Le but est de réviser les algorithmes et notions de base du programme d'informatique de PCSI et de PSI. L'idée est de d'abord vérifier que l'on connait son cours et passer la question si on est certain à 100% de connaitre la réponse ou savoir écrire le code demandé en Python. Sinon on se force à l'écrire.

LES VARIABLES

(*Compléments sur ces notions aux pages 6 à 9 du formulaire d'informatique PCSI sur Cahier de Prépa*)

1. Quelle commande permet d'accéder au dernier élément d'une liste (ou chaine de caractères) `LC` ?
2. Donner 2 différences fondamentales entre les listes de caractères et les chaines de caractères, qui font que l'on pourra préférer utiliser l'une plutôt que l'autre dans un script Python.
3. Donner un avantage, puis un inconvénient des dictionnaires sur les listes/chaînes.
4. Soit `x` un flottant, on veut vérifier si `x` est nul. Le test `x==0` est-il adéquat ? Justifier.
   Si vous avez répondu non, quel test feriez-vous ?

LES ALGORITHMES DE TRI

(*réponses p.23 du formulaire d'informatique sur Cahier de Prépa*)

5. Donner les noms des algorithmes de tri que vous connaissez.
6. Donner la complexité de ces tris.
7. Quel est le tri le plus rapide en moyenne ? Quel(s) principe(s) utilise-t-il ?

LE PROGRAMME DE PSI

8. Rappeler les 5 grands thèmes d'informatique au programme de PSI.
9. Rappeler deux exemples classiques étudiés en programmation dynamique.
10. Quel différence fondamentale y a-t-il entre les algorithmes « d'intelligence artificielle » KNN et celui des k-moyennes ?

LES ALGORITHMES « FACILES » CLASSIQUES

11. Soit `texte` une chaîne de caractères, écrire une fonction `Presence(lettre,texte)` qui vérifie si `texte` contient le caractère `lettre`.
12. Soit `L` une liste de flottants, écrire une fonction `moy(L)` qui calcule la moyenne des éléments de `L`.
13. Ajouter dans `moy(L)` la signature ainsi qu'un test d'assertion qui vérifie que `L` est bien une liste et qu'elle est non vide.
14. Soit `L` une liste de flottants, écrire une fonction `Mini(L)` qui calcule le minimum des éléments de `L`.
15. Modifier votre fonction précédente pour qu'elle renvoie non seulement la valeur mais aussi la position du minimum.
16. Soit `D` un dictionnaire dont les clefs et les valeurs sont toutes des chaînes de caractères. Écrire une fonction `PlusLong(D)` qui renvoie la clef dont la valeur associée est de longueur maximale.
17. Écrire des commandes qui permettent de définir une image blanche de taille $h \times w$ (c'est à dire une liste de h listes de longueur w ne comportant que des 255).
18. Soit `texte` une chaîne de caractères, écrire une fonction `EltMajoritaire(texte)` qui renvoie l'un des éléments qui apparait le plus souvent souvent dans cette chaîne.

LES MOINS FACILES MAIS CLASSIQUES

On définit la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par $F_0 = 0$, $F_1 = 1$ et $\forall n \geqslant 2, F_{n+2} = F_{n+1} + F_n$.

19. Écrire une fonction récursive `F(n)` qui renvoie la n-ième valeur de la suite de Fibonacci.
20. Écrire une fonction itérative `F2(n)` qui renvoie la n-ième valeur de la suite de Fibonacci.
21. Quelle est la complexité des fonctions `F` et `F2` écrites ci-dessus ?
22. Proposer une amélioration de `F` en utilisant le principe de mémoïsation. Quelle est alors la complexité ?
23. Écrire une fonction `PlusProches(L)` qui prend en entrée une liste de flottants et renvoie un couple (tuple) d'indices distincts de valeurs les plus proches l'une de l'autre dans cette liste.
24. Écrire une fonction `SecondMax(L)` qui prend en entrée une liste de flottants et renvoie la valeur du second maximum de cette liste (la seconde valeur maximale après le maximum).

LES GRAPHES

Soit `G` un graphe non pondéré, codé par son dictionnaire d'adjacence noté `G`. Les clefs sont les noms des sommets et les valeurs associées la liste des sommets adjacents au sommet.

25. Quelle commande permet de récupérer l'ordre du graphe ?
26. Écrire une fonction `Voisins(G,S)` qui permet de renvoyer la liste des sommets voisins du sommet `S` du graphe `G`.
27. Écrire une fonction `Isole(G)` qui permet de renvoyer la liste des sommets isolés du graphe `G`.

Soit `G` un graphe non orienté et pondéré, dont les sommets sont numérotés de 0 à $n-1$, (n étant donc l'ordre du graphe) codé par sa matrice d'adjacente notée `G`.

28. A quoi correspond la valeur de `G[i][j]` dans le graphe ?
29. Quelle propriété a la matrice `G` du fait que le graphe soit non orienté ?
30. Quelle commande permet de récupérer l'ordre du graphe ?
31. Écrire une fonction `Voisin(G,S)` qui permet de renvoyer la liste des sommets voisins du sommet n°`S` du graphe `G`.
32. Écrire une fonction `EstComplet(G)` qui teste si le graphe `G` est complet.

CORRECTION

LES VARIABLES

(*Compléments sur ces notions aux pages 6 à 9 du formulaire d'informatique sur Cahier de Prépa*)

1. Quelle commande permet d'accéder au dernier élément d'une liste (ou chaine de caractères) `LC` ?
   La commande est `LC[-1]`.
2. Donner 2 différences fondamentales entre les listes et les chaines de caractères, qui fait que l'on pourra préférer d'utiliser l'une plutôt que l'autre dans un script Python.
   Les chaînes de caractères sont immuables (on ne peut pas les modifier en place : `chaine[0]='a'` impossible).
   Les listes subissent l'effet de bord (modifiées même à l'intérieur d'une fonction) et les copies doivent se faire en profondeur (pour éviter de se retrouver avec un simple alias pointant sur la même liste).
   Rappel : pour effectuer une copie profonde d'une liste on fait l'instruction `L2 = L1.copy()` ou `L2 = L1[:]` ou `L2 = [x for x in L1]`.
3. Donner un avantage, puis un inconvénient des dictionnaires sur les listes/chaînes.
   Avantage : la recherche dans un dictionnaire est en $\mathcal{O}(1)$ contre $\mathcal{O}(\texttt{len(L)})$ pour une liste.
   Inconvénient : il n'y a pas d'ordre dans un dictionnaire, donc pas d'indice non plus !
   Rappel : les éléments d'un dictionnaire sont des couples (clés,valeur)
4. Soit `x` un flottant, on veut vérifier si `x` est nul. Le test `x==0` est-il adéquat ?
   Justifier, si vous avez répondu non, quel test feriez-vous ?
   Ce test n'est pas adéquat : un flottant ne sera pas codé informatiquement exactement par le code associé à 0.
   Un meilleur test possible serait : `abs(x) < 10**(-10)` par exemple.
   L'idée est d'exprimer que le flottant `x` est suffisamment petit pour être assimilé à 0, les erreurs d'approximation dues à la représentation des nombres en binaire sont ainsi évitées.

LES ALGORITHMES DE TRI

(*réponses p.23 du formulaire d'informatique sur Cahier de Prépa*)

5. Donner les noms des algorithmes de tri que vous connaissez.
   Le tri fusion, le tri rapide, le tri bulle, le tri par insertion et le tri par sélection.
6. Donner la complexité de ces tris.
   Tri rapide : $\mathcal{O}(n\ln(n))$ en moyenne mais $\mathcal{O}(n^2)$ en pire cas.
   Les autres tris étudiés ont leur complexité pire cas égale à leur complexité moyenne.
   Tri fusion : $\mathcal{O}(n\ln(n))$; Tri bulle, par insertion et par sélection : $\mathcal{O}(n^2)$.
7. Quel est le tri le plus rapide en moyenne ? Quel(s) principe(s) utilise-t-il ?
   Le tri rapide et le tri fusion en $\mathcal{O}(n\ln(n))$ sont les plus rapides en moyenne.
   Ils utilisent le principe de dichotomie (couper l'entrée en deux parties traitées indépendamment), qui est un cas particulier de la méthode diviser pour régner (couper l'entrée en parties traitées indépendamment).
   Ils sont en général écrits de façon récursive.

LE PROGRAMME DE PSI

8. Rappeler les 5 grands thèmes d'informatique au programme de PSI.
   - Bases de données (langage SQL)
   - Dictionnaires (et fonctions de hachage),
   - Programmation dynamique et mémoïsation,
   - Intelligence artificielle : apprentissage (supervisé ou non : K-plus proches voisins, K-moyennes)
   - Théorie des jeux (graphe de jeu, stratégie gagnante, calcul des attracteurs, stratégie MinMax).
9. Rappeler deux exemples classiques étudiés en programmation dynamique.
   Distance de Levenshtein (chaînes de caractères) et algorithme de Floyd-Warshall (distances dans un graphe).
10. Quel différence fondamentale y a-t-il entre les algorithmes « d'intelligence artificielle » KNN et celui des k-moyennes ?
    L'algorithme KNN est un algorithme d'apprentissage supervisé : il nécessite un jeu de données de référence, mais pas celui des k-moyennes (apprentissage non supervisé).
    Pour rappel l'idée de l'algorithme KNN est de repérer les K plus proches voisins (au sens d'une distance à définir selon la nature des données) dans le jeu de données de référence d'une donnée à étiqueter, et de renvoyer l'étiquette majoritaire parmi ces K plus proches voisins.
    L'algorithme des k-moyennes permet de grouper un ensemble de données en k groupes (ou cluster) en calculant le barycentre (selon un mode de calcul à définir) de chaque groupe. Initialement on répartit les éléments dans k groupes au hasard puis on modifie les éléments de groupe en fonction de leur proximité à l'élément barycentrique de chaque groupe. On répète jusqu'à ce que les groupes ne changent plus.

LES ALGORITHMES « FACILES » CLASSIQUES

11. Soit `texte` une chaîne de caractères, écrire une fonction `Presence(lettre,texte)` qui vérifie si `texte` contient le caractère `lettre`.
    Le parcours d'une liste peut se faire élément par élément ou position par position. Ne pas mélanger les deux.

```
def Presence(lettre,texte):
  for car in texte:
    if car == lettre:
      return True
  return False
```

```
def Presence(lettre,texte):
  for i in range(len(texte)):
    if texte[i] == lettre:
      return True
  return False
```

    Attention à l'indentation du `return False`!

12. Soit `L` une liste de flottants, écrire une fonction `moy(L)` qui calcule la moyenne des éléments de `L`.

```
def moy(L):
  S=0
  for x in L:
    S+=x
  return S/len(L)
```

13. Ajouter dans `moy(L)` la signature ainsi qu'un test d'assertion qui vérifie que `L` est bien une liste et qu'elle est non vide.

```
def moy(L:list)->float:
  assert type(L)==list and len(L)!=0
  S=0
  for x in L:
    S+=x
  return S/len(L)
```

14. Soit `L` une liste de flottants, écrire une fonction `Mini(L)` qui calcule le minimum des éléments de `L`.

```
def Mini(L):
  m = L[0]
  for x in L:
    if x<m:
      m=x
  return m
```

15. Modifier votre fonction précédente pour qu'elle renvoie non seulement la valeur mais aussi la position du minimum.

```
def Mini(L):
  m = L[0]
  position = 0
  for i in range(len(L)):
    if L[i]<m:
      m = L[i]
      position = i
  return m, position
```

16. Soit `D` un dictionnaire dont les clefs et les valeurs sont toutes des chaînes de caractères. Écrire une fonction `PlusLong(D)` qui renvoie la clef dont la valeur associée est de longueur maximale.

```
def Pluslong(D):
  M=0
  for key in D:
    if len(D[key])>M:
      M=len(D[key])
      keyM=key
  return keyM
```

17. Écrire des commandes qui permettent de définir une image blanche de taille $h \times w$ (c'est à dire une liste de h listes de longueur w ne comportant que des 255).

    Dans la construction d'une liste de listes, attention à ne pas confondre le nombre de lignes (= nombre de sous-listes = hauteur) et le nombre de colonnes (= taille d'une sous-liste = largeur).

```
img=[[255 for j in range(w)] for i in range(h)]
```

ou

```
img=[]
for i in range(h):
  ligne=[255 for j in range(w)]
  img.append(ligne)
```

ou

```
img=[]
for i in range(h):
  ligne=[]
  for j in range(w):
    ligne.append(255)
  img.append(ligne)
```

18. Soit `texte` une chaîne de caractères, écrire une fonction `EltMajoritaire(texte)` qui renvoie l'un des éléments qui apparait le plus souvent souvent dans cette chaîne.

```
def EltMajoritaire(texte):
  D={}
  for elt in texte:
    if elt in D:
      D[elt]+=1
    else:
      D[elt]=1
  M=0
  for key in dico:
    if D[key]>M:
      M=D[key]
      EltMaj=key
  return EltMaj
```

LES MOINS FACILES MAIS CLASSIQUES

On définit la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par $F_0 = 0$, $F_1 = 1$ et $\forall n \geqslant 2, F_{n+2} = F_{n+1} + F_n$.

19. Écrire une fonction récursive `F(n)` qui renvoie la n-ième valeur de la suite de Fibonacci.

```
def F(n):
  if n<=1:
    return n
  return F(n-1)+F(n-2)
```

20. Écrire une fonction itérative `F2(n)` qui renvoie la n-ième valeur de la suite de Fibonacci.

```
def F2(n):
  if n<=1:
    return n
  f0,f1=0,1
  for k in range(2,n+1):
    f2=f0+f1
    f1=f2
    f0=f1
  return f2
```

Attention à l'ordre dans lequel on décale les variables, et attention à ce que la fonction soit correcte aussi pour les petites valeurs de n.

21. Quelle est la complexité des fonctions `F` et `F2` écrites ci-dessus ?

La complexité de la fonction récursive `F` est exponentielle : le nombre d'appels double quasiment à chaque étape (en fait la vraie complexité est en $\mathcal{O}(\varphi^n)$ ou $\varphi \approx 1,618$ est le nombre d'or). La complexité de la fonction itérative `F2` est linéaire : une seule boucle faite n fois et ne contenant que des instructions simples.

22. Proposer une amélioration de `F` en utilisant le principe de mémoïsation. Quelle est alors la complexité ?

```
memo={}

def F2(n):
  if n<=1:
    memo[n]=n
    return meno[n]
  if n not in memo:
    memo[n]=F2(n-1)+F2(n-2)
  return memo[n]
```

ou

```
memo={0:0,1:1}

def F2(n):
  if n not in memo:
    memo[n]=F2(n-1)+F2(n-2)
  return memo[n]
```

L'important est de comprendre qu'il faut d'abord tester si l'entrée correspond à une clé du dictionnaire de mémoïsation (sortie déjà calculée), sinon on effectue le calcul par récursivité et on n'oublie pas de stocker le résultat dans le dictionnaire avant de renvoyer le résultat !

23. Écrire une fonction `PlusProches(L)` qui prend en entrée une liste de flottants et renvoie un couple (tuple) d'indices distincts de valeurs les plus proches l'une de l'autre dans cette liste.
Il s'agit de l'algorithme du minimum sur l'ensemble des couples d'éléments distincts possibles.

```
def PlusProches(L):
    d=abs(L[0]-L[1])
    ind1,ind2=0,1
    for i in range(len(L)):
        for j in range(i):
            e=abs(L[i]-L[j])
            if e<d:
                d=e
                ind1,ind2=i,j
    return (ind1,ind2)
```

Il est important que la double boucle soit triangulaire stricte (il ne faut pas que j prenne la même valeur que i) sinon on va croire faussement que le minimum de distance est 0 et renvoyer 2 fois la même position.

24. Écrire une fonction `SecondMax(L)` qui prend en entrée une liste de flottants et renvoie la valeur du second maximum de cette liste (la seconde valeur maximale après le maximum).
L'important est de comprendre que l'on recherche le maximum dans un premier temps, puis que l'on applique l'algorithme du maximum en écartant les éléments égaux au maximum (notamment pour initialiser le second maximum `M2`.

```
def SecondMax(T):
    M=Maximum(T)
    debut=0
    while T[debut]==M:
        debut+=1
    M2=T[debut]
    for k in range(debut,len(T)):
        if T[k]>M2 and T[k]!=M:
            M2=T[k]
    return M2
```

LES GRAPHES

Soit `G` un graphe non pondéré, codé par son dictionnaire d'adjacence noté `G`. Les clefs sont les noms des sommets et les valeurs associées la liste des sommets adjacents au sommet.

25. Quelle commande permet de récupérer l'ordre du graphe ?

```
len(G) # la longueur du dictionnaire = nombre de sommets...
```

26. Écrire une fonction `Voisins(G,S)` qui permet de renvoyer la liste des sommets voisins du sommet `S` du graphe `G`.

```
def Voisins(G,S):
  return G[S]
```

27. Écrire une fonction `Isole(G)` qui permet de renvoyer la liste des sommets isolés du graphe `G`.

```
def Isole(G):
  L=[]
  for S in G:
    if len(G[S])==0:
      L.append(S)
  return L
```

Soit `G` un graphe non orienté et pondéré, dont les sommets sont numérotés de 0 à $n-1$, (n étant donc l'ordre du graphe) codé par sa matrice d'adjacente notée `G`.

28. A quoi correspond la valeur de `G[i][j]` dans le graphe ?
    `G[i][j]` *correspond au point de l'arête reliant le sommet n°i au sommet n°j et* 0 *s'il ne sont pas reliés (ou* $+\infty$ *parfois)*
29. Quelle propriété a la matrice `G` du fait que le graphe soit non orienté ?
    La matrice est symétrique.
30. Quelle commande permet de récupérer l'ordre du graphe ?

```
len(G) # la taille de la matrice = nombre de sommets...
```

31. Écrire une fonction `Voisin(G,S)` qui permet de renvoyer la liste des sommets voisins du sommet n°`S` du graphe `G`.

```
def Voisins(G,S):
  V=[]
  for j in range(len(G)):
    if G[S][j]!=0:
      V.append(j)
  return V
```

32. Écrire une fonction `EstComplet(G)` qui teste si le graphe `G` est complet.

```
def EstComplet(G):
  for i in range(len(G)):
    for j in range(len(G)):
      if i==j and G[i][j]!=0:
        return False
      elif i!=j and G[i][j]==0:
        return False
  return True
```