Memento Python

Types

`None`	rien
`int`	entier
`float`	flottant
`bool`	booléen `True` ou `False`
`str`	chaîne de caractères
`(a,b)`	tuple (immuable)
`list[int]`	liste d'entiers, `[1,3,7,9]`
`dict`	dictionnaire
`numpy.uint8`	entier non signé sur 8 bits

Opérateurs

`+ - * /`	arithmétiques
`/`	renvoie un **flottant**
`//`	renvoie un **entier**
`%`	modulo (reste)
`==  !=`	tests d'égalité ou de différence
`<= >= < >`	tests de comparaison
`and, or, not`	et, ou, non logiques
`5 << 3`	renvoie $40 = 5 \times 2^3$
`abs(x)`	valeur absolue de x

Création / Affectation

`a = 3`	création d'un entier
`b = False`	création d'un booléen
`L = []`	liste vide
`D = {}`	dictionnaire vide
`S = ""`	chaîne de caractères vide
`i += 1`	i = i + 1
`j -= 2`	j = j - 2

Structure alternative

Le `else` ou le `elif` **ne sont pas obligatoires**.
Les conditions doivent être mutuellement exclusives.
Le cas par défaut est le chemin d'exécution du `else`.

```
if condition1:
    ... bloc 1
elif condition2:
    ... bloc 2
else:
    ... bloc par défaut
```

Boucles

La fonction `range` prend un **entier** en paramètre !
`range(start, stop, step)` **s'arrête à** `stop-1`.

```
for i in range(n): # pour i de 0 à n-1
    ...  #  ne pas modifier i !
    ...  #  i incrémenté automatiquement

for elem in L:
    # parcourir les éléments d'une liste

i = 0 #déclarer les variables nécessaires
while i < 10: # condition
    i += 1 # incrémenter
```

Listes (muables)

```
L = []
L = [1,2,3,4,5]
L.append(6)
n = len(L)
first = L[0]
last = L[len(L)-1]
last = L[-1]
last = L.pop() # retiré de la liste !
fourth = L[3]
L = [ k for k in range(10) ]
# Liste de listes
L = [ [] for _ in range(50) ]
M = [[0 for _ in range(8)] for _ in range(5)]
M[i][j] # accès à un élément
tranche = L[3:7] # de 3 à 6
L = L1 + L2 # concaténation de liste
```

Dictionnaires (muables)

Les clefs sont nécessairement **immuables** : entiers, chaînes de caractères ou tuples.

```
d = {}     # création d'un dictionnaire vide
d = { "rouge" : 0, "bleu" : 13}
d[k] # accès à la valeur associée à une clé k
d["vert"] = 42 # ajout  clef "vert" -->  42
d = {0 : [1,3], 42 : [7,21,49], 66:[]}
if k not in d:  # recherche d'une clef O(1)
    d[k] = 13 # insertion de la clef et la valeur
```

Chaînes de caractères (immuables)

```
s = "Hello"             # initialisation
ch = s + " Olivier !" # concaténation
s[2]  # accès au troisième caractère
n = len(s)  # longueur de la chaîne
s1 == s2    # test d'égalité de deux chaînes
s[3] < s[2] # comparer deux caractères
s < ch  # comparer deux chaînes de caractères
tranche = s[2:5] # de 2 à 4
```

Numpy

Numpy permet d'utiliser des tableaux de taille fixe, de faire du calcul élément par élément (vectoriel) et du calcul matriciel. Les opérations vectorielles étant compilées, le calcul est rapide.

```
import numpy as np
t = np.array([[1,2],[3,4]]) # tableau d'entiers
t = np.array([[1.,2.],[3.,4.]]) # flottants
t = np.zeros((n,m))
t = np.ones((n,m))
t[2] = 3.45 # affectation d'une valeur
t[i,j] # accès à un élément d'un tableau
t[3:5, :]  # lignes 3 et 4 toutes les colonnes
a = np.array([1,2,3])
b = np.array([7,8,9])
c = (a-5) + 3*b # calcul vectoriel
```

Une liste de liste n'est pas un tableau statique ni un tableau Numpy !

Exemples de fonctions

```
def  vmax(a,b):
    if b >  a:
        return b
    else:
        return a

# récursive
def  pgcd(a,b):
    if b == 0:
        return a
    else:
        return pgcd(b, a%b)
```

Fonctions incontournables

```python
def occurrences(L):
    occ = {}
    for e in L:
        if e in occ:
            occ[e] += 1
        else:
            occ[e] = 1
    return occ

def count_if_sup(L, v):
    c = 0
    for elem in L:
        if elem > v:
            c += 1
    return c

def average(L):
    if len(L) > 0:
        acc = 0
        for elem in L:
            acc +=elem
        return acc/len(L)
    else:
        return None

def max_val(L):
    if len(L) > 0:
        maxi = L[0]
        for elem in L:
            if elem > maxi:
                maxi = elem
        return maxi
    else:
        return None

def max_index(L):
    if len(L) > 0:
        maxi = L[0]
        index = 0
        for i in range(1, len(L)
            if L[i] > maxi:
                maxi = L[i]
                index = i
        return index
    else:
        return None
```

Tri par insertion, générique $O(n^2)/O(n)$

```python
def insertion_sort(t):
    for i in range(1, len(t)):
        to_insert = t[i]
        j = i
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert # j est alors la bonne place
```

Tri par comptage, que sur les entiers $O(n)$

```python
def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:            # décompter
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0
    for v in range(v_max + 1):   # créer le tableau trié
        for j in range(count[v]):
            output[i] = v
            i += 1
    return output
```

Recherche dichomotique dans un tableau trié

```python
def rec_dicho(t, g, d, elem): # Approche récursive
    if g > d:
        return None
    else:
        m = (d + g) // 2 # division entière !
        if t[m] == elem:
            return m
        elif elem < t[m]:
            return rec_dicho(t, g, m-1, elem)
        else:
            return rec_dicho(t, m+1, d, elem)
def dichotomic_search(t, elem): # Approche impérative
    g = 0; d = len(t) - 1
    while g <= d:
        m = (d + g) // 2  # division entière !
        if t[m] == elem:
            return m
        elif t[m] < elem:
            g = m + 1
        else:
            d = m - 1
    return None
```

Tris (diviser pour régner)

Tri fusion, générique $O(nlogn)$

```python
def fusion(t1,t2):
   n1 = len(t1)
   n2 = len(t2)
   if n1 == 0:
       return t2
   elif n2 == 0:
       return t1
   else:
       if t1[0] <= t2[0]:
           return [t1[0]] + fusion(t1[1:], t2)
       else:
           return [t2[0]] + fusion(t1, t2[1:])

def tri_fu(t):
    n = len(t)
    if n < 2:
        return t
    else:
        t1, t2 = t[:n//2], t[n//2:]
        return fusion(tri_fu(t1), tri_fu(t2))
```

Tri rapide, générique
$O(nlogn)$ (meilleur cas) $O(n^2)$ (pire cas)

```python
from random import randrange
def partition(t):
    # pivot aléatoire
    i_pivot = randrange(0, len(t))
    t1, t2 = [], []
    for i in range(len(t)):
        if i == i_pivot:
            pivot = t[i]
        elif t[i] <= t[i_pivot]:
            t1.append(t[i])
        else:
            t2.append(t[i])
    return t1, pivot, t2

def quick(t):
    if len(t) < 2:  # cas de base
        return t
    else:
        t1, pivot, t2 = partition(t)
        return (quick(t1)+[pivot]+quick(t2))
```

Graphes

On représente un graphe par :

1. une liste d'adjacence `adj_lst = [[1,2],[0,3],[0],[1]]`
2. une matrice d'adjacence
   `adj_mat = [[0,1,1,0],[1,0,0,1],[1,0,0,0],[0,1,0,0]]`

Le parcours en largeur utilise une liste d'adjacence, une file d'attente et un tableau `deja_vu`. Il est de complexité $O(n+m)$, on parcours tous les sommets et toutes les arêtes.
Le parcours en largeur peut permettre de :

— lister tous les sommets accessibles depuis un sommet de départ

— de calculer un fonction sur chaque sommet (éventuellement)

— trouver un chemin d'un sommet à un autre (sortie anticipée)

```python
# liste d'adjacence
adj_lst = [[1,2],[0,3],[0],[1]]
# matrice d'adjacence
adj_mat = [[0,1,1,0],[1,0,0,1],[1,0,0,0],[0,1,0,0]]

def parcours_largeur(g, depart):
    file = []
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.append(depart)
    decouverts[depart] = True
    while len(file) > 0:
        u = file.pop(0)   # O(n) pas efficace, cf module queue
        parcours.append(u)
        for x in g[u]:
            if not decouverts[x]:   # O(1)
                decouverts[x] = True
                file.append(x)
    return parcours

def dfs(g, s, decouverts, parcours): # parcours en profondeur
    parcours.append(s)
    decouverts[s] = True
    for u in g[s]:
        if not decouverts[u]:
            dfs(g, u, decouverts, parcours)  # récursif
```

On peut effectuer un parcours en profondeur en utilisant un parcours en largeur et une pile au lieu d'une file.
L'algoritme de Dijsktra est un parcours en largeur qui utilise une file de priorité. Ce dernier ne fonctionne que si les valuations des arêtes du graphe sont positives.

Sac à dos

```python
def glouton_kp(objets, pmax): # Glouton, sans détruire la liste objets.  O(n)
    poids = 0  # poids total du sac
    valeur = 0  # valeur total du sac
    sac = []  # contenu du sac
    objets = sorted(list(objets))  # tri ascendant selon la valeur des objets
    i = 0  # Variable pour parcourir la liste objets
    while i < len(objets) and poids < pmax:
        # choix glouton (la valeur la plus grande)
        v, p = objets[len(objets) - i - 1]
        if poids + p <= pmax:  # cela pourrait-il être une solution partielle ou totale
            sac.append((v, p))  # on l'ajoute
            poids += p
            valeur += v
        i = i + 1
    return sac, poids, valeur

def dyn_kp(objets, pmax): # Programmation dynamique ascendante, O(n.pmax)
    n = len(objets)
    s = [[0 for _ in range(pmax + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for p in range(pmax + 1):
            if i == 0 or p == 0:
                s[i][p] = 0  # pas d'objet pas de solution
            else:
                vi, pi = objets[i - 1]  # on considère le ième objet
                if pi <= p:
                    s[i][p] = max(vi + s[i - 1][p - pi], s[i - 1][p])
                else:
                    s[i][p] = s[i - 1][p]
    return s[n][pmax]

OBJETS = ((100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2))
def mem_kp(n, pmax, S): # Programmation récursive (descendante) et mémoïsation
    if (n, pmax) in S:
        return S[(n, pmax)]  # déjà mémorisé, on s'en sert
    elif n == 0 or pmax == 0:
        return 0  # condition d'arrêt
    else:
        v, p = OBJETS[n - 1]  # on considère le nième objet
        if p > pmax:
            S[(n, pmax)] = mem_kp(n - 1, pmax, S)  # mémoïsation
            return S[(n, pmax)]
        else:
            S[(n, pmax)] = max(v + mem_kp(n - 1, pmax - p, S)
                               , mem_kp(n - 1, pmax, S))
            return S[(n, pmax)]
```

SQL

Opérateurs	Action
`SELECT ... FROM ...`	Projection des colonnes
`SELECT DISTINCT ... FROM ...`	Idem mais sans doublons
`WHERE ...`	Condition de sélection des lignes
`GROUP BY ...`	Regrouper les résultats
`HAVING ...`	Filtrer les regroupements
`ORDER BY ... ASC/DESC`	Ordonner les résultats
`LIMIT n`	Limiter le nombre de résultats
`OFFSET n`	Écarter les n premiers résultats
`UNION, INTERSECT, EXCEPT`	Opérations ensemblistes
`MIN, MAX, AVG, COUNT, SUM`	Fonctions d'agrégation

```sql
SELECT case_id, lat, long
FROM cases
WHERE viscosity > 0.02
ORDER BY case_id;

SELECT COUNT(case_id)
FROM cases
WHERE long < -14.0 and viscosity > 0.0018
LIMIT 5;

SELECT MIN(poids), AVG(poids), MAX(poids)
FROM robots;

SELECT DISTINCT(fab_nom)
FROM fabricant
JOIN fraise ON fabricant.fab_id = fraise.fab_id
WHERE dur > 250
ORDER BY fab_nom;

SELECT robot_id, AVG(viscosity)
FROM cases
JOIN crossings ON cases.case_id = crossings.case_id
GROUP BY robot_id;

SELECT COUNT(fraise.fraise_id), AVG(perte), fab_nom FROM fraise
JOIN fabricant ON fraise.fab_id = fabricant.fab_id
JOIN mesure ON fraise.fraise_id = mesure.fraise_id
GROUP BY fab_nom
HAVING AVG(perte) < 5

SELECT robots.robot_name, chefs.robot_name
FROM robots
JOIN robots as chefs ON robots.chef=chefs.robot_id
```

Complexité temporelle

Pour trouver la complexité temporelle d'une fonction :

1. Trouver le(s) paramètre(s) de la fonction étudiée qui influe(nt) sur la complexité.
2. Déterminer si, une fois ce(s) paramètre(s) fixé(s), il existe un pire ou un meilleur des cas.
3. Calculer la complexité en :
   — calculant éventuellement une somme d'entiers (fonction itérative),
   — posant une formule récurrente sur la complexité (fonction récursive).

Récurrence	Complexité	Algorithmes
$T(n) = 1 + T(n-1)$	$\to O(n)$	factorielle
$T(n) = 1 + T(n/2)$	$\to O(\log n)$	dichotomie, exponentiation rapide
$T(n) = n + 2T(n/2)$	$\to O(n \log n)$	tri fusion, transformée de Fourier rapide

Cas général : toujours justifier la complexité d'un algorithme

Dans l'exemple ci-dessous, si `f` n'est exécutée en temps constant $O(1)$, alors cet algorithme n'est **pas** en $O(n)$.

```python
b = 0
for i in range(n):
    a = f(n)     # ? complexité de f ?
    b = a + b    # opération élémentaire effectuée en temps constant O(1)
```

Opérations sur les listes

Opération	Exemple	Complexité
Création d'une liste vide	`L=[]`	$O(1)$
Accès à un élément	`L[i]`	$O(1)$
Longueur	`len(L)`	$O(1)$
Ajout en fin de liste	`L.append(1)`	$O(1)$
Suppression en fin de liste	`L.pop()`	$O(1)$
Concaténation	`L1+L2`	$O(n_1 + n_2)$
Tranchage (slicing)	`L[n1: n2]`	$O(n_2 - n_1)$
Compréhension	`[f(k) for k in range(n)]`	$O(n)$ si `f(k)` est en $O(1)$
Suppression au début de la liste	`L.pop(0)`	$O(n)$

Opérations sur les dictionnaires

Opération	Exemple	Complexité
Création	`d = \{\}`	$O(1)$
Test d'appartenance d'une clé	`cle in d`	$O(1)$
Ajout d'un couple clé/valeur	`d[cle]= valeur`	$O(1)$
Valeur correspondant à une clé	`d[cle]`	$O(1)$

Complexité des tris

Tri d'un tableau de taille n

Tris	Pire des cas	En moyenne	Meilleur des cas
par insertion	$O(n^2)$	$O(n^2)$	$O(n)$
par comptage	$O(n + v\_max)$	$O(n + v\_max)$	$O(n + v\_max)$
fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
rapide	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Graphe et complexité

Graphe d'ordre n et possédant m arêtes

Algorithme	Pire des cas
Parcours en largeur	$O(n + m)$
Parcours en profondeur	$O(n + m)$
Dijkstra	$O((n + m) \log n)$
Bellmann-Ford	$O(nm)$
Floyd-Warshall	$O(n^3)$

Terminaison

Pour prouver la terminaison d'un algorithme, si cela est possible, il suffit de prouver que les boucles se terminent et donc de :

1. trouver un variant de boucle (entier, positif, strictement décroissant),
2. montrer que le variant est minoré, qu'il franchit nécessairement une valeur limite liée à la condition d'arrêt.

Dans le cas d'un algorithme récursif, on montre que la suite des paramètres appels récursifs est à positive, entière et strictement monotone et que la condition d'arrêt est nécessairement atteinte.
Exemple : $v = |\text{file}| + |\overline{\text{decouverts}}|$ est un variant de boucle pour l'algorithme du parcours en largeur d'un graphe.

Correction

Pour prouver la correction d'un algorithme, on cherche un invariant, une **propriété** liée aux variables qui n'est pas modifiée par les instructions. Dans le cas d'une boucle, on vérifie que l'invariant :

1. est vrai au début de la boucle,
2. est invariant par les instructions de la boucle à chaque itération,
3. donne le résultat escompté si la condition de boucle est invalidée.

Exemple : la correction du parcours en largeur peut se prouver en utilisant l'invariant de boucle : **«Pour chaque sommet v ajouté à** decouverts **et enfilé dans** file**, il existe un chemin de** sdepart **à v.»**

Représentation des nombres

La décomposition d'un **entier** sur une base est **unique**.

$$198_{10} = 1 \times 10^2 + 9 \times 10^1 + 8 \times 10^0 = 100 + 90 + 8$$

$$198_{10} = 11000110_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^2 + 1 \times 2^1 = 128 + 64 + 4 + 2$$

$$198_{16} = C6_{16} = C \times 16^1 + 6 \times 16^0 = 12 \times 16 + 6$$

En base b, on peut représenter b^n nombres avec n chiffres. Par exemple, avec 3 chiffres en base 10, on peut compter de 0 à 999, soit 10^3 nombres.

En base 2, on peut donc représenter 2^n **nombres avec** n **bits**. Pour encoder 256 nombres entiers de 0 à 255, 8 bits suffisent.

Binaire

En **binaire**, en base 2, les chiffres sont 0 ou 1. Un **octet** est composé de 8 bits.
Un octet représente un entier non signé entre 0 et 255 ou signé entre -128 et 127. Un octet peut être représenté par deux chiffres hexadécimaux :

$$11000110_2 = 0xC6 = 12 \times 16^1 + 6 \times 16^0 = 192_{10} + 6_{10} = 198$$

Entier signé

Un nombre entier négatif peut être noté en complément à 2^n.
Par exemple, pour écrire le nombre -67_{10} sur 8 bits en complément à 2^8, on calcule $2^8 - 67 = 189$ qui s'écrit 10111101_2.

Flottants

Un nombre flottant est composé :

— d'un bit de signe s,

— d'un exposant biaisé E,

— et d'une pseudo-mantisse M : $\pm 1, M.2^e$.

C'est pourquoi il est codé en machine par `s E M`.

— En simple précision (32 bits), 6 chiffres significatifs en base 10.

— En double précision (64 bits), 15 chiffres significatifs en base 10.

Pour faire les opérations sur les flottants (addition, multiplication), une mise à sur la même l'échelle des puissances est opérée ce qui peut dégrader la précision du calcul. Ce mécanisme est nommé **mécanisme d'absorption**.