", "C", "P"]` (Trèfle, Carreau, Coeur, Pique). On distribue au hasard une « *main* », c'est-à-dire 5 cartes distinctes, et on s'intéresse à des mains particulières :

- les « *une paire* » (2 cartes de même valeur et 3 cartes de valeurs différentes) ;
- les « *deux paires* » (2 cartes d'une même valeur, 2 cartes d'une même autre valeur, et une carte d'une troisième valeur) ;
- les « *fulls* » (3 cartes d'une même valeur et 2 cartes d'une même autre valeur) ;
- les « *carrés* » (les 4 cartes d'une même valeur, et une autre).

1. Construire la liste **cartes** des 32 cartes à partir des deux listes **valeurs** et **couleurs**, chaque carte étant représentée par la paire $[valeur, couleur]$. Vérifier que le nombre de cartes obtenu est correct.
2. Écrire une fonction **tirerMain** sans argument qui renvoie une liste de 5 cartes distinctes tirées au hasard. On pourra utiliser la fonction **sample** du module **random**.
3. Écrire une fonction **LV** d'argument une main et qui renvoie la liste croissante des nombres de valeurs identiques dans la main. Par exemple, si la main est « *une paire* », la fonction **LV** renvoie [1,1,1,2]. On pourra utiliser la fonction intrinsèque `sorted`.
4. À partir de 50000 tirages aléatoires de mains, estimer les probabilités d'obtenir une « *une paire* », « *deux paires* », un full et un carré.
5. Comparer ces estimations avec les probabilités obtenues par dénombrement. On pourra utiliser la fonction **comb** du module **scipy.special**, ou faire autrement.

2019.9 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Écrire une fonction **`sp`** d'argument une liste L qui renvoie la liste obtenue en prenant d'abord le dernier terme de L, puis le premier, puis l'avant-dernier, puis le deuxième, puis l'antépénultième, puis le troisième, *etc.* (On parle de permutation *en spirale*). Vérifier que **`sp([1,2,3,4,5,6])`** donne bien **`[6,1,5,2,4,3]`**, et que **`sp([1,2,3,4,5,6,7])`** donne **`[7,1,6,2,5,3,4]`**.
2. Vérifier que si l'on itère la fonction **`sp`** sur $L = [1, 2, 3, 4, 5, 6]$ on retombe sur L à la sixième itération. Qu'en est-il pour $L = [1, 2, 3, 4, 5, 6, 7]$?
3. Écrire une fonction **`periode`** d'argument un entier n qui renvoie le nombre minimum p d'itérations de la fonction **`sp`** pour retomber sur la liste $[1, 2, \cdots, n]$, en partant de cette même liste.
4. Déterminer les entiers inférieurs ou égaux à 99 tels que `periode`$(n) = n$.
5. À l'aide des instructions **`plot`** et **`show`** du module **`matplotlib.pyplot`**, faire tracer `periode`$(n)/n$ en fonction de n, pour n compris entre 1 et 99. Déterminer la valeur de n qui minimise `periode`$(n)/n$.

2019.10 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

Soit p un nombre premier. On note E_p l'ensemble des entiers naturels compris entre 1 et $(p-1)$. Pour deux éléments x et y de E_p, on appelle alors « *produit de x et y dans E_p* », noté $[\,x\,y\,]_p$, le reste de la division euclidienne de $x\,y$ par p, ce reste étant nécessairement non nul. Le « *carré de x dans E_p* », noté $[\,x^2\,]_p$, sera ainsi défini comme le produit de x par x dans E_p. Plus généralement, pour un entier naturel k non nul, $[\,x^k\,]_p$ sera défini par récurrence : $[\,x^1]_p = x$ et, pour tout k dans $\mathbb{N}^\star$, $[\,x^{k+1}\,]_p$ est le produit de x par $[\,x^k\,]_p$ dans E_p.

1. Lorsque p=17, construire de proche en proche la liste des p premiers $[\,2^k\,]_p$ et l'afficher.
2. Pour chaque $x \in E_p$, l'ensemble des $[\,x^k\,]_p$ pour $k \in \mathbb{N}$ est fini car il est inclus dans E_p.

   Écrire une fonction **`orb`** de deux arguments p et x qui renvoie la liste L des $[\,x^k\,]_p$ pour k entier de 1 à n, où n est l'unique valeur telle que la liste L ne contient que des valeurs distinctes et $[\,x^{n+1}]_p \in L$.

   Faire afficher **`orb(17,`**x**`)`** pour tous les éléments x de E_{17}. Que constatez-vous ?
3. On admettra que quel que soit le nombre premier p, il existe un élément g de E_p tel que l'ensemble des $[\,g^k\,]_p$ pour $1 \leqslant k \leqslant p$ soit exactement l'ensemble E_p. On dit que g *est un générateur de E_p*.

   En utilisant la propriété admise que g est un générateur de E_p si et seulement si $p-1$ est le plus petit entier non nul k tel quel $[\,g^k\,]_p = 1$, écrire une fonction **`ppg`** d'argument un nombre premier p qui renvoie le *plus petit générateur* de E_p.

   Déterminer le plus petit générateur de E_p pour p=17, puis pour p=106031.
4. Si g est un générateur de l'ensemble E_p, indiquer pourquoi l'application $x \mapsto [\,g^x\,]_p$ définit une bijection de E_p dans lui-même.

   Écrire ensuite une fonction **`recip`** de trois arguments, un nombre premier p, un générateur g de E_p et y un élément de E_p, qui renvoie l'élément x de E_p tel que $y = [\,g^x\,]_p$. La recherche de y se fera par essais successifs des éléments de E_p.

   Dans le cas p=106031, déterminer l'antécédent de y=2.
5. Quel est le coût de la fonction **`recip`** dans le meilleur des cas, dans le pire des cas ?

2019.11 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Le fichier **`algo118-pi-digits.txt`** se trouvant dans le répertoire **`data`** contient les premières décimales de π, sur une seule ligne et sans espace entre les chiffres.

   Récupérer le contenu de ce fichier sous forme d'une chaîne de caractères que l'on nommera **`decpi`**.

2. Faire afficher les 10 premiers caractères de **`decpi`**, ses 10 derniers, ainsi que le nombre **`nbdec`** de caractères de **`decpi`**.

3. Écrire une fonction **`trouve`** de deux arguments, deux chaînes de caractères P et M, qui renvoie un entier naturel p si M est une sous-chaîne de P commençant à la position p, et -1 si M n'est pas une sous-chaîne de P. On n'utilisera pas la méthode **`find`** de la classe **`str`**.

   Par exemple, **`trouve("BanquePT","an")`** donne 1, **`trouve("BanquePT","PT")`** donne 6, alors que **`trouve("BanquePT","PSI")`** donne -1. Comparer avec **`"BanquePT".find("an")`**, *etc.*

4. Les nombres **`"314159"`**, **`"123456"`**, **`"12345"`**, et **`"1789"`** se trouvent-ils dans **`decpi`**, et si oui, en combien d'exemplaires et à quelle(s) position(s) ?

5. Le fichier **`algo118-pi-false.txt`** se trouvant dans le répertoire **`data`** contient les premières décimales de π, avec quelques chiffres erronés. Combien et en quelles positions ?

2019.12 – **Exercice à dominante algorithmique**

Cet exercice devra être fait avec le langage Python. À chaque question, les instructions ou les fonctions écrites devront être testées.

On appelle *entier palindrome* un entier naturel non nul qui est égal à l'entier obtenu en renversant l'ordre de ses chiffres. Par exemple, 22, 121 et 2552 sont des entiers palindromes et 13, 211 et 2525 n'en sont pas. Attention, il ne peut y avoir de 0 comme premier ou dernier chiffre d'un entier palindrome.

1. Après avoir observé dans la console ce que donne `list(str(123))`, écrire une fonction **`isp`** d'argument un entier naturel non nul qui renvoie un booléen indiquant si l'entier est un palindrome, ou pas.

   Par exemple, `isp(1245421)` donne `True` alors que `isp(1245422)` donne `False`.

2. Afficher la liste de tous les palindromes compris entre 1 et 1000, obtenue grâce à la fonction **`isp`**, ainsi que le nombre de ses éléments.

3. Écrire une fonction **`palp`** d'argument un entier naturel non nul p qui renvoie la liste de tous les palindromes à $2\,p$ chiffres, construite à partir des nombres de p chiffres.

   Vérifier que 1001, 2002, 3003 sont éléments de la liste `palp(2)`, qui doit comporter 90 éléments.

4. En déduire une fonction **`palindromes`** d'argument un entier naturel non nul n qui renvoie la liste de tous les palindromes à n chiffres.

5. Afficher tous les entiers palindromes d'au plus 8 chiffres, de carrés palindromes.

   Que peut-on conjecturer ?

6. Écrire une fonction **`postulants`** d'arguments un entier naturel non nul p, qui renvoie la liste des palindromes d'au plus $2\,p$ chiffres qui ne contiennent que des 0 et des 1, avec éventuellement un 2 en position centrale ou aux extrémités.

   Afficher le nombre d'éléments de la liste `postulants(6)`, ainsi que le nombre d'éléments de celle-ci dont les carrés ne sont pas des palindromes.

2019.13 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On note $\lfloor u \rfloor$ la partie entière de u. Soit x un réel. On lui associe les suites $(a_n)_{n\geqslant 0}$ et $(v_n)_{n\geqslant 0}$ définies par

$$v_0 = x, \quad a_n = \lfloor v_n \rfloor, \quad \text{et} \quad v_{n+1} = \begin{cases} \dfrac{1}{v_n - a_n} & \text{si } v_n \neq a_n \\ 0 & \text{sinon} \end{cases} \quad \text{pour } n \geqslant 0.$$

1. Calculer les 10 premiers termes de la suites $(a_n)_{n\geqslant 0}$ pour $x = \sqrt{3}$.
2. Écrire une fonction `A` qui, à un réel x et un entier N, associe les N premiers termes de la suite. On testera les valeurs $x = 2/3, \frac{\sqrt{5}-1}{2}, \mathrm{e}, \mathrm{e}^2, \pi$.
3. Écrire une fonction `ND` de trois arguments entiers a, n et d qui renvoie le numérateur et le dénominateur de $a + \dfrac{d}{n}$ (sans simplification éventuelle).
4. En déduire une fonction `frac` qui, à un réel x et un entier N, associe le numérateur et le dénominateur de la fraction

$$c_N = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \; a_{r-2} + \cfrac{1}{a_{r-1} + \cfrac{1}{a_r}}}}}$$

   où les a_i dépendent de x et où r est le plus grand entier inférieur ou égal à N tel que $a_r \neq 0$.
5. On admet que la suite $(c_n)_{n\geqslant 0}$ tend vers x (elle peut être constante à partir d'un certain rang).

   Écrire un programme qui à un réel x et un réel positif ε associe un couple (p, q) tel que

$$\left| x - \frac{p}{q} \right| < \varepsilon.$$

2019.14 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Un nombre palindrome est un entier positif symétrique écrit dans une certaine base a comme ceci :

$$a_1\, a_2\, a_3 \cdots \cdots a_3\, a_2\, a_1 \ .$$

Par exemple, il existe 90 nombres palindromes de trois chiffres :

```
101, 111, 121, 131, 141, 151, 161, 171, 181, 191,
                          ...
909, 919, 929, 939, 949, 959, 969, 979, 989, 999.
```

Dans tout l'exercice, on manipulera des nombres écrits en base 10.

1. Écrire une fonction **`Retourner`** d'argument un nombre entier et renvoie son « *retourné* ». On pourra utiliser en *Python* les instructions **`str`** et **`int`**.
2. Écrire une fonction **`EstPalindrome`** d'argument un nombre entier n et qui renvoie un booléen indiquant si n est un palindrome ou pas. La tester sur différentes valeurs.
3. Écrire une fonction **`Palindromes`** d'argument un nombre entier N qui renvoie la liste de tous les entiers palindromes inférieurs à N.

Nous avons le procédé suivant pour construire des nombres palindromes :

— Prendre un nombre au hasard (par exemple 39) ;

— lui ajouter le nombre « *retourné* » $(39 + 93 \to 132)$;

— à ce résultat, ajouter son nombre « *retourné* » $(132 + 231 \to 363)$;

— on répète le procédé jusqu'à obtention d'un nombre palindrome.

Attention : dans de rares cas ce calcul peut ne pas aboutir !

4. Écrire une fonction **`ConstruitPalindrome`** d'argument un nombre entier n et qui renvoie le nombre palindrome construit à partir de l'algorithme précédent.
5. Modifier cette fonction pour qu'elle renvoie aussi le nombre d'étapes nécessaires pour obtenir un palindrome.
6. Modifier cette fonction pour qu'elle s'arrête automatiquement au bout d'un nombre K de pas si un palindrome n'a pas été obtenu. Essayer de construire un nombre palindrome à partir de 196.

2019.15 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On considère la suite (u_n) à valeurs réelles vérifiant :

$$u_0 = x\,, \quad u_1 = y\,, \quad u_2 = z\,, \quad \text{et}\,, \quad \forall n \geqslant 2\,,\ u_{n+3} = 2\,u_{n+2} + u_{n+1} - u_n\ .$$

On note λ, μ et ν les trois racines du polynôme $P = X^3 - 2X^2 - X + 1$ vérifiant $|\lambda| > |\mu| > |\nu|$.
On admettra dans le cadre de cet exercice qu'il existe pour toutes valeurs initiales x, y et z, trois coefficients réels a, b et c tels que :

$$\forall n \in \mathbb{N} \quad u_n = a\,\lambda^n + b\,\mu^n + c\,\nu^n$$

1. Écrire une fonction `LU` de quatre arguments N, x, y et z, qui renvoie la liste des N premières valeurs de la suite u_n.

   Faire afficher les 10 premières valeurs de la suite pour $x = 1$, $y = 2$ et $z = 3$.
2. À l'aide du quotient $\dfrac{u_{n+1}}{u_n}$, terme général d'une suite dont on déterminera la limite au brouillon, trouver une valeur approchée de λ qui rend la valeur du polynôme P inférieure à 10^{-10} en valeur absolue. On admettra que pour $x = 1$, $y = 2$ et $z = 3$, le coefficient a est non nul.
3. Trouver de même une valeur approchée de ν à l'aide du polynôme $Q = X^3\,P\,(1/X)$ et de la suite récurrente v_n associée.
4. En déduire une valeur approchée de μ.

2019.16 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On définit la suite $(t_n)_{n\in\mathbb{N}}$ à valeurs dans $\{0,1\}$ par :

$$t_0 = 0 \quad \text{et,} \quad \forall n \in \mathbb{N}, \quad \begin{cases} t_{2n} & = & t_n \\ t_{2n+1} & = & 1 - t_n \end{cases} .$$

1. Écrire une fonction récursive **`t`** d'argument un entier naturel n et renvoyant t_n.

On appelle « *mot* T » le mot binaire infini obtenu en concaténant les t_i :

`"0110100110010110 ⋯ "`

2. Écrire une fonction **`mot`** d'argument un entier naturel n non nul et renvoyant, sous forme d'une chaîne de caractères, le mot binaire constitué des n premiers caractères du *mot* T.

   Par exemple, l'appel de **`mot(5)`** doit renvoyer la chaîne `"01101"`.

   Faire afficher **`mot(20)`**.
3. Écrire une fonction **`nbseq`** de deux arguments, un entier naturel n non nul et une chaîne de caractères **`seq`**, calculant le nombre d'occurrences du mot binaire **`seq`** dans le mot **`mot(n)`**.

   Par exemple, l'appel **`nbseq(15,"11")`** doit renvoyer 3 car la séquence `"11"` apparaît 3 fois dans **`mot(15)=`**`"011010011001011"`. Attention, on comptera deux fois `"11"` dans `"111"`.
4. Conjecturer la limite de **`nbseq(n,seq)/n`** pour n tendant vers l'infini, pour tous les mots binaires **`seq`** de un, deux puis trois chiffres.
5. Quelle est la complexité de la fonction **`mot`** ? Quelle(s) amélioration(s) pourrait-on essayer d'apporter ?

2019.17 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On considère $n+1$ cases numérotées de 0 à n ; on place une bille dans la première case puis on répète n fois l'action suivante : faire avancer la bille d'une case ou la laisser dans sa case, avec équiprobabilité. Le résultat final de l'expérience est le numéro de la case dans laquelle se trouve la bille à la fin.

1. Créer une fonction **`tirer`** d'argument n qui effectue la simulation de l'expérience précédente et renvoie le numéro de la case dans laquelle se trouve la bille à la fin. 0n pourra utiliser la fonction **`randint`** du module **`random`**.
2. On effectue N fois le processus précédent. Pour $n = 10$ et $N = 500$, tracer les points représentant, en fonction du numéro k de la case, la proportion de billes se trouvant à la fin dans la case k.
3. Programmer une fonction renvoyant le coefficient binomial $\binom{n}{k}$.
4. Rajouter sur la figure précédente les points représentant la loi de probabilité associée à l'expérience.
5. Refaire ce qui précède en supposant que, maintenant, la bille a une probabilité p d'avancer d'une case (et $1-p$ de rester sur place). 0n pourra utiliser la fonction **`random`** du module **`random`**, qui tire une valeur entre 0 et 1 avec une distribution uniforme.

2019.18 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Une fourmi ivre suit les lignes d'un quadrillage dont les intersections sont représentées par les couples de nombres entiers. Lorsqu'elle arrive à une intersection, elle a autant de chances de continuer tout droit, que de tourner à droite, de tourner à gauche, ou de revenir sur ses pas.

On note C_a le carré formé des couples $[x, y]$ avec $-a \leqslant x \leqslant a$ et $-a \leqslant y \leqslant a$.

1. Écrire une fonction **`avancer`** sans argument qui renvoie l'un des 4 couples **`[1,0]`**, **`[0,-1]`**, **`[0,1]`** ou **`[-1,0]`** avec équiprobabilité.

   On pourra utiliser la fonction **`randint`** du module **`random`**.
2. Simuler une trajectoire de la fourmi en partant de $[0, 0]$ puis en faisant afficher les nœuds du quadrillage par où elle passe, jusqu'à ce qu'elle sorte du carré C_2.
3. Écrire une fonction **`traj`** d'argument a qui renvoie une trajectoire tirée au hasard, partant du centre $[0, 0]$ et s'arrêtant lorsque la fourmi tente de sortir du carré C_a.
4. Tracer sur un même graphique 6 trajectoires différentes pour $a = 10$.
5. Soit L_a la longueur du trajet effectué par une fourmi pour sortir du carré C_a en partant de son centre. Définir une fonction **`LM`** de deux arguments a et N qui renvoie la moyenne des longueurs de trajet de N fourmis (N réalisations de la variable aléatoire L_a).
6. Tracer pour a entier compris entre 1 et 20, une estimation de $\mathbb{E}(L_a)$, l'espérance de L_a.

2019.19 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

---

1. Écrire une fonction `P0` d'argument un entier naturel N non nul renvoyant un vecteur de dimension $(N+1)$ comportant un 1 en première composante et des zéros ailleurs.
2. Écrire une fonction T d'argument un entier naturel N non nul renvoyant la matrice carrée d'ordre $(N+1)$ de terme général $(t_{ij})_{0\leqslant i,j\leqslant N}$ tel que :

$$t_{ij} = \begin{cases} 1-j/N & \text{si} \quad j=i-1 \\ j/N & \text{si} \quad j=i+1 \\ 0 & \text{sinon} \end{cases} .$$

   On pourra utiliser la fonction **`zeros`** du module **`numpy`** de *Python*.

   Par exemple, **`T(4)`** doit donner la matrice $\begin{pmatrix} 0 & 0.25 & 0 & 0 & 0 \\ 1 & 0 & 0.5 & 0 & 0 \\ 0 & 0.75 & 0 & 0.75 & 0 \\ 0 & 0 & 0.5 & 0 & 1 \\ 0 & 0 & 0 & 0.25 & 0 \end{pmatrix}$.
3. Vérifier que la somme des coefficients de chaque colonne de la matrice `T(10)` vaut 1.
4. Pour une valeur de N donnée, on considère la suite récurrente de vecteurs $(\mathbf{P}_n)_{n\in\mathbb{N}}$ définie par :

$$\mathbf{P}_0 = \texttt{P0(N)} \quad \text{et} \quad \mathbf{P}_{n+1} = \texttt{T(N)}\,\mathbf{P}_n.$$

   Pour $N = 100$, faire tracer sur un même graphique les points représentant $\mathbf{P}_n = [p_{0\,n}, p_{1\,n}, \cdots, p_{N\,n}]$ (avec i en abscisses et $p_{i\,n}$ en ordonnées), pour $n = 20$, 50, 100 et 200.
5. Les i et les $p_{i\,n}$ décrivent en fait la loi de probabilité d'une variable aléatoire finie X_n :

$$\forall i \in \mathbb{N}\,,\ 0 \leqslant i \leqslant N\,,\ \mathbb{P}(X_n = i) = p_{i\,n} \quad .$$

   Définir une fonction **`EX`** de deux arguments N et n qui renvoie $\mathbb{E}(X_n)$, l'espérance de X_n.
6. Pour $N = 100$, faire tracer $\mathbb{E}(X_n)$ en fonction de n, pour $0 \leqslant n \leqslant 300$.
   Rajouter ensuite la courbe de $n \mapsto (N/2)\,(1-\exp(-2\,n/N))$.

2019.20 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

Pour tout entier naturel n, on considère la fonction f_n définie sur $\mathbb{R}$ par : $f_n(x) = x^{n+1} - x^n - 1$.

1. Au brouillon, étudier les variations de f_n. Montrer notamment que f_n est strictement croissante sur $[1, +\infty[$ et que $f_n(2) > 0$ si $n \geqslant 1$.
2. On note α_n l'unique solution de $f_n(x) = 0$ sur $[1, 2]$. Que vaut α_0 ?
3. Tracer les courbes de f_n, pour n entier de 1 à 10. La plage d'affichage sera le rectangle $[1, 2]\times[-1, 1]$ (fonctions **`xlim`** et **`ylim`** du module **`matplotlib.pyplot`** en *Python*).

   Conjecturer le comportement de la suite $(\alpha_n)_{n\geq 0}$.

On rappelle ci-dessous l'algorithme de dichotomie.

Soit f continue sur un segment $[a, b]$ à valeurs réelles. On suppose que f s'annule exactement une fois sur $[a, b]$ en un point que l'on note α. On définit les suites $(a_k)_{k\geq 0}$ et $(b_k)_{k\geq 0}$ de la façon suivante :

— $a_0 = a$ et $b_0 = b$.

— On pose : $\forall k \in \mathbb{N}, c_k = \dfrac{a_k + b_k}{2}$ et

$$\begin{array}{lllll} \text{si} & f(a_k)f(c_k) \leq 0, & \text{alors} & a_{k+1} = a_k & \text{et} \quad b_{k+1} = c_k \\ & & \text{sinon} & a_{k+1} = c_k & \text{et} \quad b_{k+1} = b_k \end{array}.$$

On sait qu'alors les deux suites $(a_k)_{k\geq 0}$ et $(b_k)_{k\geq 0}$ convergent toutes les deux vers α, en vérifiant :

$$\forall k \in \mathbb{N}, a_k \leq \alpha \leq b_k \text{ et } \forall k \in \mathbb{N}, b_k - a_k = \frac{b-a}{2^k}.$$

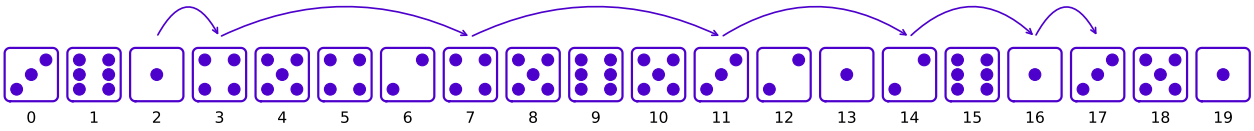
On peut alors montrer que si $\dfrac{b-a}{2^k} \leq \varepsilon$, alors a_k et b_k sont des valeurs approchées de α à ε près.

4. En utilisant l'algorithme précédent, déterminer des valeurs approchées de α_n à 10^{-6} près pour n variant de 2 à 200. Quelle conjecture pouvez-vous faire ?
5. Pour ces mêmes valeurs de n, comparer graphiquement α_n avec la formule empirique $(1 + 1.6\,(n + 2.6)^{-0.83})$.

2019.21 – **Exercice à dominante simulation numérique**

Cet exercice est prévu pour le langage Python (et ses bibliothèques numpy, scipy, matplotlib). À chaque question, les instructions ou les fonctions écrites devront être testées.

On lance n dés cubiques usuels aux faces numérotées de 1 à 6. Les dés sont placés sur une ligne côte à côte. On choisit un dé dans le début de la liste et on avance, de la gauche vers la droite, d'autant de dés que la valeur marquée sur le dé départ. On recommence jusqu'à ce que cela ne soit plus possible. La situation est illustrée par le dessin ci-dessous.



1. Écrire une fonction **`lancer`** sans argument qui simule un lancer de dé et renvoie donc un entier entre 1 et 6. On pourra utiliser la fonction **`randint`** du module **`numpy.random`** de *Python*.
2. Écrire une fonction **`liste`** d'argument un entier n qui renvoie la liste des résultats de n lancers de dés.
3. Écrire une fonction **`arrivee`** de deux arguments, un entier k et une liste L de résultats, qui renvoie l'indice dans la liste L du dernier dé atteint en partant du dé numéro k, avec le procédé décrit dans l'énoncé.

   On testera la procédure en calculant les numéros d'arrivée pour toutes les positions de départ d'un tirage de 15, puis 20, puis 25 dés. Que constatez-vous ?
4. Écrire une fonction **`commun`**, d'argument une liste de dés L, qui renvoie le plus grand entier k tel que les dés en position $0, 1, \ldots, k$ aboutissent à la même position d'arrivée.
5. En utilisant la fonction **`commun`**, estimer la probabilité que les k premières positions d'une liste de n dés tirée au hasard aboutissent à la même position d'arrivée.
6. En déduire une estimation du nombre minimal n tel que les six premiers dés aient 95% de chances d'aboutir à la même position d'arrivée.