MÉMENTO ITC

Types

`None`	rien
`int`	entier
`float`	flottant
`bool`	booléen `True` ou `False`
`str`	chaîne de caractères
`list[int]`	liste d'entiers, `[1,3,7,9]`
`(a,b)`	tuple (immuable)

Opérateurs

`+ - * / abs`	arithmétiques
`/`	renvoie un flottant
`//`	renvoie un entier
`%`	modulo
`==!=`	égalité ou de différence
`<= >= < >`	comparaison
`and, or, not`	et, ou, non logiques
`1 << 3`	renvoie 8 = 2**3

Affectation ou assignation

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

Structure conditionnelle

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:
    ...
elif condition2:
    ...
else:
    ...
```

Boucles

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

```
i = 0 #déclarer les variables nécessaires
while condition:
    i += 1 # incrémenter les variables si besoin

# pour i de 0 à n-1
for i in range(n):
    ...  #  attention à ne pas modifier i, le for s'en charge
    ...  #  i est incrémenté automatiquement à la fin de l'itération
# pour chaque élément de la liste L

for elem in L:  # syntaxe pour parcourir les éléments d'une liste
    ... #  si pas besoin de i ou de modifier elem
```

Listes (muables)

```
L = []
L = [1,2,3,4,5]
L.append(6)
L = [ k for k in range(10) ]
L = [ [] for _ in range(50) ]
# Liste de listes de 0 de dimension 100 x 200
M = [[0 for _ in range(200)] for _ in range(100)]
M[i][j] # accès à un élément d'une liste de listes
n = len(L)
first = L[0]
last = L[len(L)-1]
last = L[-1]
last = L.pop() # retiré de la liste !
fourth = L[3]
tranche = L[3:7] # de 3 à 7 exclu
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 d'une clé ("vert") de valeur 42
k in d    # test s'il existe une clef k

# exemple : les clefs sont des entiers et les valeurs des listes.
d = {0 : [1,3], 42 : [7,21,49], 66:[]}
k = 1
if k not in d:
    d[k] = [] #  on insère la clef et la valeur
else:
    # la clef existe, la valeur aussi
    d[k].append(3) # : on peut donc modifier la liste
```

Chaînes de caractères (immuables)

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

Numpy

Numpy permet d'utiliser des tableaux statiques (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.]]) # tableau de flottants
t = np.zeros((n,m))
t = np.ones((n,m))
t[2] = 3.45 # affectation d'une valeur dans une case
t[i,j] # accès à un élément d'un tableau
t[3:5, :]  # lignes 3 et 4 sur 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!

Fonctions (exemples)

```python
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 : list[int]) -> dict:
    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
```

Graphes

Le parcours en largeur peut permettre de :

- de calculer un fonction sur chaque sommet
- 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]]

# parcours en largeur (listes : complexité par optimale !)
# utiliser les files pour être efficace (module queue)
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)
        parcours.append(u)
        for x in g[u]:
            if not decouverts[x]:    # O(1)
                decouverts[x] = True
                file.append(x)
    return parcours

import queue    # Module queue
def parcours_largeur(g, depart):
    file = queue.Queue()     # File d'attente
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.put(depart)
    decouverts[depart] = True
    while not file.empty():
        u = file.get()              # O(1)
        parcours.append(u)
        for x in g[u]:
            if not decouverts[x]:
                decouverts[x] = True
                file.put(x)          # O(1)
    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
```

Le parcours en largeur utilise une file d'attente alors que 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.

Tris

```python
# Tri par insertion, générique O(n)   / O(n^2)
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

# Tri fusion, générique O(n log n)
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_fusion(t):
    n = len(t)
    if n < 2:
        return t
    else:
        t1, t2 = t[:n//2], t[n//2:]
        return fusion(tri_fusion(t1), tri_fusion(t2))

# Tri rapide, générique O(n log n) / O(n^2)
from random import randrange
def partition(t):
    i_pivot = randrange(0, len(t))  # random pivot
    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_sort(t):
    if len(t) < 2:  # cas de base
        return t
    else:
        t1, pivot, t2 = partition(t)
        return (quick_sort(t1) + [pivot] + quick_sort(t2))

# Tri par comptage, que sur les entiers  O(n)
def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:  # création de l'histogramme
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0  # indice de parcours du tableau résultat
    for v in range(v_max + 1):
    # Exploitation de l'histogramme
        for j in range(count[v]):
            output[i] = v
            i += 1
    return output
```

Recherche dichotomique

```python
# Impératif
def dichotomic_search(t : list[int], elem : int) -> int :
    g = 0
    d = len(t) - 1
    while g <= d:
        m = (d + g) // 2  # la division entière !
        if t[m] == elem:
            return m
        elif t[m] < elem:
            g = m + 1
        else:
            d = m - 1
    return None

# Récursif
def rec_dicho(t, g, d, elem):
    if g > d:
        return None
    else:
        m = (d + g) // 2
        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)
```

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 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
```

Complexités temporelles

Méthode 1 — Complexité d'une fonction Pour trouver la complexité 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**
$T(n) = 1 + T(n-1)$	$\rightarrow O(n)$	factorielle
$T(n) = 1 + T(n/2)$	$\rightarrow O(\log n)$	dichotomie, exponentiation rapide
$T(n) = n + 2T(n/2)$	$\rightarrow O(n \log n)$	tri fusion, transformée de Fourier rapide

Cas général : toujours justifier la complexité d'un algorithme

Dans l'exemple ci-dessous, si `f` n'est pas 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]`	$O(1)$

Opérations sur les `deque` (files d'attente)

Opération	Exemple	Complexité
Création	`q=deque()`	$O(1)$
Ajout à la fin	`q.append(e)`	$O(1)$
Suppression au début	`e=q.popleft()`	$O(1)$
Longueur	`len(q)`	$O(1)$

Tri d'un tableau de taille n

Tris	Pire des cas	Moyen	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 d'ordre n et possédant m arêtes

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

A 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.

B Correction

Pour prouver la correction d'un algorithme, on cherche un invariant, c'est-à-dire 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 $\mathcal{I}$: **«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**.

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

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

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

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

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) ou double précision (64 bits).
