Informatique MP Arago

Document de compléments de cours et d'exercices

Ph. Langlois

version partielle du 23 janvier 2015

Table des matières

Bibliographe utilisée et conseillée

[1] T. Audibert and A. Oussalah. *Informatique. Programmation et calcul scientifique en Python et Scilab*. Ellipses, 2014.

[2] B. Cordeau and L. Pointal. Une introduction à Python. `http://perso.limsi.fr/pointal/_media/python:cours:courspython3.pdf`, 2014. version 1.618.

[3] T. Cormen. *Algorithmes. Notions de base*. Dunod, 2013.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique*. Dunod, seconde edition, 2002.

[5] P. Damphousse. *Petite introduction à l'algorithmique. À la découverte des mathématiques du pas à pas*. Ellipses, 2005.

[6] G. Dowek, J.-P. Archambault, E. Baccelli, C. Cimelli, A. Cohen, C. Eisenbeis, T. Viéville, and B. Wack. *Informatique et Sciences du Numérique - Spécialité ISN en Terminale S*. Eyrolles, 2012.

[7] E. Le Nagard. *Informatique. Initiation à l'algorithmique en Scilab et Python*. Pearson, 2013.

[8] B. Wack, S. Conchon, J. Courant, M. de Falco, G. Dowek, J.-C. Filliâtre, and S. Gonnord. *Informatique pour tous en classes préparatoires aux grandes écoles*. Eyrolles, 2014.

Première partie

Rappels de Python

Chapitre 1

Sur les listes python

1.1 Définir des listes en compréhension

Principe : `[fonction(x) for x in uneListe if condition-de-fitrage]`

```
# les carrés des éléments
liste = [1, 2, 3, 4, 5, 6, 7]
les_carres = [x ** 2 for x in liste]

# Afficher les nombres pairs
print [x for x in liste if x % 2 == 0]
# Plus simple que filter, également :)

# Affiche les carrés pairs (combinaison des deux)
print [x ** 2 for x in liste if x ** 2 % 2 == 0]
# ou print [x for x in [a ** 2 for a in liste] if x % 2 == 0]
```

1.2 Principales méthodes sur les listes

`L.append(x)` : Ajoute l'élément x à la fin de la liste L

`L1.extend(L2)` : Étend la liste L1 en y ajoutant tous les éléments de la liste L2

`L.insert(i, x)` : Insère l'élément x dans la liste L à la position i

`L.remove(x)` : Supprime de la liste L le premier élément dont la valeur est x.

`L.pop([i])` : Enlève de la liste L l'élément situé à la position i
Si aucune position n'est indiqué, `L.pop()` enlève et retourne le dernier élément de la liste

`L.index(x)` : Retourne la position du premier élément de la liste L ayant la valeur x.

`L.sort()` : Trie les éléments de la liste, en place.

`L.reverse()` : Inverse l'ordre des éléments de la liste, en place.

Chapitre 2

Tracer et mesurer des performances

2.1 Tracer avec Turtle

`turtle` est un module qui permet de tracer des figures géométriques à l'aide de mouvements élémentaires effectués par une ...tortue. La tortue est représentée par un icone Δ qui fixe sa direction de déplacement. Le tracé est créé par ses déplacements,

- en levant (`up()`) ou non (`down`) un crayon,
- déplacements contrôlés par des changements de direction : `right(angle)` et `left(angle)`,
- et des distances vers l'avant : `forward(distance)` ou l'arrière : `backward(distance)`.
- `goto(x,y)` permet de se déplacer au point (x, y) sans modifier la direction de la tête.
- `reset()` initialise la fenêtre graphique et `mainloop()` permet de la conserver à l'écran une fois le tracé terminé.

D'autres options (couleur et taille du trait, vitesse de déplacement, ...) se comprennent à l'aide de l'exemple suivant.

```
import turtle as t
"""Turtle : mouvements elementaires"""

def carre(c, colorstring):
    """dessine un carre de cote c et le remplit de color"""
    t.fillcolor(colorstring)
    t.begin_fill()
    for i in range(4):
        t.forward(c)
        t.right(90)
    t.end_fill()

# parametres deplacement
un = 30
deux = 2*un

# init : taille ecran en pixels, clear ecran, reset des variables
t.screensize(100,100)
t.clear()
t.reset()
# localiser (0,0)
t.goto(0,0)
t.dot()
t.up()

# un carre
t.goto(10,10)
t.down()
```

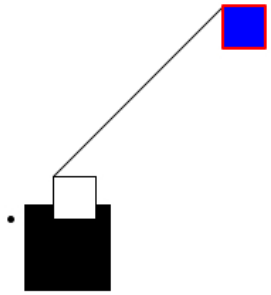
```
carre(deux, "black")

# on se deplace sans laisser de trace
t.up()
t.goto(un, un)
t.down()

# un carre plus petit
carre(un, "white")

# et un autre carre en couleur lie au precedent
t.goto(5*un,5*un)
t. pencolor("red")
t.pensize(2)
t.hideturtle() # pour aller plus vite
carre(un, "blue")

# pour que le trace reste a l ecran
t. mainloop()
```



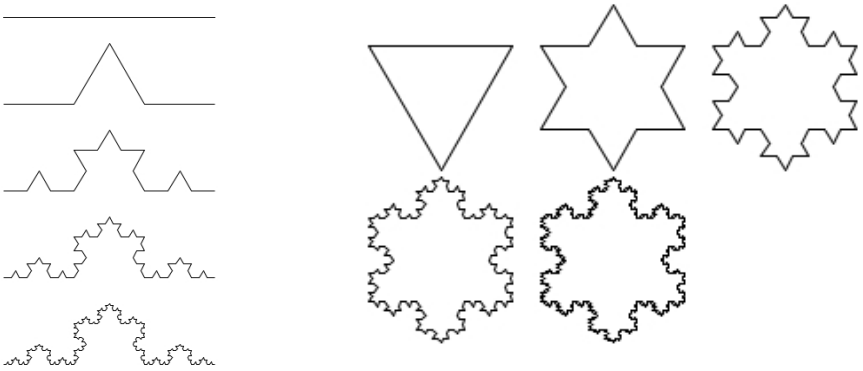
Plein d'autres détails sont bien sûr accessibles en ligne à : `https://docs.python.org/3/library/`

Utiliser `turtle` pour réaliser les tracés suivants.

Exercice 1. (P) Le flocon de Von Koch.
Le flocon de Von Koch (1906) est une courbe fractale (continue partout et dérivable nulle part) obtenue comme la limite de la transformation suivante appliquée récursivement à un segment de départ.

1. le segment est découpée en 3 parties égales,
2. le segment intermédiaire est remplacé par deux segments formant un triangle équilatéral avec le segment supprimé,
3. et ainsi de suite sur les 4 segments ainsi obtenus.

Voici les cinq premières courbes obtenues à partir d'un segment droit (à gauche) ou d'un triangle équilatéral : le flocon de Von Koch (à droite).



1. Un segment.
   (a) `segment-vk` : écrire la suite de commandes `turtle` qui trace la n-ième transformation d'un segment donné.
   (b) `segment-vk-n` : compléter `segment-vk` pour obtenir le tracé ci-dessus.
2. Le flocon ! On effectue la même transformation sur chacun des cotés d'un triangle équilatéral.
   (a) `flocon` : écrire la suite de commandes `turtle` qui trace la n-ième transformation d'un triangle équilatéral donné.

(b) `flocon-n` : compléter `flocon` pour obtenir le tracé ci-dessus.

Le module `turle` est aussi utilisé pour tracer les labyrinthes parfaits et les chemins de sortie de l'exercice 19.

2.2 Tracer avec `matplotlib`

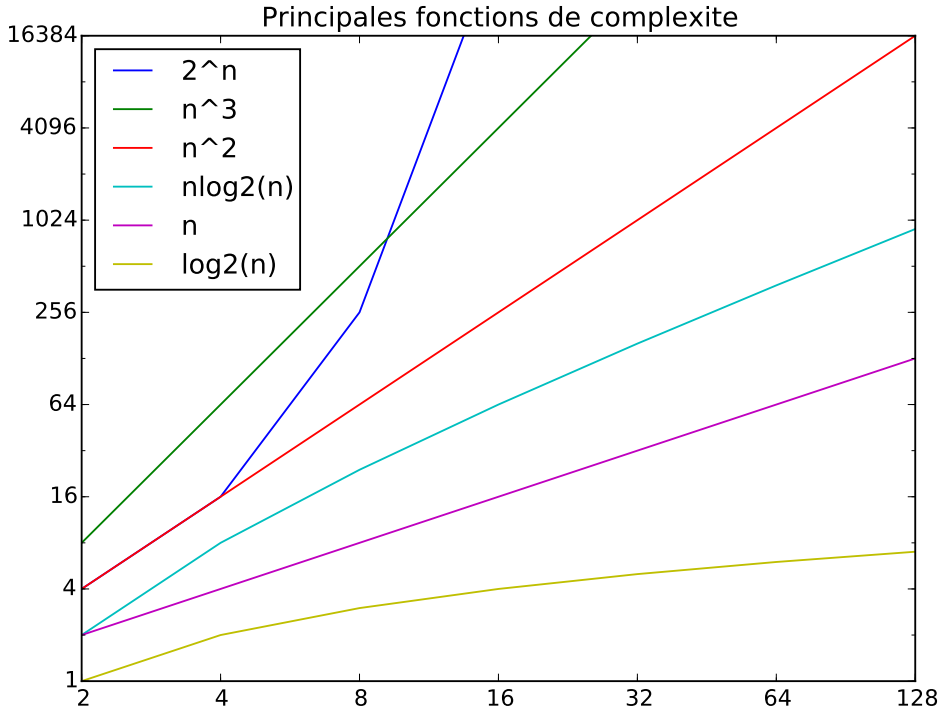
`matplotlib` est le module adapté à tous les types de tracés scientifiques : courbes, histogrammes, nuages de points, ...Ses options sont très nombreuses, de même que la documentation en ligne ou "livrée avec". L'utilisation basique consiste à tracer des courbes $x - y$ à partir de listes de points, d'un fichier de valeurs ou d'une expression, de choisir les bons axes, les bonnes échelles, d'indiquer les noms des courbes et des axes, de visualiser les tracés à l'écran et de les sauvegarder dans des fichiers (pdf ou png). Il peut être utile de tracer plusieurs "sous-figures" au sein d'une même figure. `numpy` est le compagnon naturel de `matplotlib` — `pylab` est la réunion de deux.

Une première prise en main est très bien décrite par :
`http://www.labri.fr/perso/nrougier/teaching/matplotlib/`
ou le chapitre 3 du *user guide* (qui comporte plus de 2500 pages).

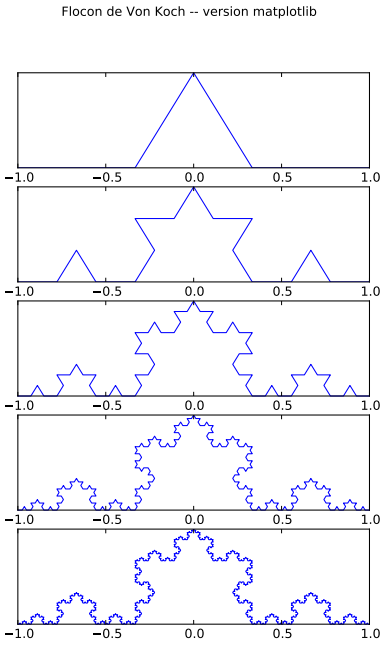
Exercice 2. (P) Principales complexités.
Cet exercice permet d'utiliser les commandes de base pour :

- afficher de courbes : `plot,figure,title,legend`,
- gérer les axes : `xticks,xlim,loglog,semilogx`,
- gérer les affichages : `show,savefig,close`.

1. Obtenir le tracé suivant des principales classes de complexités : $log_2 n, n, nlog_2 n, n^2, n^3, 2^n$ pour les premières puissances de 2 (par exemple de 2 à 128). Des échelles logarithmiques sont adaptées.



Exercice 3. (P) Encore le flocon de Von Koch.
Il plus rapide de calculer les coordonnées des sommets des transformés successifs du segment de départ, puis de les tracer comme une courbe. On obtient ainsi le tracé suivant.



1. Écrire la fonction `segvk` qui à partir des extrémités d'un segment $[a, b]$, par exemple définies par ses affixes complexes, calcule et retourne la liste (des affixes) des points significatifs d'une itération de Von Koch de $[a, b]$.
2. Écrire la fonction `listevk` qui applique une itération de Von Koch à une liste de segments.
3. Apliquer ces fonctions pour obtenir le tracé `matplotlib` des premières itérations du segment $[-1, 1]$. Il est agréable d'obtenir ces tracés sous la forme de sous-figures comme présenté ci-dessus.

2.3 Mesurer le temps d'exécution d'un algorithme

Mesurer le temps d'exécution d'un programme sur une machine actuelle est une une tâche beaucoup plus ardue qu'il n'y parait. En effet, l'impression que l'exécution de votre programme mobilise *à elle seule* les ressources de votre ordinateur est une illusion. Le temps que vous attendez pour obtenir votre résultat, même si cela vous parait instantané, est en fait partagé : pendant ce même temps, votre ordinateur effectue certainement de nombreuses autres tâches, par exemple liées au système d'exploitation, aux périphériques, à d'autres utilisateurs si votre machine gère plusieurs sessions, ...Par ailleurs, les unités de calculs et l'organisation (des niveaux) de la mémoire compliquent très fortement la mesure fiable des temps d'exécution et leur analyse.

En pratique, plus ce que vous voulez mesurer est court, plus la mesure sera incertaine et non reproductible. Pour ce qui nous intéresse, l'obtention d'une mesure significative consistera à répéter l'exécution dans une boucle, de mesurer le temps d'excution de *cette boucle* et d'en retenir la moyenne. Il

n'est pas inutile de répéter ce type de mesures et le cas échéant de traiter l'échantillon obtenu.

Les "gros problèmes" sont aussi sujets à remarque. Des tailles de données importantes nécessitent un volume important de transferts entre la mémoire et les unités de calcul. Ces transferts sont complexes et leurs temps est, d'une part non linéaire par rapport à la taille des données, et d'autre part beaucoup plus important que les temps de calcul souvent associés. A titre indicatif, il y a une facteur 10 entre ce temps de transfert et le temps d'un calcul arithmétique élémentaire. Ainsi, les mesures peuvent être surprenantes lorsqu'elles deviennent "dominées" par ces temps de transfert. Le modèle de nos analyses de la complexité des algorithmes ne prend pas en compte cet aspect très difficile.

Pour finir, ceci illustre la différence entre les propriétés d'un algorithme et celles de ses mises en oeuvres et, par la même occasion, confirme l'intérêt des notations asymptotiques $\mathcal{O}$, θ, Ω dont on a souvent signalé qu'elle masquent pas mal de détails.

Mesurer avec le module `time` Ce module fournit beaucoup de fonctions en rapport avec le temps : gestions de calendriers, des systèmes horaires, d'horloges ...Il permet aussi une estimation des temps d'exécution des programmes python (et autres). La qualité de cette estimation dépend des mises en oeuvre de ce module et des machines visées. Vous pouvez être relativement confiant sur vos PC-linux ou mac.

La fonction `time()` est annoncée avec une précision de la seconde, sans que ce soit garanti pour toutes les machines. Une seconde est suffisamment long pour effectuer 2 giga = $2 \cdot 10^9$ opérations ...Elle peut donc être utile pour mesurer des simulations ou des exécutions conséquentes.

La fonction `perf_counter()` est définie depuis Python 3.3. Lorsqu'elle est disponible, elle permet les mesures les plus fines possibles adaptées à la mesure des temps d'exécution de programmes (sans être pour autant exactes, ni reproductibles). Elle utilise des fonctions spécifiques aux architectures des machines qui exploitent les compteurs matériels.

Ces fonctions sont très simples à utiliser. On procède en deux appels qui encadrent la portion de code à mesurer.

- **`t0 = time.perf_counter()` :** le premier appel initialise une valeur initiale `t0` ;
- **`t = time.perf_counter()` :** le second appel mesure `t` à l'issue de l'exécution de la portion de code,

et `t-t0` est la mesure de ce temps d'exécution (moyennant les réserves précédentes). Ne pas oublier de boucler autour de la portion à analyser et de meurer à l'extérieur de cette boucle.

Exercice 4. (P) Compteurs de temps ...
Comparer `time` et `perf_counter` pour un algorithme linéaire.

Exercice 5. (P) Performance d'algorithmes de complexités diverses ...
Citer un algorithme d'algèbre linéaire pour chacune des complexités suivantes : linéaire, quadratique et cubique. Observer leurs temps d'exécution pour des ensembles de dimension correctement choisies. Proposer un tracé qui exhibe les complexités attendues.

Deuxième partie

Exercices

Chapitre 3

Remise en jambe et récursivité

3.1 Premières récursions

Exercice 6. Exponentiation rapide (introduit en cours).

1. (P) Écrire `expo_rapide` à l'aide de `divmod`.
2. (P) Ajouter un `print` pour tracer les appels successifs.
3. Quels sont appels à `expo(x,n)` évités par `expo_rapide(x,n)`.

Exercice 7. Fibonacci (introduit en cours).

1. (P) Ajouter un `print` pour tracer les appels successifs.

Exercice 8. Écrivons récursif !

1. (P) Écrire une fonction qui calcule 2^n pour n entier.
2. (P) Écrire une fonction qui calcule le somme des n premiers entiers.
3. Quelle sont les complexités de ces fonctions ?

Exercice 9. Euclide récursif et itératif . . .surtout.

1. (P) Écrire les versions récursive et itérative de l'algorithme d'Euclide vues en séance de façon à pouvoir comptabiliser (et tracer) les appels ou les itérations.
2. (P) Écrire des versions qui utilisent `divmod`.
3. Pour la version itérative :
   (a) Montrer sa terminaison.
   (b) Déterminer et prouver un invariant de boucle.
   (c) Prouver sa correction.
   (d) ($\star$) Étudier sa complexité.

Exercice 10. Syracuse . . . c'est loin ?.
L'exemple classique d'une fonction récursive dont on ne sait pas démontrer la terminaison est la suite de Syracuse. En voici une première version . . . économique.

$$\begin{array}{ll} s(0) = s(1) = 1 & \\ s(n) = s(n/2) & \text{si } n \text{ est pair,} \\ s(n) = s(3n+1) & \text{si } n \text{ est impair.} \end{array}$$

1. Premiers contacts.

(a) (P) Écrire une fonction qui calcule $s(n)$ et obtenir les 30 premiers termes de cette suite.

(b) (P) Modifier ce programme pour afficher les valeurs intermédiaires nécessaires au calcul de $s(n)$. Observer raisonnablement.

(c) (P) Modifier ce programme pour compter le nombre $c(n)$ de valeurs intermédiaires calculées dans $s(n)$.

(d) (P*) Tracer $(n, s(n), c(n))$ pour $n \in \{1, 30\}$. *Cette question nécessite d'utiliser matplotlib.*

(e) (P) Modifier le premier programme pour calculer et retourner la séquence de valeurs intermédiaires associée à $s(n)$.

2. Pour les curieux. Comme vous vous en doutiez, on conjecture que pour tout $N > 0$, $s(N) = 1$. La suite u_n des valeurs intermédiaires calculées à partir de $s(N)$ est la suite de Syracuse associée à N. Elle s'écrit :

$$\begin{array}{ll} u_0 = N & \\ u_{n+1} = u_{n/2} & \text{si } u_n \text{ est pair,} \\ u_{n+1} = 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{array}$$

(a) Tracer les valeurs intermédiaires pour un choix arbitraire de N. Observer en particulier les cas $N = 15$ et $N = 127$ — mais pas qu'eux ! Que se passe-t-il si $u_n = 1$?

(b) Vu la forme précédente, plusieurs notions sont définies autour de cette suite (source Wikipédia).
— Le temps de vol est le plus petit indice n tel que $u_n = 1$.
— Le temps de vol en altitude est le plus petit indice n tel que $u_n \leq N$.
— L'altitude maximale est la valeur maximale u_n de la suite.
Écrire des fonctions qui calculent ces notions pour un N arbitraire, puis jouer avec !

3.2 C'est vraiment parti !

Exercice 11. Décomposition binaire récursive.
La représentation binaire (aussi appelée la valeur binaire) d'un entier positif n est la suite de 0 et 1 (*bits*) notée $(n)_2 = b_p \cdots b_1 b_0 = \sum_0^p b_i 2^i$, où $b_i \in \{0, 1\}$. La suite de bits de la représentation binaire peut être stockée comme un tableau (ou une liste) de bits selon un ordre à préciser. Ainsi $(13)_2 = [1, 0, 0, 1]$ et $(16)_2 = [1, 0, 0, 0, 0]$ pour l'ordre décroissant des puissances de 2.
Bien sûr, la fonction python `bin(n)` calcule cette valeur et la retourne sous la forme d'une chaîne de bits (string en format binaire).

1. Propriétés préliminaires.

(a) (P) Écrire une fonction qui calcule p tel que 2^p est la plus grande puissance de 2 inférieure ou égale à n.

(b) Montrer que cette fonction termine.

(c) Déterminer et prouver un invariant de boucle.

(d) Quelle est la longueur minimale t du tableau nécessaire au stockage de la représentation binaire de l'entier positif n.

(e) Borner $n - 2^p$ pour $2^{p-1} \leq n < 2^p$.

(f) Écrire n en fonction de $n - 2^p$. En déduire une relation entre les tableaux qui stockent les valeurs binaire de $n, 2^p$ et $n - 2^p$.

2. Version récursive.

(a) Décrire les étapes d'un algorithme récursif, d'entrées deux entiers naturels n et t, qui calcule la valeur binaire de n et la retourne dans un tableau de longueur t. Dans un premier temps, on supposera la longueur t suffisante pour stocker des entiers positifs `int` codés en machine sur 32 bits.

(b) (P) Écrire cet algorithme sous la forme d'une fonction (récursive) python. Vérifier son comportement sur quelques exemples bien choisis.

(c) Prouver que cet algorithme termine et est correct.

(d) Borner le nombre d'appels récursif de cet algorithme. Expliciter les valeurs de n correspondantes.

(e) (P) Modifier le programme précédent pour observer ce nombre d'appels.

(f) (P) Modifier le programme initial pour que le tableau t soit de longueur arbitraire. Justifier votre choix.

Exercice 12. Les tours de Hanoï : première visite.

Le problème des *tours de Hanoï* (Édouard Lucas, 1883) consiste à trouver une stratégie pour déplacer une pile de n disques (troués en leur milieu) de diamètres tous différents ($n \geq 1$), d'une tour A vers une tour C en passant par une tour intermédiaire B. Au départ, les disques sont tous empilés sur la tour A par diamètre décroissant, comme indiqué sur la figure ci-dessous, pour $n = 3$.

Ensuite, les règles de déplacement sont les suivantes :

(R1) un seul disque est déplacé à chaque déplacement,

(R2) un disque est empilé uniquement sur un disque de plus grand diamètre ou sur une tour vide.

Ce problème est résolu récursivement grâce à l'algorithme `Hanoi(n,src,dst,int)` qui déplace n disques de la tour `src` vers la tour `dst` en utilisant la tour intermédiaire `int` :

— Si $n = 1$: déplacer ce disque de la tour `src` vers la tour `dst`,

— Si $n \neq 1$:

1) déplacer $(n-1)$ disques de la tour `src` vers la tour `int` en passant par la tour `dst`,
2) déplacer 1 disque de la tour `src` vers la tour `dst`,
3) et déplacer $(n-1)$ disques de la tour `int` vers la tour `dst` en passant par la tour `src`.

L'appel principal sera : `Hanoi(n,A,C,B)`.

1. Prise en main.
   (a) Pour $n = 3$, dérouler "à la main" l'algorithme `Hanoi(n,A,C,B)`. Expliciter l'arbre des appels récursifs et les déplacements successifs. Vérifier que le problème est ainsi résolu.
   (b) En déduire l'arbre des appels pour $n = 4$.
   (c) Combien de déplacements sont nécessaires à la résolution du problème pour $n = 3$. ? Et pour $n = 4$?
   (d) Expliciter des pré-conditions pour que `Hanoi` s'exécute correctement.
2. Codages et expérimentations.
   (a) (P) Coder une première version de l'algorithme récursif qui calcule et affiche la séquence de déplacements qui résolvent le problème des tours de Hanoï (sous la forme `A -> C`,...).
   (b) (P) Coder une autre version de `Hanoi` afin i) de compter les nombres de déplacements de chaque disque, leur nombre total pour un n arbitraire (attention en pratique !) et d'afficher ces déplacements.

(c) (P) Proposer et coder une dernière version de l'algorithme initial en remplacant l'étape 2) du cas $n \neq 1$ par un appel à l'algorithme `Hanoi`.

3. Correction et terminaison. Soit $P_1(n)$ la propriété :

   $P_1(n)$: l'algorithme `Hanoi(n,src,des,int)` déplace n disques de diamètre consécutif de la tour `src` vers la tour `des` en respectant les règles de déplacements (R1) et (R2).

   (a) Prouver la propriété $P_1(n)$, pour $n \geq 1$, et conclure que l'algorithme résoud les tours de Hanoï.

   (b) Déduire également que l'agorithme termine.

4. Complexité.

   (a) Montrer que pour tout entier n, $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

   (b) On note c le coût du déplacement d'un disque d'une tour vers une autre, et $C(n)$ le coût de la résolution de `Hanoi(n,A,B,C)`. Donner l'expression du nombre $C(n)$.

   (c) (*) En utilisant les propriétés des opérateurs asymptotiques, montrez les propriétés suivantes.

      i. Si $f(n) = 2^n + c$ avec $c \in \mathbb{R}$, alors $f = \Theta(2^n)$.

      ii. Si $f_1 = \Theta(g_1)$ et $f_2 = \Theta(g_2)$ alors $f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$.

   (d) (*) En déduire la complexité asymptotique de l'algorithme `Hanoi`.

Chapitre 4

Les piles

4.1 Exercices de base

Exercice 13. Piles à capacité bornée ou non (introduit en cours).

1. (a) (P) Pour les piles à capacité bornée, écrire les fonctions `creer_pile, empiler, depiler, sommet, est_vide, taille` en veillant au bon traitement des cas exceptionnels.
   (b) (P) Écrire `test_pile` qui utilise chacune de ces fonctions pour des arguments admissibles ou non.
2. (a) Écrire les algorithmes de création, modification et observation de piles à capacité non bornée. Commenter les principales différences avec le cas borné.
   (b) (P) Écrire les fonctions de la question 1a pour des piles à capacité non bornée.
   (c) (P) Reprendre `test_pile` et observer le comportement des traitements exceptionnels.

Exercice 14. Manipuler des piles.

1. (a) En utilisant exclusivement les fonctions de base des piles, écrire un algorithme qui duplique une pile et retourne cette copie.
   (b) Évaluer le nombre d'appels aux fonctions de base. Évaluer le coût de l'occupation mémoire selon que le système optimise ou non cette occupation.
   (c) (P) Écrire le programme `copier` correspondant.
2. (a) En utilisant exclusivement les fonctions de base des piles, écrire un algorithme retournant une pile qui contient, en ordre inverse, les éléments d'une pile `p` donnée, cette dernière n'étant pas modifiée (à l'issue du traitement).
   (b) Écrire le programme `inverser` qui effectue ce traitement.

Exercice 15. (P) Manipuler des piles (épisode 2).
Toujours en utilisant uniquement les fonctions de base des piles.

1. Écrire le programme `item` qui lit le n-ième élément d'une pile. La pile sera laissée inchangée et le cas où la pile n'est pas de taille suffisante sera traité.
2. Écrire le programme `tete_en_bas` qui place en fond de pile l'élément-sommet d'une pile non vide, les autres éléments restant dans l'ordre initial.

Exercice 16. Deux piles et un seul tableau.

1. Proposer une réalisation de deux piles dans un seul tableau `T` de façon à ce qu'on puisse empiler au choix sur l'une d'entre elle tant que le tableau n'est pas plein.
2. (P) Écrire `empiler(e,b)` qui empile l'élément `e` sur l'une ou l'autre des piles selon la valeur du booléen `b`.

4.2 Premières applications

Exercice 17. Les tours de Hanoï : seconde visite.

On revient sur les tours de Hanoï de l'exercice 12.

1. (P) Reprendre les programmes précédents pour calculer les états successifs des tours A, B, C au fur et à mesure de la résolution en utilisant le module de piles de l'exercice 13 pour représenter et manipuler les tours.
2. (P) Faire de même en comptant les nombres de déplacements de chaque disque, leur nombre total pour un n arbitraire et en affichant ces déplacements.

Exercice 18. (P) Mots bien parenthésés.

1. (P) Écrire un programme `parentheses` qui vérifie la correction d'un mot parenthésé et qui affiche les paires des indices associés (ouvrant/fermant) comme vu en cours.
2. (P) Modifier le programme `parentheses` pour traiter des mots constitués de parenthèses "(,)", de crochets "[,]" et d'accolades "{,}". Le mot "([ ] { })" est bien parenthésé à la différence de "({] } [ )".
3. (P) Modifier le programme `parentheses` pour traiter des mots constitués de parenthèses et d'autres caractères. Le mot "3*(4-5*(6+7)-8)" est bien parenthésé mais pas "3*(4-5)*6)".

4.3 Applications plus conséquentes

Exercice 19. Labyrinthe parfait.
Sur la grille discrète $[0, n-1] \times [0, n-1]$, on considère les n^2 noeuds $c = (i, j)$. Deux noeuds de la grille sont voisins si il existe une arête horizontale ou verticale les reliant. Un chemin entre deux noeuds c_1 et c_2 est la suite de noeuds successivement voisins $c_1, c_i, c_j, \ldots, c_k, c_2$, d'extrémités c_1 et c_2.
Un *labyrinthe parfait* est tel que, pour toute paire de noeuds de la grille, il existe un et un seul chemin entre ces noeuds. Le labyrinthe d'entrée c_1 et de sortie c_2 sera noté :

$$(c_1, c_i), (c_i, c_j), (c_i, c_k), \ldots, (c_k, c_r), (c_j, c_l), (c_l, c_2)$$

ou aussi :

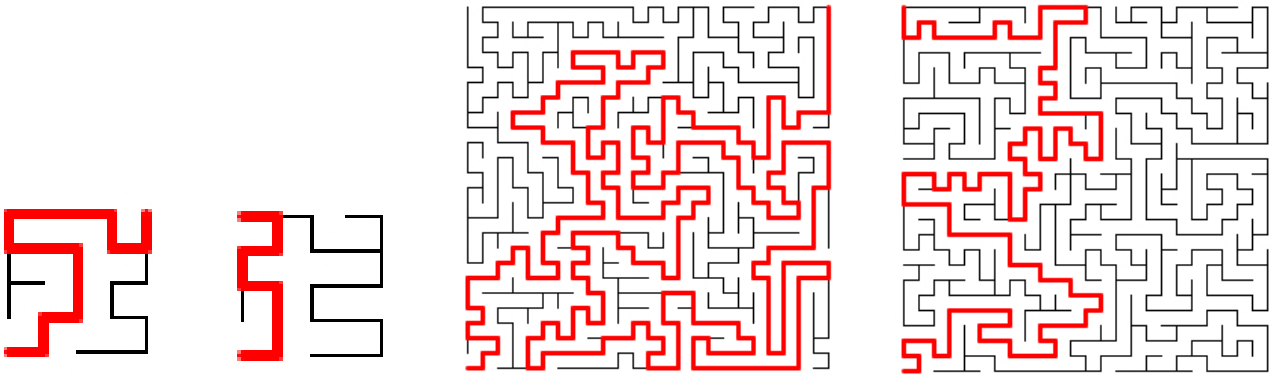
$$(c_1, c_i), (c_i, c_j), (c_j, c_l), (c_l, c_2), (c_i, c_k), \ldots, (c_k, c_r).$$

Ici le chemin $c_i, c_k, \ldots, c_r$ où $r \neq 2$ mène à un cul-de-sac, et le chemin c_1, c_i, c_j, c_l, c_2 est l'unique chemin qui permet de sortir du labyrinthe.
Voici des exemples de labyrinthes parfaits de dimension 5×5 et 25×25, et de chemins qui entrent au coin inférieur gauche $(0, 0)$ et sortent aux coins supérieurs droits ou gauche – resp. $(4, 4), (24, 24)$ ou $(0, 4), (0, 24)$.
À la différence des livre de jeux pour enfants, cheminer dans ces labyrinthes consiste à suivre les "lignes-chemins" plutôt qu'à se déplacer entre des "lignes-parois".

La construction d'un tel labyrinthe consiste à visiter une seule fois tous les noeuds de la grille. L'approche suivante profite de la structure de pile. On se donne :

— une `grilleDesVisites` de booléens pour identifier les noeuds déjà visités,
— une pile `p` des noeuds non encore visités,
— une pile `laby` qui stocke le labyrinthe en construction.

On part du noeud d'entrée, par exemple $(0, 0)$, que l'on empile sur `p` et que l'on marque dans `grilleDesVisites`.
Tant que la pile `p` n'est pas vide :



i) on dépile le “noeud-sommet” c de la pile `p`,

ii) on identifie ses noeuds voisins non encore visités et si il y en a au moins un :

. on en choisit un aléatoirement : s,

.. on empile le chemin (c, s) sur `laby`,

... on empile c puis s sur `p`,

.... on marque s dans `grilleDesVisites` et on reprend.

1. Les questions suivantes vous accompagnent dans la mise en oeuvre de cet algorithme. La seule difficulté est d’éviter de sortir de la grille.
   (a) Définir les trois structures de données `grilleDesVisites`, `p`, `laby` et des fonctions d’affichage adaptées.
   (b) Écrire `visiter` qui met à jour la `grilleDesVisites`.
   (c) Écrire `dejaVisite` qui retourne l’état de visite d’un noeud donné.
   (d) Écrire `voisins` qui retourne la liste des noeuds voisins et non encore visité d’un noeud donné.
   (e) Écrire `choisir` qui retourne aléatoirement le prochain noeud à visiter.
   (f) Finalement, écrire `labyrinthe` qui construit et retourne un labyrinthe parfait sur une grile carrée de côté de longueur n arbitraire.
2. Utiliser le module `turtle` pour tracer ce labyrinthe.
3. (*) Compléter `labyrinthe` pour :
   (a) qu’il retourne le chemin qui mène de l’entrée $(0, 0)$ jusqu’à la sortie en $(n-1, n-1)$,
   (b) puis d’une entrée à une sortie arbitrairement choisies.

Exercice 20. (P) La magie des coupes et des mélanges d’un jeu de cartes.  

Nous étudions des manipulations de cartes, d’apparences magiques, proposées par Gilbreath entre 1958 et 1989. Pour cela nous utilisons trois articles récents de Lachal et Schott parus dans *Quadrature* en 2012 et 2013 — chercher “cartomagie, Lachal, Schott” sur le net.

1. Toujours en utilisant uniquement les fonctions de base des piles.

(a) Écrire le programme `couper` qui prend une pile `p` et coupe ses n élements de tête, $0 <$ $n <$`taille(p)`, et les regroupe dans le même ordre en une seconde pile, c'est-à-dire comme lorsqu'on coupe et "ré-empile" un jeu de cartes. On ajoute le cas $n = 0$ pour que la valeur effective de n soit choisie au hasard dans $[\![0, \texttt{taille(p)}[\![$. Par exemple la pile `[0,1,2,3,4]` est coupée en `[0,1,2]` et `[3,4]` si $n = 2$ (la tête de pile est la valeur de droite dans cette notation sous forme de liste).

(b) Écrire le programme `melanger` qui prend en argument deux piles `P1` et `P2` et retourne une troisième pile obtenue à partir du tirage aléatoire d'éléments-sommet de `P1` ou `P2`, et ce tant que ces piles le permettent — ces deux piles finissent vides. Ce mélange sera rendu stable : l'ordre relatif des éléments des piles initiales est conservé dans la pile mélangée.

Les mélanges `[3,0,1,4,2]` et `[0,1,3,2,4]` sont par exemple obtenus à partir des deux piles de la coupe précédente. Les fonctions `copier` et `inverser` de l'exercice 14 peuvent être utilisées.

Un tel mélange est appelé *mélange à la queue d'aronde* ou *mélange américain* dans les références mentionnées.

(c) `test` : écrire un programme qui teste `couper` et `melanger` pour des valeurs arbitraires des paramètres proposées par l'utilisateur, et ce tant qu'il le désire.

2. Lachal et Schott énoncent le premier principe de Gilbreath comme suit.

> "Si un jeu de cartes classé en rouges et noires alternées une à une est coupé en deux paquets, avec une carte noire sur la face de l'un des paquets et une carte rouge sur la face de l'autre, et si ces deux paquets sont mélangés l'un dans l'autre au moyen d'un mélange à la queue d'aronde, alors chaque paire de cartes consécutives du jeu sera composée d'une carte rouge et d'une carte noire."

`magie1` : construire un paquet de cartes en empilant p fois les mêmes paires de cartes. Par exemple, $p = 16$ paires rouge-noir pour construire un paquet de 32 cartes à deux couleurs successivement alternées.

`couper` ce paquet en deux paquets tels que leurs têtes soient de couleur différente : par exemple en choisissant n impair.

`melanger` ces paquets et observer.

Magie : comme annoncé par Gilbreath, le paquet final contient toujours p blocs de paires de cartes, ces paires pouvant apparaître dans un ordre différent. Ici : 16 paires rouge-noir ou noir-rouge.

Il pourra être commode de pouvoir afficher le contenu d'une pile (d'un jeu de cartes) par paquets de k éléments consécutifs.

3. Lachal et Schott rapportent le "truc" suivant pour ne pas contraindre le résultat de la coupe précédente, *i.e.* imposer des têtes de couleur différente.

> "Si l'on préfère laisser le spectateur couper le jeu faces en bas sans se soucier de la couleur des cartes qui se trouvent sur la face des deux paquets coupés, il suffit, une fois le mélange effectué, de procéder à l'ajustement suivant : prenez le jeu, retournez-le faces en haut, [...], repérez deux cartes de même couleur, [...], coupez au milieu de cette paire et complétez la coupe [...]. "

`magie2` : Modifier `magie1` en `magie2` de façon à vérifier que ce "truc" fonctionne correctement. "Compléter la coupe" signifie "ré-empiler" les 2 paquets après la coupe.

4. Le second principe de Gilbreath dit :

> "Si deux séries de cartes classées en ordre inverse l'une de l'autre sont mélangées ensemble avec un mélange à la queue d'aronde, les deux moitiés du groupe qui en résulte seront chacune composées de cartes similaires à celles des séries originelles."

(a) `magie3` : choisir r lettres et construire deux paquets composés de deux séries : r minuscules et r majuscules. Inverser la seconde série. Mélanger les deux séries et vérifier que le second principe de Gilbreath s'applique.

(b) Modifier `magie3` pour ne manipuler qu'un seul paquet de "cartes".

(c) Modifier `magie3` pour vérifier que si le principe reste vrai après un nombre arbitraire de coupes et de mélanges. Justifier votre observation.

5. Le principe précédent a été généralisé par Hudson (1966).

   "Lorsqu'un nombre quelconque de cartes sont classées en séries répétées, elles peuvent être mélangées entre elles au moyen d'un mélange à la queue d'aronde sans altérer le contenu de chaque série, à condition que les cartes de l'un des deux paquets qui doit être mélangé avec l'autre soient inversées."

   `magie4` : proposer une vérification de cette généralisation.

Chapitre 5

Trier

5.1 Les tris vus en cours

Exercice 21. À la main
Traiter les questions suivantes pour chacun des algorithmes vus en cours : tri insertion, tri fusion et tri rapide.

1. Détailler leur "déroulement à la main" sur le vecteur `[15, 4, 2, 8, 17, 23, 0, 1]`. Comptabiliser les comparaisons et les affectations et comparer aux résultats du cours.
2. Proposer un exemple de pire et de meilleur cas pour un vecteur d'entrée de taille 8.

Exercice 22. (P) Tri insertion

1. Écrire `triinsertion` vu en cours.
2. Que peut-on dire de la complexité du tri insertion lorsqu'on double le nombre de valeurs à trier ?
3. Observations expérimentales des complexités.
   (a) Modifier `triinsertion` pour compter le nombre de comparaisons.
   (b) Écrire des fonctions qui génèrent les meilleurs et les pires cas du tri insertion pour un nombre de valeurs à trier (dimension) arbitraire.
   (c) Écrire une fonction qui génére un échantillon de cas quelconques de dimension arbitraire.
   (d) Écrire `testcomptriinsertion` pour illustrer les complexités vues en cours. Le résultat attendu est un tracé matplotlib pour des dimensions qui doublent de 100 à 6400 (par exemple).
      i. Complexités des meilleurs et des pires cas ;
      ii. complexité moyenne obtenues sur des échantillons de cas quelconques de façon à illustrer le résultat vu en cours.
4. Reprendre la question précédente en observant les temps d'exécutions du tri insertion. Proposer un tracé qui exhibe les résultats de la question 2.

Exercice 23. (P) Tri fusion

1. Que peut-on dire de la complexité du tri fusion lorsqu'on double le nombre de valeurs à trier ?
2. Écrire les différentes versions de fusion vues en cours. On suppose que les arguments sont des (sous-)tableaux de longueur arbitraire mais triés.
   (a) `fusion0` qui fusionne les tableaux `t1` et `t2` en un tableau `t`.
   (b) `fusion1` qui fusionne les sous-tableaux `t[d,m]` et `t[m,d]` avec recopie et sentinelles.

(c) `fusion2` qui fusionne les sous-tableaux `t[d,m]` et `t[m,d]` avec recopie.

(d) Dans chacun des cas, écrire `testfusion` adapté à des vérifications.

3. (a) Écrire `trifusion` basé sur l'une des fusions précédentes.

   (b) Écrire une version de `trifusion` qui explicite la succession des appels récursifs.

   (c) Écrire `testtrifusion` pour se rassurer.

4. (a) Modifier `trifusion` pour compter le nombre de comparaisons et d'affectations (des éléments du tableau d'entrée).

   (b) À la manière des questions 3 et 4 de l'exercice 22, écrire `testcomptrifusion` et `testtimestrifusi` pour illustrer les complexités (pire cas, meilleur cas, en moyenne) vues en cours.

5. Analyse plus fine : fusion avec recopie et sentinelles *vs.* fusion avec recopie seuelement.

   (a) Écrire `trifusion1` et `trifusion2` qui utilisent respectivement les fusions avec copie locales et sentinelles ou sans ces dernières, resp. `fusion1` et `fusion2` de la question 2.

   (b) Modifier chacune de ces versions pour compter le plus précisement possible nombre de comparaisons.

   (c) Reprendre la question 4b en observant plus spécialement les comparaisons dans le pire cas et en moyenne. Faire le lien avec ce qui a été présenté en cours.

6. Reprendre la question précédente en observant les temps d'exécutions du tri fusion. Proposer un tracé qui exhibe les résultats de la question 1.

Exercice 24. (P) Tri rapide

1. Que peut-on dire de la complexité du tri rapide lorsqu'on double le nombre de valeurs à trier ?
2. Écrire `trirapide` vu en cours.
3. Modifier `trirapide` pour compter le nombre d'appels récursifs.
4. (a) Modifier `trirapide` en `trirapide_comp` pour compter le nombre de comparaisons et d'affectations (des éléments du tableau d'entrée).

   (b) La fonction `permutations` de la bibliothèque `itertools` retourne, sous la forme d'un itérateur, l'ensemble des permutations des composantes d'un vecteur d'entrées. Par exemple, la boucle :

   ```
   for t in permutations([0, 1, 2]):
       print(t)
   ```

   retourne les 6 t-uplets suivants :

   ```
   (0, 1, 2)
   (0, 2, 1)
   (1, 0, 2)
   (1, 2, 0)
   (2, 0, 1)
   (2, 1, 0)
   ```

   Exploiter cette fonction pour tester exhaustivement `trirapide_comp` – be careful : size matters !

5. À la manière des questions 3 et 4 de l'exercice 22, écrire `testcomptrirapide` et `testtimetrirapide` pour illustrer les complexités (pire cas, meilleur cas, en moyenne) vues en cours. Proposer les tracés adaptés.

Exercice 25. Comparaison des algorithmes de tri
Reprendre les trois exercices précédents pour produire une courbe qui permet de comparer les timings moyens des algorithmes de tri vus en cours.

5.2 Autres algorithmes de tri et compléments

Exercice 26. Tri par insertion dichotomique

L'étape d'insertion du tri par insertion peut être réalisée par dichotomie.

1. Justifier pourquoi.
2. Quelle modification apporter à la preuve de la correction du tri par insertion ?
3. Quelle est la complexité en espace de cet algorithme ?
4. Quelle est la complexité en temps dans le pire cas de cet algorithme ? Commenter et expliciter un pire cas.
5. (P) Coder ce tri par insertion dichotomique. Observer le gain par rapport à la version classique.

Exercice 27. Autres versions du tri rapide

1. Autre choix détermiste.
   (a) (P) Écrire une version du tri rapide qui choisisse comme pivot la première valeur du tableau à trier.
   (b) Prouver la correction de cette version.
   (c) Quels sont les meilleur et pire cas ?
2. Version randomisée.
   (a) (P) Écrire une version du tri rapide qui choisisse comme pivot une valeur aléatoirement choisie dans le tableau à trier.
   (b) Expliciter l'invariant de cette version.
   (c) Qu'en est-il des meilleur et pire cas ?

Exercice 28. Tri par sélection

Le tri par sélection consiste à parcourir successivement tous les éléments du tableau pour en extraire la valeur minimale et la placer en tête de tableau (tri par ordre croissant). Ce principe est répété itérativement sur le reste du tableau jusqu'à obtenir le tableau complétement trié.

1. Proposer deux stratégies pour la gestion de la valeur minimale dont une de complexité en espace constante (et minimale) pour tout type d'éléments de tableau.
2. (P) Coder cet algorithme.
3. Démontrer la correction de cet algorithme.
4. Évaluer sa complexité en espace. Qu'en déduire ?
5. Évaluer sa complexité en temps en exhibant les meilleur et pire cas.

Exercice 29. (P) Application du tri par sélection

Le tri par insertion est souvent utilisé pour accélérer les autres algorithmes de tri lorsque le nombre d'éléments à trier est petit, c'est-à-dire inférieur à une certaine taille fixée à l'avance — à 5 par exemple.

1. Modifier le tri rapide avec cette accélération. Observer le gain.
2. Modifier le tri fusion avec cette accélération. Observer le gain.

Exercice 30. Tri à bulles

Le tri à bulles consiste à parcourir le tableau de la gauche vers la droite en permutant successivement chaque paire d'éléments consécutifs "mal" ordonnée. Ce parcours est répété tant que des permutations ont lieu.

1. Appliquer ce principe à la main sur le vecteur `[15, 4, 2, 8, 17, 23, 0, 1]`.
2. (P) Coder l'algorithme après avoir codé une fonction de permutation.
3. Quel est l'effet du premier parcours du tableau ? du second ? Expliquer sa dénomination.
4. Prouver la correction de l'algorithme.
5. Évaluer sa complexité en espace. Qu'en déduire ?
6. Évaluer sa complexité en temps en exhibant les meilleur et pire cas.

Exercice 31. Un tri linéaire : le tri par dénombrement

L'optimalité de la complexité semi-log des tris par comparaison peut être améliorée si on peut trier sans . . .comparer.
Le tri par dénombrement s'applique quand on connait toutes les valeurs du tableau à trier. Par exemple : $t[i] \in \{1, p\}$ pour $i \in \{1, n\}$. Il suffit alors de compter le nombre d'occurrences dans t de chacune des p valeurs possibles.

1. (P) Écrire et coder cet algorithme.
2. Quelle est sa complexité en temps dans le pire cas ? Exhiber un tel pire cas.
3. Qu'en est-il dans le meilleur cas ?
4. Quelle est sa complexité en espace ? Qu'en déduire ?

Exercice 32. Un autre tri linéaire : le tri-base ou tri radix ou encore *radix sort*

Compléter votre culture et affuter votre anglais en consultant la page wikipedia :
`http://en.wikipedia.org/wiki/Radix_sort`

Chapitre 6

Couleurs, images et traitements

6.1 Principes

Nous utiliserons les modules pyplot et image de matplotlib, les fichiers d'image et les tableaux numpy. Pour cela, il faut auparavant "avoir tout le matos".
Soit : importer `numpy`, `matplotlib.image` et `matplotlib.pyplot` qui seront respectivement appelés : `np`, `plt` et `mpimg`

Traitement basique. Les 3 grandes étapes du traitement avec matplotlib :

`file` → `np.ndarray` → `plt.img` → `file`

`mpimg.imread()` : file → ndarray
Lire un fichier `jpeg` ou `png` (par exemple) contenant une image. L'objet ndarray `t` créé est un tableau numpy. Le type de ces valeurs dépend du format du fichier image. Il est décrit par l'attribut `dtype`.

`float32` avec des valeurs comprises entre 0.0 et 1.0 si file est de type `png` ;

`unint8` avec des valeurs généralement comprises entre 0 et 255 pour les autres formats de fichiers image (`jpeg` en particulier). Attention : dans ce cas, l'arithmétique de ces valeurs est celle de $\mathbb{Z}/256\mathbb{Z}$.

C'est une étape de "pixelisation" : le tableau `t` est un ensemble de lignes ; chaque ligne est un ensemble de valeurs. Ces valeurs sont les pixels. Le tableau `t` permet tous les traitements numériques possibles ; voir l'exemple suivant et les exercices.

`plt.imshow()` : ndarray → img
Convertir un tableau numpy en un objet-image.
`plt.show()` permet alors de visualiser l'image en cours et, le cas échéant, de corriger son traitement.

`plt.savefig()` : img → file
Enregistrer l'objet-image dans un fichier png, pdf, ps, eps ou svg. :
Attention : cette commande est incompatible avec `plt.show()` (au moins dans mon environnement). Commenter cette dernière avant l'étape de sauvegarde.

Un exemple :

```
""" traitement image avec matplotlib.image et pyplot"""
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

f = 'arago.jpg'
t = mpimg.imread(f)  #t est un objet ndarray
# traitement global tres simple :) et en place
```

```
t = 255 - t
#
img = plt.imshow(t) #img est une image matplotlib

plt.axis('off') # pas d'axes x,y avec echelles
# choisir une des deux options suivantes (choix exclusif)
#plt.show()
plt.savefig('arago-inv.png', transparent=True, frameon= False, bbox_inches=
    'tight', pad_inches=0.0) # ou .pdf si on veut
```

Remarque. On est parti d'un fichier jpeg sans obtenir un fichier transformé de même format. D'une part, le format png (compression sans perte) est tout à fait adapté aux affichages numériques et les formats pdf, ps (eps) adaptés aux affichages papier. Si du jpeg est vraiment nécessaire, plusieurs solutions sont possibles dont l'utilisation de la bibliothèque `PIL` à la place de `matplotlib`. C'est en fait cette dernière qui est appelée par `matplotlib` pour les formats autres que png. On peut donc espérer l'intégration totale de ces traitements dans `matplotlib` dans des prochaines versions.
Mais tout ça, c'est de la bidouille : les concepts "fichier/objet image/tableau" sont invariants !

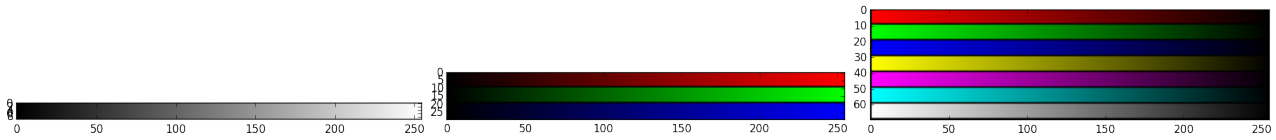
Tous les détails et les nombreuses autres commandes sont à l'URL : `http://matplotlib.org/users/image_tutorial.html`

6.2 On commence en couleurs !

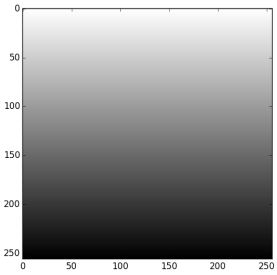
Exercice 33. (P) Des palettes en barres, en carrés et un premier traitement.

Plusieurs petits programmes pour être à l'aise avec les couleurs et l'attirail de traitement.

1. Écrire les programmes qui produisent les affichages suivants sous forme de barre 1D (ou presque) :
   (a) le spectre des gris ;
   (b) le spectre des trois couleurs R, V, B ;
   (c) le spectre précédent complété des 4 mélanges de couleurs possibles.

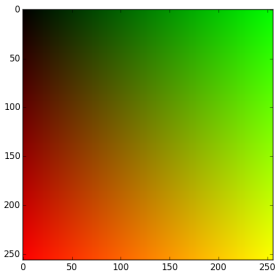
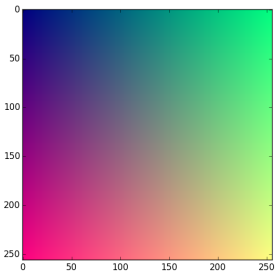
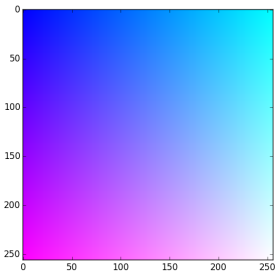


2. Générer et tracer une palette carrée de gris.
3. La totalité des couleurs s'observe en 3D. Des palettes presque complètes sont possibles en 2D en paramétrant la valeur d'une des couleurs, le bleu par exemple.



(a) Écrire une fonction `palettecouleurpart(b)` qui génère (sans tracé) une palette 2D de couleurs (R,V,B) pour une valeur de paramètre de B.

(b) Écrire la fonction de tracé correspondante qui donne par exemple les figures suivantes pour `b`$= 0, 128, 255$.

6.3 Des images et du traitement

Le traitement avec des ndarray en `float32` issus du format `png` est moins délicat que pour les `uint8` obtenus à partir des autres formats, `jpg` en particulier.
Les images `arago.jpg` et `ballon.png` seront par exemple utilisées dans les traitements suivants.

Exercice 34. (P) Griser des photos couleur `png` et `jpeg`.

1. Les deux triplets (α, β, γ) suivants définissent des transformations classiques d'un pixel (R, V, B) en un niveau de gris $G = \alpha R + \beta V + \gamma B$. Pour les images naturelles, $(\alpha, \beta, \gamma) = (0.2125, 0.7154, 0.0721)$ et pour les images numériques : $(\alpha, \beta, \gamma) = (0.299, 0.587, 0.114)$.
2. Proposer et coder une solution simple pour transformer une image couleur jpeg (`uint8`) en niveaux de gris. Comparer l'image obtenue avec celles issues qu'un choix arbitraire de R, V ou B.
3. À partir de l'image en niveaux de gris, générer une image partagée horizontalement en 2 parties : l'une originale et l'autre en négatif. Généraliser ce traitement pour un nombre arbitraire de découpages.

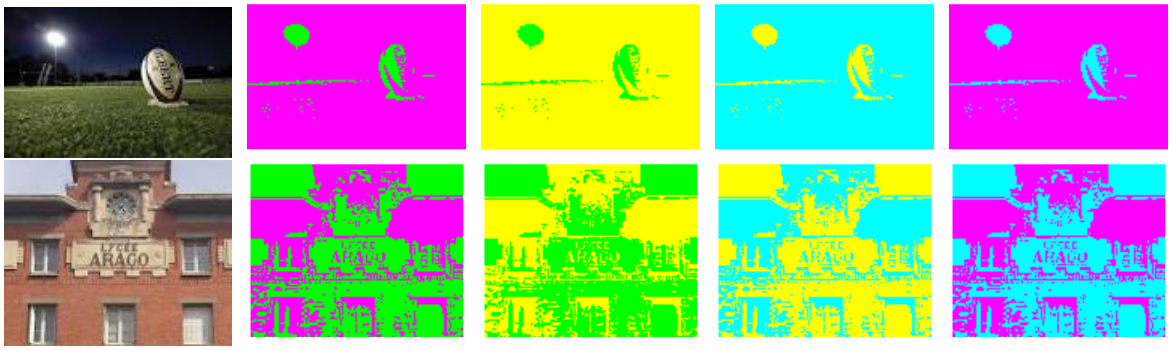
Exercice 35. (P) Des traitements classiques
Réaliser les traitements suivants à partir d'une image en niveaux de gris.

Contrastes 1. La solution "brute force" consiste à générer une image en noir et blanc. Pour cela, il suffit de choisir un seuil, de "blanchir" les pixels plus clairs que ce seuil et de noircir les autres. On pourra essayer plusieurs valeurs de seuil et discuter de leur intérêt pour des images plutôt claires ou plutôt sombres.

2. Définir une fonction qui augmente le contraste de façon plus continue que la précédente. Essayer d'abord sur $[0, 1]$ puis sur $[0, 255]$. Coder et observer.
3. Proposer et coder une stratégie pour diminuer le contraste.

Luminance Ajouter ou retrancher une constante à la valeur de chaque pixel. Coder et observer.

Pop Art et wharolitude Proposer et coder une stratégie pour obtenir des images de ce genre :



Réduire la taille Proposer et coder une stratégie pour diminuer, par 2 par exemple, la taille d'une image.

Fusionner deux images 1. Préparatifs : dégager la table et poser un objet, votre stylo par exemple. Prendre une photo de la table avec cet objet de façon à ce que l'objet ne soit pas omniprésent par rapport à l'ensemble de la photo. **Sans trop bouger** l'appareil, déplacer significativement l'objet et prendre une autre photo.
2. Fusion additive : après avoir transformé ces photos en niveaux de gris, les fusionner en une seule image par addition pixel avec pixel.
3. Fusion soustractive : à quoi peut-elle servir ? Essayer en bougeant très très peu le stylo entre les 2 photos et toujours sans bouger l'appareil !

Troisième partie

Devoir maison

Chapitre 7

Novembre 2014

Exercice 1. Dans toutes cette partie, on s'interdit l'usage de la fonction `min` préprogrammée en Python et permettant d'obtenir directement le minimum d'une liste donnée en argument. En revanche, on se donne la fonction `mini` suivante, écrite en Python.

```
1  def mini(t):
2   "'Calcule le minimum d'un tableau d'entiers ou de flottants."'
3   if len(t) == 0:
4      return None
5   p=t[0]
6   for i in range(len(t)):
7      if t[i] <= p:
8         p = t[i]
9   return p
```

1. Expliquer le déroulement pas à pas (évolution de la valeur des variables) lors de l'appel `mini([6,2,15,2,15])`, puis donner la valeur renvoyée.
2. Prouver que lorsque `t` est une liste non vide d'entiers ou de flottants, `mini(t)` renvoie la valeur minimale des éléments de `t`. On exhibera un invariant de boucle précis.
3. Évaluer la complexité temporelle de l'appel `mini(t)` en fonction du nombre `n` d'éléments de `t`.
4. Proposer une modification de la fonction `mini` pour que la valeur renvoyée soit le maximum et non le minimum. On pourra utiliser la numérotation des lignes pour préciser le lieu d'éventuelles modifications et ainsi éviter de réécrire toute la fonction.

   On souhaite récupérer non plus le minimum d'une liste mais la (une) position dans le tableau où le minimum est atteint. Dans l'exemple vu plus haut, il y a deux positions où ce minimum est atteint : 1 et 3.
5. Expliquer le principe d'une fonction réalisant cette opération et en particulier le rôle des variables manipulées.
6. Cette question ne sera lue que si la question précédente a été traitée. Écrire une fonction `position_mini` réalisant effectivement cette opération.
7. Préciser l'indice renvoyé si le minimum est présent plusieurs fois dans la liste. Proposer une modification permettant de changer ce comportement

   On souhaite maintenant déterminer la valeur minimale d'un tableau bidimensionnel d'entiers/de flottants. Un appel de cette fonction pourrait être :

   ```
   >>> mini2D([[10,3,15],[5,13,10]])
   3
   ```

8. Expliquer le principe d'une fonction réalisant ce travail.
9. Programmer effectivement cette fonction (on supposera les listes internes de taille non nulle).
10. Évaluer la complexité temporelle de cette fonction.

    On souhaite, partant d'une liste constituée de couples (chaîne, entier), déterminer la/une chaîne pour laquelle l'entier associé est minimal :

    ```
    >>> chaine_mini([['Tokyo',7000],['Paris',6000],['Londres',8000]])
    'Paris'
    ```

11. Écrire une fonction `chaine_mini` réalisant effectivement cette opération.
12. Écrire enfin une fonction `majores_par` prenant en entrée une liste `t` d'entiers ainsi qu'un entier seuil et renvoyant le nombre d'éléments de `t` majorés (au sens strict) par seuil :

    ```
    >>> majores_par([12,-5,10,9],10)
    2
    ```