Table des matières

AIDE PYTHON 2
BASE 2
Affichage 2
Opérateurs 2
Opérateur unpacking * 3
Boucles 3
Affectations 3
Transformations 3
Listes 3
Tuples 3
Strings 4
Dictionnaires 4
Ensembles 4
Tri 5
Mapping 5
Zipping 5
DATACLASSES 5
MATH 5
HEAPQ 6
COLLECTIONS 6
DefaultDict 6
Deque 6
Counter 7
ITERTOOLS 7
Accumulate 7
Combinations 7
Permutations 7
Cycle 8
Repeat 8
FUNCTOOLS 8
Cache 8
Reduce 8
COMPARAISON STRUCTURES DE DONNÉES 8
ALGORITHMES 9
GRAPHES 9
Recherche en largeur (BFS) 10
Recherche en profondeur (DFS) 10
Détection de cycles (via DFS) 11

Composantes connexes 11
Plus court chemin (Dijkstra) 11
Max flow (Push-Relabel) 12
Min cut (BFS) 13
Arbre couvrant minimal (Prim's algorithm 1) 13
Arbre couvrant minimal (Prim's algorithm 2) 14
Arbre couvrant minimal (Boruvka) 14
Cycle Eulérien (vérification) 15
Cycle Eulérien (Fleury's algorithm) 15
Cycle Hamiltonien (Backtracking) 16
Points les plus éloignés 16
Diamètre 17
GÉOMÉTRIQUE 17
Enveloppe convexe (Andrew's algorithm) 17
Intersection de 2 droites 17
Aire d'un polygone non-croisé (Shoelace algorithm) 18
Paire de points la plus proche 18
Problème de la galerie d'art 19
ARITHMÉTIQUE 20
Décomposition en facteur premiers 20
Image d'un polynôme (Horner's method) 20
Produit matriciel 21
Puissance de matrices 21
AUTRES 21
String Suffix array 21
Longest Common Prefix - Kasai 23
PARSING 23
Exemples 23
Exemple texte 23
Exemple graphique 24

AIDE PYTHON

BASE

Affichage

```
print('...', end="")              # écrire ... sans sauter de ligne
print(f"{a} et {b}")              # afficher la valeur de a et b
print(6, 7, 8, 9, 10, sep=";")    # écrire les chiffres séparés par ";"
```

Opérateurs

Arithmétique	Bit à bit	Conditionnel	

`+  # Addition`
`-  # Subtraction`
`*  # Multiplication`
`/  # Division`
`%  # Modulo`
`** # Exponentiation`
`// # Division entière`
	`&  # AND`
	`|  # OR`
	`^  # XOR`
	`~  # NOT`
	`<< # Left shift`
	`>> # Right shift`
		`==  # =`
		`!=  # ≠`
		`>  # >`
		`<  # <`
		`>=  # ≥`
		`<=  # ≤`
			`and  # ∧`
			`or  # ∨`
			`not  # ¬`

Opérateur unpacking *

```
def nb_args(*args):              # Peut prendre autant de paramètres qu'on veut
    return len(args)
nb_args([5, 1], 7, "oui", {})    # return 4
A, *B, C = [0, 1, 2, 3, 4]       # A = 0, B = [1, 2, 3], C = 4
A, *B, C = [0, 1, 2]             # A = 0, B = [1], C = 2
```

Boucles

```
for i in range (1, 6, 2):   # répétition de 1 inclus à 6 exclus, par pas de 2
for element in liste:       # répétition dans la liste (element prenant chaque valeur)
for i, element in enumerate(liste):
# répétition dans la liste (i prenant chaque indice)
while condition:            # répétition selon une condition
continue                    # passer à la boucle suivante
break                       # sortir de la boucle
```

Affectations

```
variable = input("Valeur ?")   # demande une valeur pour la variable (un string)
mot = " ".join(liste)          # Mot prend les valeurs de la liste collée avec un espace qui
sépare chaque valeur
x += 5                         # Équivalant à x = x + 5
```

Transformations

```
round()      # arrondir (entier)
round(x,5)   # arrondir x à 5 chiffres après la virgule
list()       # transformer en liste
len("abc")   # donne la longueur de "abc" qui est 3
```

Listes

```
liste = input("...").split("-")
# Faire entrer dans la liste les éléments de l'utilisateur avec "-" comme délimiteur
liste2 = [i**2 for i in range(10) if i%2]
# renvoie une liste des nombres pairs au carré de 0 à 10 exclus
liste[index]            # renvoie l'élément à l'index indiqué
liste.insert(2, "d")    # mettre "d" à la troisième position de la liste
del liste[i]            # supprime l'élément d'indice i
liste.pop(i)            # autre méthode (par défaut i=-1)
liste.remove("a")       # supprimer la valeur "a" de la liste
liste.index(1)          # position de la valeur 1 dans la liste
liste.count(5)          # renvoie le nombre de 5 dans la liste
copie_liste = liste[:]  # crée une copie de liste
liste[start:end:step]   # renvoie la sous-liste de l commençant à l'indice start (inclus),
finissant à l'indice end (exclus) à un pas de step
min(liste)              # renvoie le minimum de la liste (peut avoir une clé, voir tri)
max(liste)              # renvoie le maximum de la liste (peut avoir une clé, voir tri)
```

Tuples

```
tuplet = (1, 2, 3)   # crée un tuple
tuplet1 = (1,)       # crée un tuple d'une seule valeur (ne pas oublier la virgule)
tuplet.index(1)      # première position de la valeur 1 dans la liste
tuplet[2] = 5
# TypeError: 'tuple' object does not support item assignment
tuplet[start:end:step]
# renvoie la sous-liste de l commençant à l'indice start (inclus), finissant à l'indice end
(exclus) à un pas de step (de la même manière que range(start, stop, end))
```

Strings

```
"bonJour"                  # string contenant bonJour
"bonJour".upper()          # BONJOUR
"bonJour".lower()          # bonjour
"bonJour".swapcase()       # BONjOUR
"bonJour".capitalize()     # Bonjour
"bonJour".center(11, '!')  # !!bonJour!!
"bonJour".ljust(10, '!')   # !!!bonJour
.startswith("oui")         # False
.endswith("our")           # True
.isalnum()                 # uniquement des chiffres ou des lettres ?
.isalpha()                 # uniquement des lettres ?
.isascii()                 # uniquement ascii ?
.isdecimal()               # un nombre décimal ?
.isdigit()                 # uniquement des chiffres ?
.islower()                 # uniquement minuscule ?
.isupper()                 # uniquement majuscule?
.istitle()                 # tous les mots commencent par une majuscule ?
```

Dictionnaires

```
dico = {"cle1": valeur1, "cle2": valeur2}
{}                          # créer un dictionnaire vide (PAS UN ENSEMBLE !)
dico["cle"] = valeur        # ajouter/modifier un élément dans dico
dico.keys()                 # liste les clés du dico
[*dico]                     # liste les clés du dico
dico.values()               # liste des valeurs du dico
dico.pop("cle")             # supprime du dico le couple de clé “cle”
dico.get("cle", défaut)     # donne la valeur associé à "cle" et défaut sinon
dico |= dico2               # fais l'union des deux dictionnaires (si les deux partagent une clé,
prends la valeur de dico2)
cle in dico                 # Vérifie si cle est dans dico
```

Ensembles

```
Ensemble = set()            # Crée un ensemble vide
Ensemble |= {element}       # Ajoute un élément (ou plus) dans l'ensemble
Ensemble -= {element}       # Supprime un élément (ou plus) de l'ensemble
len(Ensemble)               # Donne la taille de l'ensemble
valeur in Ensemble          # Vérifie si valeur est dans Ensemble
# Les opérateurs utiles
A | B    # A.union(B)
A & B    # A.intersection(B)
A - B    # A.difference(B)
A ^ B    # A.symmetric_difference(B) ((A union B) privé de (A inter B))
A < B    # A sous-ensemble strict de B (A ≠ B)
A <= B   # A sous-ensemble de B
```

Tri

```
A = [0, 8, 6, 4, 2, 3, 1]
A.sort()                    # trie la liste A par ordre croissant
A.sort(reverse=True)        # trie la liste A par ordre décroissant
A = sorted([0, 8, 6, 4, 2, 3, 1])
# Renvoie la liste triée par ordre croissant et la stocke dans A
A.sort(key=lambda x: x if x%2 else -x)
# trie la liste A selon son image par une fonction (ne modifie pas les valeurs de A)
# [8, 6, 4, 2, 0, 1, 3]
B = [(1, 2), (-4, 8), (-4, 2)]
B.sort()    # [(-4, 2), (-4, 8), (1, 2)]
# sorted fonctionne comme .sort() mais renvoie une nouvelle liste au lieu de la modifier
```

Mapping

```
A = ["0", "1", "2", "3", "4"]
A = list(map(int, A))                                  # [0, 1, 2, 3, 4]
A = list(map(lambda x: x**2, A))                       # [0, 1, 4, 9, 16]
B = list(map(lambda x, y: x*y, [1, 2, 3], [4, 5, 6]))  # [4, 10, 18] soit 1*4 2*5 3*6
```

Zipping

```
A = [1, 2, 3, 4, 5]
B = ("UN", "DEUX", "TROIS", "QUATRE", "CINQ", "SIX", "SEPT")
C = "123456"
print(list(zip(A, B, C)))   # Fusionne les 3 itérables par indice
# [(1, 'UN', '1'), (2, 'DEUX', '2'), (3, 'TROIS', '3'), (4, 'QUATRE', '4'), (5, 'CINQ', '5')]
```

DATACLASSES

Un @dataclass est un décorateur qui permet de modifier des classes pour définir directement les méthodes __init__ et __repr__ (on peut affecter les attributs directement, et l'afficher via print)

```
from dataclasses import dataclass

@dataclass
class Position:
    x: float
    y: float

print(Position(5, 2))   # Position(5, 2)
```

MATH

```
from math import *

# VARIABLES
inf              # infini
-inf             # -infini
pi, e            # constantes
# FONCTIONS
round(x)         # renvoie l'entier le plus proche de x
ceil(x)          # renvoie l'entier supérieur ou égal à x
floor(x)         # renvoie l'entier inférieur ou égal à x
trunc(x)         # renvoie le prochain entier vers 0 de x
comb(n, k)       # renvoie k parmi n en O(n) (peu importe l'ordre)
factorial(n)     # renvoie n!
fsum(iterable)   # fais la somme des flottants de iterable, sans problème de précision en O(???)
gcd(*numbers)    # renvoie le PGCD des valeurs en entrée en O(???)
lcm(*numbers)    # renvoie le PPCM des valeurs en entrée en O(???)
perm(n, k=None)  # renvoie le nombre de choix de k objets parmi n objets (renvoie n! si k n'est
pas spécifié) (ordre pris en compte)
exp(x)           # calcule e ** x plus précisément
log(x, base=e)   # calcule le log(x) à la base indiquée, ou ln(x) sinon
log2(x)          # calcule log(x, 2) plus précisément
log10(x)         # calcule log(x, 10) plus précisément
```

HEAPQ

Un min-heap est une manière de stocker des valeurs dans un arbre binaire, ce qui facilite l'accès à la valeur minimale.

```
from heapq import heappop, heappush, heapify

minheap = heapify([9, 4, 3, 1, 2, 5])   # O(n), transforme la liste en min-heap
mini = minheap[0]                       # O(1)
heappush(minheap, 7)                    # O(log n), ajoute 7 au min-heap
mini = heappop(minheap)                 # O(log n)
# retire le minimum de minheap, le stocke dans mini et convertit le reste en min-heap
```

(Continue sur la page suivante)

Si on veut faire un max-heap (facilite l'accès à la valeur maximale), il suffit de prendre l'opposé de toutes les valeurs de la liste d'origine, à insérer et retournées.

```
maxheap = heapify(map(lambda x: -x, [9, 4, 3, 1, 2, 5]))   # O(n)
maxi = -maxheap[0]                 # O(1), NE PAS OUBLIER LES -
heappush(maxheap, -7)              # O(log n), pour insérer 7, on insère -7
maxi = -heappop(maxheap)           # O(log n)
```

Attention toutefois, après heapify, notre structure de données est toujours une liste, si on modifie une valeur par une autre fonction que celles de heapq, elle peut perdre sa propriété de heap, ce qui redemande un heapify.

COLLECTIONS

DefaultDict

Un defaultdict est un dictionnaire qui prend une fonction en paramètre et qui fonctionne comme un dictionnaire normal, mais si on veut accéder ou modifier une valeur qui n'existe pas dans le dictionnaire, il fera appel à la fonction pour définir sa valeur avant notre appel.

```
dico = defaultdict(set)
dico["cinq"] = 5            # associe la valeur 5 à la clé "cinq"
dico["set 8"].add(8)        # créer une valeur via la fonction set() pour la clé "set 8", et y
ajoute 8
dico["set 2 4"] |= {2, 4}   # créer une valeur via la fonction set() pour la clé "set 2 4", et y
stocke l'union de soi et de {2, 4}
```

Deque

Les deque (double ended queue) permettent de faire des insertions, des délétions et des accès du début et de la fin d'une liste en O(1), et jusqu'à O(n) pour insertions, délétions et accès autre part. Fonctionne comme une liste sinon.

```
queue = deque("oui")    # initialiser une queue à ["o", "u", "i"] (oui)
queue += ["l"]          # ajouter "l" à la fin (ouil)
queue.appendleft("r")   # ajouter "r" au début (rouil)
queue += "lerb"         # ajoute les éléments ["l", "e", "r", "b"] à la fin (rouillerb)
# cela fonctionne car un string est un itérable, ne pas oublier les [] sinon
queue.extendleft("bra") # ajoute les éléments ["a", "r", "b"] au début (arbouillerb)
queue.rotate(2)         # tourne deux fois par la droite (rbarbouille)
queue.rotate(-1)        # tourne une fois par la gauche (barbouiller)
```

Counter

Complexité: Temps O(n) avec n, Mémoire comme dict
La fonction reduce permet de réduire un itérable à une valeur via une fonction appliquée à chaque élément.

```
c1 = Counter()                          # un Counter vide
c2 = Counter('gallahad')                # un Counter d'un itérable
c3 = Counter({'red': 4, 'blue': 2})     # un Counter d'un dict
c4 = Counter(cats=4, dogs=8)            # un Counter d'arguments mot-clés
c['red']                                # accède au compte de red (0 si absent)
del c['red']                            # retire la valeur
c.total()                               # somme tous les éléments
c.clear()                               # réinitialise le Counter
list(c)                                 # liste les éléments distincts
c.items()                               # liste des (elem, cnt)
c.most_common(5)                        # liste les 5 éléments les plus présents
+c                                      # enlève les valeurs négatives et nulles
```

ITERTOOLS

Accumulate

Applique une fonction (somme par défaut) à l'élément x et le résultat du calcul précédent prev.
Fonctionne comme functools.reduce, mais renvoie un itérable avec chaque étape.

```
list(accumulate([1, 2, 3, 4, 5]))                          # [1, 3, 6, 10, 15]
list(accumulate([1, 2, 3, 4, 5], operator.mul))            # [1, 2, 6, 24, 120]
list(accumulate([1, 3, 2, 6, 0], max))                     # [1, 3, 3, 6, 6]
list(accumulate([1, 3, 5, 7, 9], lambda prev, x: 1+prev*(x-1)))
# [1, 3, 12, 72, 576]
```

Combinations

Renvoie toutes les combinaisons de n valeurs d'un itérable (en gardant l'ordre d'origine)

```
list(combinations("ABCD", 2))    # [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'),
('C', 'D')]
list(combinations(range(4), 3))  # [(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
```

Permutations

Renvoie toutes les combinaisons de n valeurs d'un itérable, (sans doublons)

```
list(permutations("ABC", 2))  # [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'),
('C', 'B')]
list(permutations("ABC"))     # [('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C',
'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]
```

Cycle

Un équivalent de range qui prends un itérable et tourne sur cet itérable

```
for i in cycle([0, 1, 2, 3]):
    print(i)
# 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, ...
```

Repeat

Un équivalent de range qui prend une valeur et qui la répète un certain nombre de fois, ou à l'infini

```
for i in repeat(8, 5):
    print(i)   # 8, 8, 8, 8, 8
for i in repeat(1)
    print(i)   # 1, 1, 1, 1, 1, 1, ...
```

FUNCTOOLS

Cache

Un @lru_cache(maxsize=128) est un décorateur faisant de la mémoïsation, gardant les `maxsize` derniers résultats, ou tous si maxsize=None (équivalent au décorateur @cache).
On peut accéder aux informations de la mémoïsation avec nom_fonction.cache_info().
ATTENTION, il faut que tous les paramètres soient hashables (donc pas de listes par exemple), tout ce qui peut être clé d'un dictionnaire. EXEMPLE PAGE SUIVANTE

```python
from functools import lru_cache # cache

@lru_cache(maxsize=None) # équivalent à @cache
def factorial(n):
    return factorial(n - 1)*n if n else 1
```

Reduce

La fonction reduce permet de réduire un itérable à une valeur via une fonction appliquée à chaque élément.
Fonctionne comme itertools.accumulate, mais ne renvoie que la dernière valeur.

```python
from functools import reduce

def product(liste):
    return reduce(lambda prev, x: prev * x, liste)
produit = product([5, 8, 6, 4, 1, 3]) # 2880
```

COMPARAISON STRUCTURES DE DONNÉES

Opération	List	Deque	Min-Heap	String
Ajouter un élément à la fin	1	1	log n	n
Ajouter un élément au début	n	1	log n	n
Ajouter un élément au milieu	n	n	log n	n
Accéder par index	1	1	Non applicable	1
Retirer un élément à la fin	1	1	log n	n
Retirer un élément au début	n	1	log n	n
Retirer un élément au milieu	n	n	log n	n
Recherche minimum	n	n	1	Non applicable
Recherche maximum	n	n	n	Non applicable

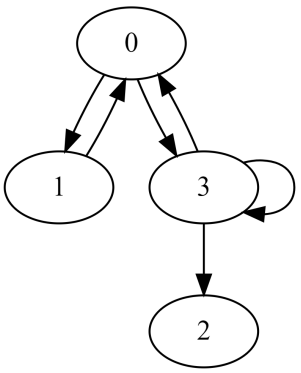
ALGORITHMES

GRAPHES

Définissons quelques diminutifs :

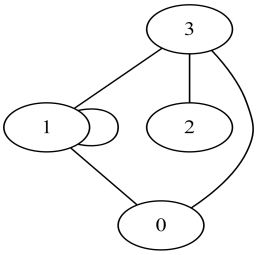
- OG: Oriented graph (dictionnaire d'ensembles de la forme { sommet: {sommets} }) :

```python
OG = {
    0: set([1, 3]),
    1: set([0]),
    2: set(),
    3: set([0, 2, 3])
}
```



- NOG: Non-oriented graph (deux variantes possibles) :
  - Comme un OG, mais chaque arête est mise dans les deux sens

```
NOG = {
    0: set([1, 3]),
    1: set([0, 1, 3]),
    2: set([3]),
    3: set([0, 1, 2])
}
```

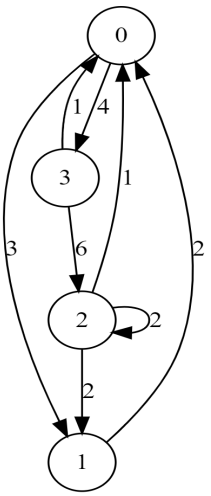


  - Via une matrice d'adjacence (NOGM)

```
NOGM = [
    [0, 1, 0, 1],
    [1, 1, 0, 1],
    [0, 0, 0, 1],
    [1, 1, 1, 0]
]
```

- OWG: Oriented weighted graph (trois variantes possibles) :
  - Un dictionnaire des arêtes avec un OG (deux paramètres séparés dans les fonctions)
  - Au lieu d'un ensemble, on met un dictionnaire avec les poids (on dira OWG

```
OWG = {
    0: {1: 3, 3: 4},
    1: {0: 2},
    2: {0: 1, 1: 2, 2: 2},
    3: {0: 1, 2: 6}
}
```



  - Au lieu d'un dictionnaire, on peut représenter en matrice :
  - On peut mettre +infini (OWGM) ou 0 (OWGM0) pour l'absence d'arête
  - Dépend du cas d'utilisation

```
OWGM0 = [
    [0, 3, 0, 4],
    [2, 0, 0, 0],
    [1, 2, 2, 0],
    [1, 0, 6, 0]
]
```

- NOWG: Non-oriented weighted graph (même variantes que pour NOG)

Recherche en largeur (BFS)

Complexité: O(V)
Note: renvoie le set des sommets atteignable depuis u dans G

```
def bfs(G, u) -> set:
    visited = set([u])
    actual = set(G[u])
    while len(actual) > 0:
        next_to = set()
        for node in actual:
            to_add = set(G[node])
            to_add.difference_update(visited)
            to_add.difference_update(actual)
            next_to.update(to_add)
        visited.update(actual)
```

```
        actual = next_to
    return visite
```

Recherche en profondeur (DFS)

Complexité: O(V)
Note: retourne, sous forme de liste, un chemin pour aller du sommet u à v

```
def dfs(G, prev, u, v, way, isNOG=False) -> tuple[bool, list]:
    for node in G[u]:
        if isNOG and node == prev:
            continue
        if node == v:
            return (True, way + [node])
        if v in way:
            continue
        res = dfs(G, u, node, v, way + [node], isNOG)
        if res[0]:
            return res
    return (False, [])

dfs(NOG, None, start, end, [start], True)      # dans le cas d'un NOG
dfs(OG, None, start, end, [start])             # dans le cas d'un OG
```

Détection de cycles (via DFS)

Complexité: O(V)
Note: fonctionne pour un graphe orienté, mais faire attention au sommet d'origine
Renvoie si le graphe contient un cycle ou non à partir du sommet u fourni

```
def DFS_has_cycle(G, u, prev, visited, isNOG=False) -> bool:
    visited.add(u)
    for v in G[u]:
        if v == prev and isNOG:
            continue
        if v in visited or DFS_has_cycle(G, v, u, visited, isNOG):
            return True
    return False

DFS_has_cycle(NOG, origin, None, set(), True)    # dans le cas d'un NOG
DFS_has_cycle(OG, origin, None, set())           # dans le cas d'un OG
```

Composantes connexes

Complexité: O(V)
Renvoie les composantes connexes d'un graphe non-orienté

```
def connected_components(NOG):
    cc = []
    done = set()
    for u in NOG:
        if u not in done:
            cc_u = bfs(NOG, u)
            done |= cc_u
```

```
            cc += [cc_u]
    return cc
```

Plus court chemin (Dijkstra)

Complexité: O(V²)
Trouve le plus court chemin entre deux sommets d'un graphe orienté non pondéré, ou à pondérations strictement positives.

```
from collections import defaultdict

def Dijkstra(OG, source, target=None, W=defaultdict(lambda: 1)):
    dist = defaultdict(lambda: math.inf)
    prev = defaultdict(lambda: None)
    Q = set(OG)
    dist[source] = 0
    while len(Q) != 0:
        u = min(Q, key=lambda x: dist[x]) # USE HEAP INSTEAD OF SET
        if u == target:
            break
        Q.remove(u)
        for v in OG[u]:
            if v in Q:
                alt = dist[u] + W[(u,v)]
                if alt < dist[v]:
                    dist[v] = alt
                    prev[v] = u
    return dist, prev

def shortest_path(OG, source, target, W=defaultdict(lambda:1)):
    dist, prev = Dijkstra(OG, source, target, W)
    if dist[target] >= math.inf:
        return "IMPOSSIBLE"
    path = []
    u = target
    while u != None:
        path = [u] + path
        u = prev[u]
    return path
```

Max flow (Push-Relabel)

Complexité: Temps - O(V^3) | Mémoire - O(V²)
Pour un graphe pondéré orienté représentant des tuyaux, cet algorithme renvoie le flux maximum de "liquide" en partant de source en allant vers sink.

```
from collections import defaultdict, deque

def max_flow(OWG, source, sink):
    OWG_copy = defaultdict(lambda: defaultdict(int))
    for u in OWG:
        for v, c in OWG[u].items():
            OWG_copy[u][v] = c
```

```
    flow = defaultdict(int)
    height = defaultdict(int)
    height[source] = len(OWG_copy)
    push_queue = deque()
    relabel_queue = deque()

    for neighbor in OWG_copy[source]:
        f = OWG_copy[source][neighbor]
        flow[(source, neighbor)] = f
        flow[(neighbor, source)] = -f
        push_queue += [neighbor]
    while push_queue or relabel_queue:
        if push_queue:
            u = push_queue.popleft()
            for v in OWG_copy[u]:
                if flow[(u, v)] > 0 and height[u] > height[v]:
                    f = min(flow[(u, v)], OWG_copy[u][v] - flow[(v, u)])
                    flow[(u, v)] -= f
                    flow[(v, u)] += f
                    if v != sink and flow[(v, u)] == 0:
                        relabel_queue += [v]
        elif relabel_queue:
            u = relabel_queue.popleft()
            min_height = float('inf')
            for v in OWG_copy[u]:
                if flow[(u, v)] > 0:
                    min_height = min(min_height, height[v])
            height[u] = min_height + 1
    return sum(flow[(source, v)] for v in OWG_copy[source])
```

Min cut (BFS)

Complexité: O(V*E)
Renvoie le nombre minimum de coupes pour séparer les deux nœuds dans un graphe non orienté.

```
from collections import deque

def min_cut(NOG, start, end):
    queue = deque([start])
    visited = set()
    while queue:
        node = queue.popleft()
        visited.add(node)
        for neighbor in NOG[node]:
            if neighbor in visited:
                continue
            queue += [neighbor]
            if neighbor != end:
                continue
            cut_edges = set()
            current = end
            while current != start:
                previous = queue[0]
                cut_edges.add((previous, current))
```

```
                current = previous
                queue.popleft()
            return cut_edges
```

Arbre couvrant minimal (Prim's algorithm 1)

Complexité: Temps : O(E log V) | Mémoire: O(V + E)
Renvoie un graphe représentant l'arbre couvrant minimal (MST), formé à partir d'un sommet.
Un MST est un graphe de poids total minimal reliant tous les sommets du graphe.

```
from typing import List, Tuple, Dict
from heapq import heapify, heappush, heappop

def prim(NOWG: Dict[int, Dict[int, float]], start: int) -> List[Tuple[int, int, float]]:
    min_heap = [(0, start, -1)]
    heapify(min_heap)
    mst = []
    visited = set()
    while min_heap:
        (cost, node, parent) = heappop(min_heap)
        if node in visited:
            continue
        visited.add(node)
        mst += [(parent, node, cost)]
        for neighbor, weight in NOWG[node].items():
            if neighbor not in visited:
                heappush(min_heap, (weight, neighbor, node))
    return mst
```

Arbre couvrant minimal (Prim's algorithm 2)

Complexité: Temps : O(V²) | Mémoire: O(V²)
Version à partir de la matrice d'adjacence avec poids.

```
def Prim(M):
    n = len(M)
    C = {v:inf for v in range(n)}
    E = {v:None for v in range(n)}
    Q = [v for v in range(n)]
    C[0] = 0
    while len(Q) != 0:
        j = 0
        for i in range(len(Q)):
            if C[Q[i]] < C[Q[j]]:
                j = i
        v = Q[j]
        Q.pop(j)
        for w in range(n):
            if w in Q and M[v][w] < C[w]:
                C[w] = M[v][w]
                E[w] = v
    m = 0
    for v in E:
```

```
        if v!=0:
            w = E[v]
            m += M[v][w]
    return E, m
```

Arbre couvrant minimal (Boruvka)

Complexité: Temps : O(E log V) | Mémoire: O(V + E)
A changer, la forme du graphe non orienté est la suivante qui n'est pas la même que dans le reste du document, cette implémentation utilise aussi nog_connected_components qui transforme le nog en og avant d'appeler connected_components

```
def Boruvka(NOG,W):
    V,E = NOG
    F = set()
    completed = False
    while not completed:
        cc = nog_connected_components((V,F))
        cc_idx = {}
        for i in range(len(cc)):
            for u in cc[i]:
                cc_idx[u] = i
        cheapest_edge = {i:None for i in range(len(cc))}
        for (u,v) in E:
            cc_u = cc_idx[u]
            cc_v = cc_idx[v]
            if cc_u != cc_v:
                ce = cheapest_edge[cc_u]
                if ce == None or W[(u,v)] <= W[ce]:
                    cheapest_edge[cc_u] = (u,v)
                ce = cheapest_edge[cc_v]
                if ce == None or W[(u,v)] <= W[ce]:
                    cheapest_edge[cc_v] = (u,v)
        completed = True
        for i in cheapest_edge:
            if cheapest_edge[i] != None:
                completed = False
                F.add(cheapest_edge[i])
    return V,F
```

Cycle Eulérien (vérification)

Complexité: Temps - O(V) | Mémoire - O(1)
Vérifie s'il existe un cycle Eulérien dans le graphe (tous les sommets de degré pair)
Un cycle eulérien est un cycle qui passe une fois et une seule par chaque arête, et revient à l'origine.

```
def has_eulerian_cycle(NOG):
    for vertex in NOG:
        if len(NOG[vertex]) % 2 != 0:
            return False
    return True
```

Cycle Eulérien (Fleury's algorithm)

Complexité: Temps - O(V) | Mémoire - O(E)
Renvoie un cycle Eulérien s'il existe, un tableau vide sinon.

```
from collections import Counter, deque

def eulerian_cycle(NOG):
    if not has_eulerian_cycle(NOG):
        return []
    start, *_ = NOG.keys()
    result = [start]
    queue = deque([start])
    vertex = start
    used_edges = Counter()
    while queue:
        if not NOG[vertex]:
            vertex = queue.pop()
            continue
        next_vertex = NOG[vertex].pop()
        queue += [next_vertex]
        result += [next_vertex]
        vertex = next_vertex
        used_edges[(vertex, next_vertex)] += 1
    return result
```

Cycle Hamiltonien (Backtracking)

Complexité: Temps - O(V!) | Mémoire - O(V)
Un cycle hamiltonien est un cycle qui passe une fois et une seule par chaque sommet, et revient à l'origine.
Ce problème est NP-complet.

```
def hamiltonian_cycle_rec(NOG, path, visited):
    if len(path) == len(NOG):
        return path[0] in NOG[path[-1]]
    for i, v in enumerate(NOG):
        if i in visited or v not in NOG[path[-1]]:
            continue
        path += [v]
        visited.add(i)
        if hamiltonian_cycle_rec(NOG, path, visited):
            return True
        path.pop()
        visited -= {i}
    return False

def hamiltonian_cycle(NOG):
    path = [0]
    if hamiltonian_cycle_rec(NOG,path,{0}):
        return path
    return []
```

Points les plus éloignés

Complexité : Temps - O(V)

Renvoie l'ensemble des points les plus loin du point p fourni, c représente la composante connexe que l'on étudie (None si c'est tout le graphe)
Remarque: Ne marche que sur des NOG

```
def furthest(p, NOG, c=None):
    if c is None:
        c = NOG
    lvl = {p}
    nb_lvl = 0
    visited = {p}
    next_lvl = None
    while len(visited) < len(c):
        nb_lvl += 1
        next_lvl = set()
        for u in lvl:
            for w in NOG[u]:
                if w not in visited:
                    next_lvl.add(w)
                    visited.add(w)
        lvl = next_lvl
    return next_lvl, nb_lvl
```

Diamètre

Complexité : Temps - O(V)
Renvoie le diamètre du Graphe/Composante connexe fourni (c)
Remarque: Ne marche que sur des NOG

```
def diameter(NOG, c = None):
    if c is None:
        c = NOG
    nl, _ = furthest(NOG[0], NOG, c)
    return furthest(nl, NOG, c)[1]
```

GÉOMÉTRIQUE

Type utilisé pour ces algorithmes :

```
from dataclasses import dataclass
@dataclass
class Position:
    x: int
    y: int
```

Enveloppe convexe (Andrew's algorithm)
Complexité: O(n log n)
Requis: Position

```
def Convex_hull(points: List[Position]) -> List[Position]:
    points = sorted(points, key=lambda p: (p.x, p.y))
    if len(points) <= 1:
        return points
    def cross(o: Position, a: Position, b: Position) -> int:
        return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x)
    lower = []
    for p in points:
```

```
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)
    return lower[:-1] + upper[:-1]
```

Intersection de 2 droites

Complexité: O(1)
Renvoie l'intersection des droites (AB) et (CD), ou False si les droites sont parallèles

```
def intersection(A: Position, B: Position, C: Position, D: Position) -> Position | bool:
    a1 = B.y - A.y
    b1 = A.x - B.x
    c1 = a1*(A.x) + b1*(A.y)
    a2 = D.y - C.y
    b2 = C.x - D.x
    c2 = a2*(C.x) + b2*(C.y)
    det = a1*b2 - a2*b1
    if (det == 0): return False
    return Point((b2*c1 - b1*c2) / det, (a1*c2 - a2*c1) / det)
```

Aire d'un polygone non-croisé (Shoelace algorithm)

Complexité: O(n)
Renvoie l'aire du polygone non-croisé.

```
def polygon_area(points: List[Position]) -> float:
    area = 0.0
    for i in range(len(points)):
        A, B = points[i], points[(i + 1) % len(points)]
        area += A.x * B.y - B.x * A.y
    return abs(area) / 2.0
```

Paire de points la plus proche

Complexité: O(n)
Renvoie la paire de point avec la plus petite distance avec la méthode divide and conquer

```
from math import sqrt

def compareX(a, b):
    p1 = a
    p2 = b
    return (p1.x - p2.x)
def compareY(a, b):
    p1 = a
    p2 = b
    return (p1.y - p2.y)
def dist(p1, p2):
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y))
```

```
def bruteForce(P, n):
    min_dist = float("inf")
    if n < 2:
        points = None
    for i in range(n):
        for j in range(i+1, n):
            if dist(P[i], P[j]) < min_dist:
                min_dist = dist(P[i], P[j])
                points = (P[i], P[j])
    return min_dist, points
def min(x, y):
    return x if x < y else y
def stripClosest(strip, size, d, points):
    min_dist = d
    strip = sorted(strip, key=lambda point: point.y)

    for i in range(size):
        for j in range(i+1, size):
            if (strip[j].y - strip[i].y) >= min_dist:
                break
            if dist(strip[i], strip[j]) < min_dist:
                min_dist = dist(strip[i], strip[j])
                points = (strip[i],strip[j])
    return min_dist, points
def closestUtil(P, n):
    if n <= 3:
        return bruteForce(P, n)
    mid = n//2
    midPoint = P[mid]
    dl, pointsl = closestUtil(P, mid)
    dr, pointsr = closestUtil(P[mid:], n - mid)
    points = pointsl if dl < dr else pointsr
    d = min(dl, dr)
    strip = []
    for i in range(n):
        if abs(P[i].x - midPoint.x) < d:
            strip.append(P[i])
    d_s, points_s = stripClosest(strip, len(strip), d, points)
    if d < d_s:
        return d, points
    return d_s, points_s
def closest(P, n):
    P = sorted(P, key=lambda point: point.x)
    return closestUtil(P, n)
```

Problème de la galerie d'art

Complexité: O(n²)
Note: ne fonctionne que pour des polygones simples (pas croisés, sinon problème NP-complet)
Le polygone représente une galerie d'art, et la question est de savoir combien de gardes stationnaires au minimum il faut pour surveiller l'intégralité de la galerie.

```
from collections import deque, Counter
from bisect import bisect_left
```

```python
def triangulate(polygon: List[Position]) -> List[Tuple[Position, Position, Position]]:
    n = len(polygon)
    triangles = []
    for i in range(n):
        j = (i + 1) % n
        k = (i + 2) % n
        triangles += [(polygon[i], polygon[j], polygon[k])]
    return triangles

def min_guards(polygon: List[Position]) -> int:
    triangles = triangulate(polygon)
    diagonals = []
    for t in triangles:
        diagonals + [(t[0], t[2])]
    diagonal_counts = Counter(diagonals)
    graph = {v: set() for v in polygon}
    for d, count in diagonal_counts.items():
        if count == 1:
            graph[d[0]].add(d[1])
            graph[d[1]].add(d[0])
    visited = {v: False for v in polygon}
    guards = []
    queue = deque([polygon[0]])
    visited[polygon[0]] = True
    while queue:
        v = queue.popleft()
        guards += [v]
        for neighbor in graph[v]:
            if not visited[neighbor]:
                visited[neighbor] = True
                pos = bisect_left(queue, neighbor)
                queue.insert(pos, neighbor)
    return guards
```

ARITHMÉTIQUE

Image d'un polynôme (Horner's method)

Complexité: temps O(n) | mémoire O(1)
Note: coeffs est un tuple contenant les exposants du polynôme par ordre de degré (à l'index i se trouve le coefficient de x^i). Renvoie le résultat du polynôme par la variable x

```python
def calc_poly(coeffs: Tuple[float], x: float) -> float:
    result = coeffs[-1]
    for i in range(len(coeffs) - 2, -1, -1):
        result = result * x + coeffs[i]
    return result
```

Produit matriciel

Complexité: Temps - O(nk) | Mémoire - O(nk) avec n, k, dimensions de la matrice de sortie.
Calcule le produit A*B. Attention aux dimensions des matrices entrées.

```python
def multiply_matrix(A, B):
    if (a := len(A)) != (b := len(B[0])):
        print(f"Erreur taille matrice: {a} != {b}")
        return
    return [[sum(a*b for a, b in zip(A_row, B_col)) for B_col in zip(*B)] for A_row in A]
```

Puissance de matrices

Complexité: Temps - O(n² log p) | Mémoire - O(n²) avec n, côté de la matrice d'entrée et p, la puissance
Calcule de manière récursive la puissance p-ième de la matrice entrée.

```python
def matrix_power(M, p):
    if p == 0: return [[1 if i == j else 0 for i in range(len(M))] for j in range(len(M))]
    if p == 1: return M
    res = matrix_power(multiply_matrix(M, M), p//2)
    if p % 2 == 1:
        res = multiply_matrix(res, M)
    return res
```

Décomposition en facteur premiers

Complexité: temps et mémoire O(sqrt(n))
Renvoie un dictionnaire des diviseurs premiers de x et de leurs puissances

```python
from collections import defaultdict

def prime_factors(x: int) -> Dict[int, int]:
    factors = defaultdict(lambda: 0)
    factor = 2
    while x > 1:
        while x % factor == 0:
            factors[factor] += 1
            x = x // factor
        factor += 1 if factor == 2 else 2
    return dict(factors)
```

AUTRES

String Suffix array

Complexité: Temps - O(n) | Mémoire - ~O(n) avec n la taille de la string.
Renvoies une liste des index des suffixes triés par ordre alphabétique

```python
def sais(text: str) -> List[int]:
    chars = sorted(list(set(text)))
    char_to_num = {ch: i + 1 for i, ch in enumerate(chars)}
    arr = [char_to_num[x] for x in text] + [0]
    return _sais(arr)
```

```
def _sais(arr: List[int]) -> List[int]:
    n = len(arr)
    max_el = max(arr)
    if max_el == n - 1:
        ans = [-1] * len(arr)
        for i, el in enumerate(arr):
            ans[el] = i
        return ans
    sl_labels = [-1] * n
    sl_labels[-1] = 0
    for i in reversed(range(n - 1)):
        if arr[i] < arr[i + 1]:
            sl_labels[i] = 0
        elif arr[i] > arr[i + 1]:
            sl_labels[i] = 1
        else:
            sl_labels[i] = sl_labels[i + 1]
    lms_idxs = [i for i in range(1, n) if sl_labels[i] == 0 and sl_labels[i - 1] == 1]
    lms_idxs_pp = [""] * n
    for i in lms_idxs:
        lms_idxs_pp[i] = "*"
    bucket_sizes = [0] * (max_el + 1)
    for el in arr:
        bucket_sizes[el] += 1
    bucket_starts = [0]
    for el in bucket_sizes:
        bucket_starts.append(bucket_starts[-1] + el)
    def _is(sa):
        bucket_ptrs = list(bucket_starts)
        for i in range(n):
            if sa[i] <= 0:
                continue
            j = sa[i]
            if sl_labels[j - 1] == 1:
                sa[bucket_ptrs[arr[j - 1]]] = j - 1
                bucket_ptrs[arr[j - 1]] += 1
        bucket_ptrs = [bucket_starts[i + 1] - 1 for i in range(max_el + 1)]
        for i in reversed(range(n)):
            if sa[i] <= 0:
                continue
            j = sa[i]
            if sl_labels[j - 1] == 0:
                sa[bucket_ptrs[arr[j - 1]]] = j - 1
                bucket_ptrs[arr[j - 1]] -= 1
        return sa
    sa = [-1] * n
    bucket_ptrs = [bucket_starts[i + 1] - 1 for i in range(max_el + 1)]
    for i in reversed(lms_idxs):
        el = arr[i]
        j = bucket_ptrs[el]
        sa[j] = i
        bucket_ptrs[el] -= 1
    sa = _is(sa)
    lms_idxs_set = set(lms_idxs)
```

```
    lms_idxs_ordered = [sa[i] for i in range(n) if sa[i] in lms_idxs_set]
    lms_block_end = {}
    for i in range(len(lms_idxs) - 1):
        lms_block_end[lms_idxs[i]] = lms_idxs[i + 1]
    lms_block_end[n - 1] = n - 1
    lms_idxs_numbered = [-1] * len(lms_idxs_ordered)
    lms_idxs_numbered[0] = 0
    for i in range(1, len(lms_idxs_ordered)):
        j = lms_idxs_ordered[i]
        j_end = lms_block_end[j]
        prev_j = lms_idxs_ordered[i - 1]
        prev_j_end = lms_block_end[prev_j]
        eq = (j_end - j == prev_j_end - prev_j)
        if eq:
            for k in range(j_end - j):
                if arr[j + k] != arr[prev_j + k]:
                    eq = False
        if eq:
            lms_idxs_numbered[i] = lms_idxs_numbered[i - 1]
        else:
            lms_idxs_numbered[i] = lms_idxs_numbered[i - 1] + 1
    lms_idx_to_num = dict(zip(lms_idxs_ordered, lms_idxs_numbered))
    reduced_arr = [lms_idx_to_num[i] for i in lms_idxs]
    reduced_sa = _sais(reduced_arr)
    lms_idxs_sorted = [lms_idxs[i] for i in reduced_sa]
    sa = [-1] * n
    bucket_ptrs = [bucket_starts[i + 1] - 1 for i in range(max_el + 1)]
    for i in reversed(lms_idxs_sorted):
        el = arr[i]
        j = bucket_ptrs[el]
        sa[j] = i
        bucket_ptrs[el] -= 1
    sa = _is(sa)
    return sa
```

Longest Common Prefix - Kasai

Complexité: Temps - O(n) | Mémoire - O(n) avec n la taille de la string.
Renvoies les valeures de Longest Common Prefix selon la string et sa Suffix Array

```
def kasai(s, sa):
    n = len(sa)
    res = [0] * n
    isf = [0] * n
    for i in range(n):
        isf[sa[i]] = i
    k = 0
    for i in range(n):
        if isf[i] == n - 1:
            k = 0
            continue
        j = sa[isf[i] + 1]
        while i + k < n and j + k < n and s[i+k] == s[j+k]:
            k += 1
```

```
        res[isf[i]] = k
        if k > 0:
            k -= 1
    return res
```

PARSING

Exemples

Dans cette sous-partie, on observe différentes manières de parser les données selon le contexte.
En général, la complexité du parsing est O(n) avec n, le nombres de valeurs entrées.

Exemple texte

Certains exercices demandent de récupérer un texte qui peut être sur plusieurs lignes

```
inp = """3
3
Vous savez, je ne crois pas qu'il y ait de bonnes ou de mauvaises situations.
Si je devais résumer ma vie, aujourd'hui avec vous, je dirais que c'est d'abord des rencontres.
Des gens qui m'ont tendu la main, peut-être à un moment où je ne pouvais pas, où j'étais seul
chez moi.
5
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Phasellus ullamcorper auctor aliquam.
Etiam non ex posuere, facilisis risus volutpat, aliquet ante.
Aenean rutrum finibus neque id efficitur.
Duis erat diam, vestibulum ornare hendrerit quis, maximus a mi.
1
Marcello, what are you doing?"""
N = int(input())
paragraphs = []
for _ in range(N):
  P = int(input())
  paragraphs += ["\n".join(input() for _ in range(P))]
print(paragraphs)
```

Input

The input consists of the following lines:

- On the first line, an integer N, describing the number of paragraphs
- then N blocks of lines follow, each describing a paragraph and composed of:
  - on the first line, and integer P, describing the number of lines the paragraph is made of
  - on the next P lines, a paragraph

Exemple graphique

Le format de données pour certains problèmes est beaucoup moins évident, il faut donc réfléchir à l'algorithme pour résoudre le problème avant de parser les données, pour éviter de les interpréter de manière inefficace, incorrecte, ou portant à confusion.
Les graphes orientés pondérés en sont un très bon exemple :

```
inp = """5 6
```

Input

The input comprises several lines, each consisting of integers separated with single spaces:

- The first line consists of the number N of nodes and the number E of edges.
- Each of the following E lines describes an edge with three integers A, B, and W:
  - A and B are the endpoints of the edge (numbered from 0 to $N-1$);

```
0 3 3
1 2 4
2 4 1
4 3 2
0 1 1
0 3 5"""
from collections import defaultdict

N, E = map(int, input().split())
G = defaultdict(dict)
for _ in range(E):
  A, B, W = map(int, input().split())
  G[A][B] = W
  G[B][A] = W  # si on a un NOG
```

D'autres formats page suivante.

Sans contexte, on a plein de méthodes pour le stocker.

Si on cherche le plus court chemin de 0 à N-1, on peut appliquer notre implémentation de Dijkstra, qui utilise un graphe au format orienté, avec un dictionnaire des poids séparés.

```
from collections import defaultdict
N, E = map(int, input().split())
OG = defaultdict(set)
weights = dict()
for _ in range(E):
  A, B, W = map(int, input().split())
  OG[A].add(B)
  weights[(A, B)] = W
Dijkstra(OG, 0, N-1, W)
```

Enfin, si on veut savoir le graphe équivalent en 5 déplacements, il vaut mieux le récupérer sous forme de OWGM0, car il suffit de mettre cette matrice à la puissance 5 pour obtenir ce graphe.

```
OWGM0 = [[0 for _ in range(N)] for _ in range(N)]
for _ in range(E):
  A, B, W = map(int, gtln().split())
  OWGM0[A][B] = W
matrix_power(OWGM0, 5)
```