

Devoir d'informatique — Cliques dans un graphe

Consignes pour la copie

Utilisez exclusivement les feuilles A4 non perforées fournies avec marge et lignes. Numérotez-les. Pour permettre leur numérisation, ne les agrafez pas et ne cornez pas leurs coins. Avant la première feuille, insérez la page avec QR code qui vous a été donnée.

Choisissez le parcours A ou B, indiquez votre choix en haut de la première feuille et suivez ce seul parcours pendant tout le devoir.

Les calculatrices sont interdites. Répondez aux questions dans l'ordre du sujet. Si vous repérez une erreur d'énoncé, signalez-la sur votre copie et poursuivez votre composition.

Le parcours A est orienté vers l'écriture de code. Le parcours B, plus ambitieux, valorise la rédaction de preuves.

Comme il est d'usage, on note x un objet dans un contexte de preuve mathématique et $\mathbf{x}$ le même objet dans un environnement de code. Lorsqu'une hypothèse est faite dans le sujet sur les paramètres d'une fonction, elle n'a pas à être vérifiée dans le code.

Un *graphe non orienté* est un couple $G = (S, A)$, où S est un ensemble fini de sommets et A un ensemble de parties de S ayant un ou deux éléments. Les éléments de A sont appelés *arêtes*. Une arête à deux éléments relie deux sommets distincts. Deux sommets x, y sont dits *adjacents* ou encore *voisins* si et seulement si $\{x, y\} \in A$. Un graphe est *complet* si tous les sommets sont voisins.

1 Introduction

Dans tout le sujet, sauf mention contraire, un graphe $G = (S, A)$ est représenté en Python par un dictionnaire de listes d'adjacence. Les deux graphes suivants serviront d'exemples.

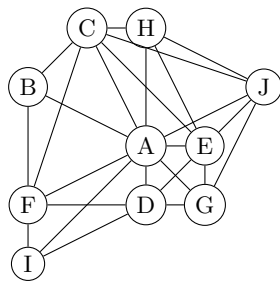


Figure 1

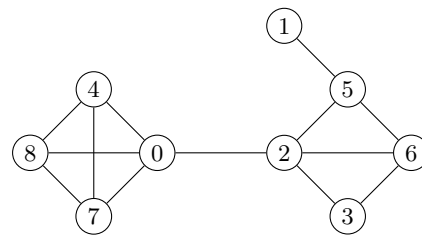


Figure 2

```
1 G_fig1 = {
2     "A": ["B", "C", "D", "E", "F", "G", "H", "I", "J"],
3     "B": ["A", "C", "F", "I"],
4     "C": ["A", "B", "E", "F", "H", "J"],
5     "D": ["A", "E", "F", "G", "I"],
6     "E": ["A", "C", "D", "G", "H", "J"],
7     "F": ["A", "B", "C", "D", "I"],
8     "G": ["A", "D", "E", "J"],
9     "H": ["A", "C", "E", "J"],
10    "I": ["A", "D", "F"],
11    "J": ["A", "C", "E", "G", "H"]
12 }
```

```
1 G_fig2 = {
2     0: [2, 4, 7, 8],
3     1: [5],
4     2: [0, 3, 5, 6],
5     3: [2, 6],
6     4: [0, 7, 8],
7     5: [1, 2, 6],
8     6: [2, 3, 5],
9     7: [0, 4, 8],
10    8: [0, 4, 7]
11 }
```

Pour les annotations de types, on adopte les définitions suivantes (Python 3.11) :

```
1 from typing import TypeAlias, TypeVar
2 Sommet = TypeVar("Sommet")
3 Graphe: TypeAlias = dict[Sommet, list[Sommet]]
```

Ainsi, `G_fig1` est de type `Graphe[str]` et `G_fig2` de type `Graphe[int]`. Toutes les clés d'un graphe sont ses sommets et chaque arête apparaît dans les deux listes d'adjacence.

Question 1. Parcours A

Dans un graphe non orienté :

1. Donner la définition d'une boucle.
2. Écrire

```
1 def voisins(G: Graphe[Sommet], u: Sommet, v: Sommet) -> bool: ...
```

qui indique si u et v sont voisins (on suppose qu'ils sont des sommets de G)

3. Écrire

```
1 def boucles(G: Graphe[Sommet]) -> int: ...
```

qui renvoie le nombre de boucles..

4. Dans un graphe complet à n sommets, quel est le nombre de cliques de cardinal k ?

Solution. une boucle est une arête dont les extrémités sont confondues. Elle est représentée par un singleton.

```
1 def voisins(
2     G: Graphe[Sommet],
3     u: Sommet,
4     v: Sommet
5 ) -> bool:
6     return v in G[u]
7
8 print(voisins(G_fig1, "C", "G"))
9 print(voisins(G_fig1, "H", "A"))
10
11 def boucles(G: Graphe[Sommet]) -> int:
12     nb = 0
13     for v in G:
14         if voisins(G, v, v):
15             nb += 1
16     return nb
```

□

Dans la suite, tous les graphes considérés sont sans boucle.

Une *clique* est un ensemble $C \subseteq S$ dont tous les sommets distincts sont deux à deux adjacents :

$$C \text{ est une clique de } G \iff \forall u, v \in C, \quad u \neq v \implies \{u, v\} \in A.$$

Dans la figure 1, $\{A, B, C, F\}$, $\{A, C, E, H, J\}$, $\{E, D, G\}$ et $\{A, I\}$ sont des cliques.

Question 2. parcours A et B

Pour le graphe de la figure 2, donner sans justification

1. une clique de chacun des cardinaux 1, 2, 3 et 4,
2. puis le nombre de cliques de cardinal 3.

Solution. a) $\{0\}, \{0, 7\}, \{0, 4, 7\}, \{0, 4, 7, 8\}$

b) le nb de cliques de cardinal 3 dans la clique de cardinal 4 est $\binom{4}{3}$ donc 4. On y ajoute les deux 3-cliques contenant 2. Donc 6 3-cliques.

c) donner dans un graphe complet de n sommets, le nombre de k -cliques.

Comme tout sous-ensemble est une clique (puisque tous les sommets sont voisins) ; le nb de cliques est $\binom{n}{k}$ $\square$

Question 3. parcours B

Pour un graphe quelconque $G = (S, A)$ sans boucle, déterminer, en justifiant, le nombre de cliques

1. de cardinal 0 ;
2. de cardinal 1 ;
3. de cardinal 2

Solution. Pour l'ensemble vide et les singletons, on attendait de justifier que ce sont des cliques.

a) l'ensemble vide est une clique en effet

$$\forall x, y, x \in \emptyset \wedge y \in \emptyset \implies \{x, y\} \subset A$$

est vrai car l'hypothèse de l'implication est fausse.

Il y a une clique de cardinal 0.

b) le nb de cliques de cardinal 1 est égal au nombre de singletons soit $|S|$.

Un singleton $\{x\}$ est une clique car

$$\forall x, y, x \in \{x\} \wedge y \in \{x\} \wedge x \neq y \implies \{x, y\} \subset A$$

est vrai car l'hypothèse est fausse.

b) le nb de cliques de cardinal 2 est le nombre d'arêtes soit $|A|$ $\square$

En Python, on représente une clique par une liste de sommets distincts, dans un ordre quelconque.

Question 4. parcours A et B

Écrire `clique(G: Graphe[Sommet], C: list[Sommet]) -> bool` qui indique si C est une clique du graphe. On suppose que les éléments de C sont distincts et appartiennent à G .

Solution. Code

```
1 def clique(G: Graphe[Sommet], C: list[Sommet]) -> bool:
2     for v in C:
3         for u in C:
4             if u==v:
5                 continue
6             if not voisins(G,u,v):
7                 return False
8     return True
```

Double boucle : $O(|C|)$ passages. Mais, si le graphe est complet `voisins(G,u,v)` a une complexité $O(n)$ avec n , nombre de sommets.

On a donc une cpx de $O(n|C|^2)$ $\square$

2 Nombre de cliques

Question 5. parcours A

1. Écrire `sommets(G: Graphe[Sommet]) -> list[Sommet]`, qui renvoie une fois chaque sommet. Pour le graphe de la figure 1, on doit obtenir `["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]`. L'ordre des sommets n'a pas d'importance.
2. Quelle est sa complexité ? Justifier.

Solution. Code

```
1 def sommets(G: Graphe[Sommet]) -> list[Sommet]:
2     return [k for k in G]
```

la cpx de la fonction est $O(n)$ où n est le nombre de sommets donc de clés. Le parcours des n clés se fait en $O(n)$ et on fait n usages de la fonction d'ajout en queue de liste.

□

Soit $n = |S|$. On veut compter les cliques parmi tous les sous-ensembles de S . On associe à chaque entier de $E_1 = \{0, \dots, 2^n - 1\}$ un élément de $E_2 = \mathcal{P}(S)$. On note $\alpha(i, n)$ l'écriture binaire de i sur exactement n bits : $\alpha(372, 10) = 0101110100$.

Question 6. parcours A et B

1. Déterminer $\alpha(678, 12)$ en détaillant le calcul.
2. Écrire `bin(i: int, n: int) -> str`, qui renvoie $\alpha(i, n)$ et déclenche une erreur lorsque $i \notin E_1$.

Solution. on fait des divisions par 2 jusqu'à le quotient soit nul. La suite des restes, inversée, est l'écriture binaire du nombre. on ajoute des zéros à gauche.

00 1010100110

```
1 def bin(i: int, n: int) -> str:
2     """Renvoie l'écriture binaire de i sur exactement n bits."""
3     if n < 0 or not 0 <= i < 2**n:
4         raise ValueError("i doit appartenir à l'intervalle [0, 2**n - 1]")
5
6     resultat = ""
7     for _ in range(n):
8         resultat = str(i % 2) + resultat
9         i //= 2
10
11     return resultat
```

L'étude de la cpx de `bin` se décompose ainsi : d'abord le calcul d'exponentiation en $O(\log_2 n)$ avec comparaison en $O(1)$. Puis les concaténations successives des chaînes de caractères ont un coût proportionnel à la somme des premiers entiers. Bref $O(n^2)$

□

Soit S une liste de tous les sommets, et $t = \alpha(i, n)$. On pose $\varphi(i) = \{S[k] \mid 0 \leq k < n, t[k] = "1"\}$. Pour la liste $S = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]$ correspondant au graphe de la figure 1, on obtient $\varphi(372) = \{B, D, E, F, H\}$.

Question 7. parcours A et B

Déterminer, en justifiant :

1. $\varphi(0)$;
2. $\varphi(2^n - 1)$.

Solution. a) $\varphi(0)$ est une suite de n zéros. Aucun sommet n'est sélectionné. Donc $\emptyset$

b) $\varphi(2^n - 1)$ est une suite de n uns. Tous les sommets sont choisis donc S . □

Question 8. parcours A et B

1. Écrire `phi(S: list[Sommet], i: int) -> list[Sommet]` qui renvoie la liste des sommets de $\varphi(i)$.

2. Écrire `cpt(G: Graphe[Sommet]) -> int` qui compte toutes les cliques de G .

On a `cpt(G_fig1) == 70` et `cpt(G_fig2) == 30`.

Solution. Code

```

1      #q8a
2  def phi(S: list[Sommet], i: int) -> list[Sommet]:
3      b = bin(i, len(S))
4      L = []
5      for i in range(len(S)):
6          if b[i] == "1":
7              L.append(S[i])
8      return L
9
10 S = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
11 phi(S, 372)
12
13 #q8b
14
15 def cpt(G: Graphe[Sommet]) -> int:
16     S = sommets(G)
17     n = len(S)
18     c = 0
19     for i in range(2**n-1):
20         if clique(G, phi(S, i)):
21             c+=1
22     return c
23
24 print(cpt(G_fig1))
25 print(cpt(G_fig2))

```

□

Question 9. parcours A et B

Écrire `phiInv(S: list[Sommet], C: list[Sommet]) -> int`, qui renvoie l'entier i tel que $C = \varphi(i)$. On suppose que S est sans doublon et que $C \subseteq S$.

Solution. Code

```

1  def phiInv(S: list[Sommet], C: list[Sommet]) -> int:
2      dictC = {v: True for v in C} # O(|C|)
3      B = [0] * len(S)
4      for i in range(len(S)):
5          if S[i] in dictC:

```

```

6         B[i] = 1
7     r = 0
8     for i in range(len(S)): #horner
9         r = B[i] + 2 * r
10    return r

```

□

Question 10. parcours A et B

Écrire `Gmax(n: int) -> Graphe[int]`, qui renvoie un graphe à n sommets tel que `cpt(Gmax(n)) == 2**n`.

3 Matrices d'adjacence et cliques de cardinal 3

On suppose maintenant $S = \{0, \dots, n-1\}$. La matrice d'adjacence M est symétrique et vérifie $M_{ij} = 1$ si et seulement si i et j sont voisins. Pour le graphe de la figure 2 :

```

1 M_fig2 = [
2     [0, 0, 1, 0, 1, 0, 0, 1, 1],
3     [0, 0, 0, 0, 0, 0, 1, 0, 0],
4     [1, 0, 0, 1, 0, 1, 1, 0, 0],
5     [0, 0, 1, 0, 0, 0, 1, 0, 0],
6     [1, 0, 0, 0, 0, 0, 0, 1, 1],
7     [0, 1, 1, 0, 0, 0, 1, 0, 0],
8     [0, 0, 1, 1, 0, 1, 0, 0, 0],
9     [1, 0, 0, 0, 1, 0, 0, 0, 1],
10    [1, 0, 0, 0, 1, 0, 0, 1, 0]
11 ]

```

Question 11. parcours A et B

1. Écrire `mat(G: Graphe[sommet]) -> list[list[int]]`, qui renvoie la matrice d'adjacence d'un graphe dont on ne connaît pas le type des sommets (ils peuvent être par exemple de type `str` comme dans la figure 1).
 2. Écrire `adj(M: list[list[int]]) -> Graphe[int]`, qui effectue l'opération inverse.
- On suppose dans cette question que les sommets de G sont $0, \dots, n-1$.

Solution. Code

```

1 #q11a : on ne connaît pas l'ensemble des sommets
2 def mat(G: Graphe[Sommet]) -> list[list[int]]:
3     """Renvoie la matrice d'adjacence du graphe G."""
4     n = len(G)
5     M = [[0 for _ in range(n)] for _ in range(n)]
6     S = sommets(G)
7     numero = {S[i]:i for i in range(len(S))}
8
9     for u in S:
10        for v in G[u]:
11            M[numero[u]][numero[v]] = 1
12
13    return M
14
15 #q11b préciser qu'on veut un graphe à sommet entiers
16
17 def adj(M: list[list[int]]) -> Graphe[int]:
18     n = len(M)
19     d = {i:[j for j in range(n) if M[i][j]] for i in range(n)}
20     return d
21
22 adj(M_fig2) == G_fig2

```

□

Question 12. parcours B

Soit G un graphe non orienté sans boucle donné par sa matrice d'adjacence $M \in \mathcal{M}_n(\{0, 1\})$. On note $M_{i,j}^k$ le coefficient ligne i colonne j de la matrice M^k pour $k \in \mathbb{N}$. Une *marche* est une suite de sommets $x_1, y_1, \dots, x_{r-1}, x_r$ dont tout sommet est voisin du suivant. C'est un chemin dans lequel on peut emprunter plusieurs fois la même arête (dans un sens ou dans l'autre).

1. Montrer que $M_{i,j}^k$ indique le nombre de marches dans G de longueur k entre i et j .
2. On pose $N = M^2$, au sens du produit matriciel usuel. Montrer que G contient une clique de cardinal 3 si et seulement si il existe i, j tel que $M_{i,j}N_{i,j} \neq 0$.
3. Écrire

```

1 def mult(
2     M1: list[list[int]],
3     M2: list[list[int]]
4 ) -> list[list[int]]:
5     ...

```

qui calcule le produit de deux matrices carrées de dimensions compatibles. Par exemple, l'appel `mult(M_fig2, M_fig2)` donne

```

1 [[4, 0, 0, 1, 2, 1, 1, 2, 2],
2  [0, 1, 1, 0, 0, 0, 1, 0, 0],
3  [0, 1, 4, 1, 1, 1, 2, 1, 1],
4  [1, 0, 1, 2, 0, 2, 1, 0, 0],
5  [2, 0, 1, 0, 3, 0, 0, 2, 2],
6  [1, 0, 1, 2, 0, 3, 1, 0, 0],
7  [1, 1, 2, 1, 0, 1, 3, 0, 0],
8  [2, 0, 1, 0, 2, 0, 0, 3, 2],
9  [2, 0, 1, 0, 2, 0, 0, 2, 3]]

```

Solution. a) Montrer que $M_{i,j}^k$ indique le nombre de chemins de longueur k entre i et j .

On suppose n sommets.

Vrai pour $k = 1$ par def de la matrice d'adjacence.

Si $P(k)$. On a $M^{k+1} = M^k \cdot M$

Le coef (i, j) est la somme des $M^k[i][r]M[r][j]$ où r varie de 0 à $n - 1$.

pour r fixé, ce coefficient est non nul ssi il existe une arête de r à i et un nombre non nul de chemins de i à j de longueur k . Par HR, ce nombre est exactement $M^k[i][r]$. Ceci nous indique que $M^k[i][r]M[r][j]$ est exactement le nombre de chemins de i à j de longueur $k + 1$ dont la dernière étape est r .

En sommant sur tous les r possibles, on obtient ainsi exactement le nombre de chemins de longueur $k + 1$ entre i et j

b) Montrer que G contient une clique de cardinal 3 si et seulement si il existe i, j tel que $M_{i,j}N_{i,j} \neq 0$ avec $N = M^2$

Observons que $M_{i,j} \neq 0$ implique $i \neq j$ car il n'y a pas de boucle.

En effet si les deux coefs sont non nuls, ils sont différents. Il existe alors une arête de i à j et un chemin de longueur 2 de i à j . Comme il n'y a pas de boucle, ce chemin de longueur 2 passe par un sommet r intermédiaire. On a donc une arête de i à j , une de i à r et une autre de r à j : cela forme un 3-cycle qui est aussi une 3-clique.

Et si il existe une 3-clique, alors il existe un 3-cycle i, r, j . Il existe alors une arête de i à j , une autre de j à r et une de r à i . Donc un chemin de longueur 2 de i à j , et une arête de i à j . Alors les coefs sont non nuls.

c) Code

```

1
2 def mult(
3     M1: list[list[int]],

```

```

4     M2: list[list[int]]
5 ) -> list[list[int]]:
6     """Renvoie le produit des matrices carrées M1 et M2."""
7     n = len(M1)
8     M = [[0 for _ in range(n)] for _ in range(n)]
9
10    for i in range(n):
11        for j in range(n):
12            for k in range(n):
13                M[i][j] += M1[i][k] * M2[k][j]
14
15    return M

```

□

Question 13. parcours A et B

Écrire `clique3(M: list[list[int]]) -> bool` qui utilise l'équivalence de la question précédente pour indiquer par un booléen si une clique de cardinal 3 est présente dans le graphe représenté par sa matrice d'adjacence M .

Solution. Code

```

1 def clique3(M: list[list[int]]) -> bool:
2     M2 = mult(M,M)
3     n = len(M)
4     for i in range(n):
5         for j in range(n):
6             if M2[i][j] * M[i][j] != 0:
7                 print(i,j)
8                 return True
9     return False

```

□

4 Clique maximale

Soit C_0 une clique de G . Une clique C contenant C_0 est *maximale pour l'inclusion* si aucun sommet extérieur à C ne peut lui être ajouté tout en conservant la propriété de clique :

$$C_0 \subset C \quad C \text{ est une clique} \quad \forall s \in S \setminus C, C \cup \{s\} \text{ n'est pas une clique.}$$

Question 14. parcours A et B

Dans le graphe de la figure 1, donner sans justification deux cliques maximales contenant $\{E, J\}$.

Solution. A, E, G, J et A, E, J, H, C

donc une clique maximale contenant E, J n'est pas nécessairement ue clique maximum.

□

Question 15. parcours A et B

Écrire `cliqueMaxi(G: Graphe[Sommet], C0: list[Sommet]) -> list[Sommet]`. Expliquer brièvement la méthode et chercher à éviter les calculs inutiles. On suppose que C_0 est une clique.

Solution. On gère une pile des sommets extérieurs à C_0 (C_0 est supposé être une clique)

Tant qu'il existe un sommet voisin de tous les sommets de C_0 , ajouter ce sommet à C_0

En sortie, C_0 est une clique et on n'a pas pu ajouter de sommet pour en faire une clique plus grosse.

Ce serait une erreur d'utiliser à chaque tour la fonction `clique` car on ajoute un sommet à une clique. Il ne faut examiner que les relations de voisinage de ce sommet aux sommets déjà présents. Il ne faut pas réévaluer les relations de voisinages entre les sommets déjà présents.

On utilise ici une pile des sommets candidats initialement constituée des sommets de $S \setminus C_0$. A chaque tour, on dépile des sommets jusqu'à en trouver un qui puisse constituer une clique plus grosse. De la sorte chaque sommet candidat n'est examiné qu'une fois.

Dans le corrigé des copies, on pénalise donc :

- a) les appels successifs à `clique` au lieu d'une fonction qui n'examine que le nouveau sommet
- b) les examens multiples d'un même sommet candidat.

```

1 def estVoisinDeTous(G,C,v):
2     for u in C:
3         if not voisins(G,v,u):
4             return False
5     return True
6
7 def cliqueMaxi(G: Graphe[Sommet], C0: list[Sommet]) -> list[Sommet]:
8     #prérequis C0 est une clique de G
9     p = [v for v in G if v not in C0] #pile de sommets disponibles
10    possible = True
11    while p and possible:
12        v = p.pop()
13        possible = estVoisinDeTous(G,C0,v)
14        while not possible:
15            if p:
16                v = p.pop()
17                possible = estVoisinDeTous(G,C0,v)
18            else:
19                break
20        if possible:
21            C0.append(v)
22    return C0

```

□

Question 16. parcours B

Une clique maximale contenant une clique C_0 est-elle une clique de cardinal maximum parmi toutes les cliques ?

Solution. Non

Prendre un graphe contenant un sommet isolé x et dont les autres sommets (au moins 2) forment une clique. La clique maximale contenant $\{x\}$ n'est pas de cardinal maximum. □

5 Graphe de permutation

Dans cette section les sommets d'un graphe G à n sommets sont les nombres $1, 2, \dots, n$. On rappelle qu'une *permutation* est une bijection de $\llbracket 1, n \rrbracket$ dans lui-même.

Pour une permutation σ de $\{1, \dots, n\}$, on définit le graphe G_σ sur ces sommets par

$$\{i, j\} \in A \iff i < j \text{ et } \sigma(i) > \sigma(j).$$

Posons $\sigma_0 : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, n \rrbracket$ telle que $\sigma_0(1) = 2, \sigma_0(2) = 5, \sigma_0(3) = 1, \sigma_0(4) = 4, \sigma_0(5) = 3, \sigma_0(6) = 6$. Le graphe G_{σ_0} associé est celui de la figure 3.

Remarque. FFF il y avait une coquille dans la définition de σ_0 . Elle a été identifiée heureusement par un étudiant et, de toute façon, P_{σ_0} donné lus bas, permettait d'etretrouver l'information.

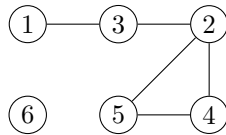


Figure 3 : graphe de permutation associé à σ_0

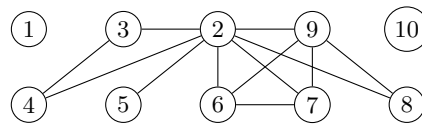


Figure 4 : graphe de la permutation σ_1

Une permutation est représentée par la liste $P_\sigma = [\sigma(1), \dots, \sigma(n)]$. On a donc $P_{\sigma_0} = [2, 5, 1, 4, 3, 6]$.

Question 17. parcours A et B

Donner sans justification la permutation σ_1 dont le graphe est celui de la figure 4.

Solution. on peut représenter une permutation σ par une liste L telle que $L[i] = \sigma(i+1)$
 $[1,9,3,2,4,7,6,8,5,10]$ □

Question 18. parcours A et B

Écrire `GPerm(P:list[int]) -> Graphe[int]`, qui renvoie G_σ pour la permutation représentée par P .

Solution. Code

```

1 #q16
2
3 def GPerm(P: list[int]) -> Graphe[int]:
4     G = {(i+1): [] for i in range(len(P))}
5     for i in range(1, len(P)+1):
6         for j in range(i+1, len(P)+1):
7             if P[i-1] > P[j-1]:
8                 G[i].append(j)
9                 G[j].append(i)
10    return G
11
12 GPerm([2, 5, 1, 4, 3, 6])

```

□

Une *sous-liste* de $P = [x_1, \dots, x_n]$ est une liste $[x_{i_1}, \dots, x_{i_k}]$ avec $i_1 < \dots < i_k$.

Question 19. parcours B

Soit σ une permutation de $\llbracket 1, n \rrbracket$ et C un ensemble d'entiers $\{i_1, i_2, \dots, i_n\}$ tels que $1 \leq i_1 < i_2 < \dots < i_k \leq n$. Montrer que $\{i_1, \dots, i_k\}$ est une clique de G_σ si et seulement si $\sigma(i_1) > \dots > \sigma(i_k)$.

Solution. Preuve du lemme

Par double inclusion

Soit $C = \{i_1, \dots, i_n\}$ un sous-ensemble de sommets de G_σ , avec $1 \leq i_1 < i_2 < \dots < i_n \leq n$.

Si σ est strictement décroissante sur C , alors d'après les règles de constructions des arêtes, il y a une arête de i_1 vers tous ses successeurs, et de façon plus générale, il y a un arc de i_k vers i_{k+r} avec $r > 0$. Ainsi, il y a une arête entre toute paire de sommets de C . Il vient que C est une clique.

Réciproquement, si C forme une clique, il y a un arc de i_k vers i_{k+r} avec $r > 0$. Cela signifie que $\sigma(i_k)$ est supérieur à toutes les images des sommets après i_k . Il vient

$$\sigma(i_1) > \sigma(i_2) > \dots > \sigma(i_n)$$

□

De ce qui précède, on déduit qu'obtenir une clique dans G_σ revient à trouver une sous-liste strictement décroissante de P_σ .

Par exemple, $[5, 4, 3]$ est une sous-liste strictement décroissante de P_{σ_0} et $\{2, 4, 5\}$ est une clique.

Considérons une liste P d'entiers distincts. On construit itérativement sur les positions dans P une liste S valant $[[[]]]$ au départ de l'algorithme.

On cherche au tour i la position k dans S la plus à droite telle que le dernier élément de $S[k]$ soit plus grand que $P[i]$ et on modifie S en conséquence.

A la fin de l'algorithme $S[-1]$ est une plus grande sous-suite décroissante de P .

Pour P_{σ_0} , on obtient par exemple la liste $[[[]], [6], [5, 4], [5, 4, 3]]$.

Question 20. parcours A et B

1. Écrire `makeS(P: list[int]) -> list[list[int]]`, qui construit S en temps $O(n^2)$ pour $n = \text{len}(P)$. Le dernier élément de la liste retournée doit être une sous-suite décroissante de P de taille maximale.
2. Écrire ensuite `cliquePerm(P: list[int]) -> list[int]`, qui renvoie une clique de cardinal maximal du graphe de permutation représenté par P .

Solution. Code

```

1 def makeS(P: list[int]) -> list[list[int]]:
2     S = [[]] # len(P)+1 pour +infini
3     k=0
4     for i in range(0, len(P)):
5         for j in range(0, len(S)):
6             #principe si l'elt courant est plus grand que liste[-1] : on remplace
7             liste=S[j]
8             if liste==[] or liste[-1]>P[i]:#on peut ajouter
9                 k=j
10            if k==len(S)-1:
11                S.append(S[k]+[P[i]])
12            else:
13                #k est le dernier élément tel que S[k][-1]>P[i]
14                # donc S[k+1][-1]<P[i]:
15                S[k+1]=S[k]+[P[i]]
16            #print(S)
17    return S
18
19 def cliquePerm(P: list[int]) -> list[int]:
20    sous_suite = makeS(P)[-1] #ls numéros sont des images par sigma
21    reciproque = {P[i]:i+1 for i in range(len(P))}
22    return [reciproque[k] for k in sous_suite]
```

□

On veut montrer la correction de `makeS`. On convient de noter S_i le contenu de la liste S à la fin du tour i avec la convention que $i = -1$ désigne le temps avant la boucle et $i = n - 1$ (où n désigne la longueur de P) est le dernier tour.

On donne l'invariant $\text{inv}(i)$ en 3 points suivant :

Point a) À la fin du tour i , pour tout $j < |S_i|$, $S_i[j]$ est une sous-suite décroissante de $P[0 : i + 1]^1$ dont la taille est j .

Point b) À la fin du tour i , pour tout $j < |S_i|$, $S_i[j]$ est optimale (c.a.d. que $S_i[j]$ est décroissante à dernier élément maximal parmi toutes les sous-suites décroissantes de $P[0 : i + 1]$ de longueur j).

Point c) À la fin du tour i , si j est la longueur d'une sous-suite décroissante de $P[0 : i + 1]$, alors $j \leq |S_i| - 1$.

Question 21. parcours A et B

1. Montrer que l'invariant est vérifié avant le premier passage dans la boucle sur les positions dans P .
2. Montrer que si l'invariant est vérifié après le dernier passage dans la boucle sur les positions dans P , alors la fonction `makeS` est correcte.

Solution. C'est vérifié avant la boucle car $P[(-1) + 1] = P[0 : 0]$ est vide et $S_{-1}[0]$ est effectivement la liste vide. C'est la meilleure sous-liste à 0 élément et chaque valeur k plus petite que 1 (donc juste, vérifiée que $S_{-1}[k]$ est une sous-liste décroissante de $P[0 : 0]$).

Correction de l'algorithme : Par invariant, à la fin du tour $n - 1$, toute longueur licite ℓ d'une sous-suite de $P[n - 1 + 1] = P[n] = P$, est l'indice d'une sous-suite décroissante de P dans S_n . C'est vrai pour la plus grande longueur possible de sous-suite. Or cette plus grande longueur possible est le cardinal de la clique maximale. $\square$

Question 22. parcours B

On suppose que l'invariant est vérifié à la fin du tour $i < n - 1$. Il y a donc un tour $i + 1$.

1. Montrer que S_{i+1} vérifie le point a) de l'invariant.
2. Montrer que S_{i+1} vérifie le point b) de l'invariant.
3. Montrer que S_{i+1} vérifie le point c) de l'invariant.

Solution. le tour $i = -1$ désigne le temps avant l'entrée dans la boucle ; $i = n$ est le dernier tour de boucle. S_i désigne la liste S au tour i .

A la fin du tour i , a) S_i contient à chaque position une sous-suite de $P[0 : i + 1]$ décroissante strictement et b) le dernier élément de chacune de ces suites est maximum. Enfin c) si ℓ est une longueur licite de sous-suite décroissante de $P[0 : i + 1]$, alors $S_i[\ell]$ est une telle suite.

C'est vérifié avant la boucle car $P[(-1) + 1] = P[0 : 0]$ est vide et $S_{-1}[0]$ est effectivement la liste vide.

Hérédité. On suppose qu'à la fin du tour i , S_i contient à chaque position une sous-suite de $P[0 : i + 1]$ décroissante strictement et avec dernier élément maximum. Et si ℓ est une longueur licite de sous-suite décroissante de $P[0 : i + 1]$, alors $S_i[\ell]$ est une telle suite.

Si $i < n$, alors il y a un tour $i + 1$. On cherche le plus grand indice k tel que le dernier élément de la liste $S_i[k]$ est plus grand que $P[i + 1]$.

Constats : Aucune des sous-suites décroissantes de $P[0 : i + 1]$ (ce que sont les éléments de S_i) ne peut contenir $P[i + 1]$. Si $P[i + 1]$ est dans une sous suite décroissante au tour $i + 1$, il est obligatoirement son dernier élément. Et si une sous-suite décroissante contient $P[i + 1]$, tous ses éléments sauf le dernier forment une sous-suite décroissante de $P[0 : i + 1]$. De plus la différence de longueur entre une plus-grande sous-liste décroissante de $P[0 : i + 1]$ et une plus-grande sous-liste décroissante de $P[0 : i + 2]$ vaut 0 ou 1.

La suite $L = S_i[k] + [P[i + 1]]$ est de longueur $k + 1$ et strictement décroissante.

Si $S_i[k + 1]$ existe, alors $S_{i+1}[k + 1]$ vaut L . Si $S_i[k + 1]$ n'existe pas, $S_{i+1}[k + 1] = L$.

Dans tous les cas, au plus un élément de S_i a été modifié ou un élément a été ajouté. De la sorte, les éléments de S_{i+1} sont encore des suites décroissantes de P dont la taille est donnée par la position dans S_{i+1} . Le point a) est vérifié.

Montrons le point b) en fonction de la position dans S_{i+1} .

Etudions la position $k + 1$ (elle peut être ou non la dernière position de S_{i+1}) :

1. Selon la convention de découpage de Python, $P[a:b]$ désigne la sous-liste de P formée des éléments d'indices a à $b - 1$.

- Supposons $k + 1 \leq |S_i| - 1$. $S_i[k + 1]$ est optimale dans $P[: i + 1]$ et son dernier élément est plus petit que $P[i + 1]$. Donc L est meilleure que toutes les sous-suites de $P[: i + 1]$ de longueur $k + 1$. Et une sous-suite L' de $P[: i + 2]$ meilleure que L contient forcément $P[i + 1]$, sinon ce serait une sous-suite de $P[: i + 1]$ de longueur $k + 1$ et L serait donc meilleure que L' . Mais une sous-suite de $P[: i + 2]$ de longueur $k + 1$ contenant $P[i + 1]$ termine par $P[i + 1]$ comme L . Ainsi $S_{i+1}[k + 1]$ est optimal.
- Supposons $k + 1 = |S_i|$. Dans ce cas il n'existe pas de sous-suite de $P[: i + 1]$ de longueur $k + 1$. En particulier, une sous-suite de $P[: i + 2]$ de longueur $k + 1$ contient $P[i + 1]$. Comme $P[i + 1]$ est obligatoirement en dernière position, toute sous-suite décroissante de $P[: i + 2]$ de longueur $k + 1$ est optimale. C'est le cas de L .

Montrons que le dernier élément est maximum pour une suite de $R = S_{i+1}[k + r]$ avec $r \geq 2$. Le dernier élément de R est plus petit que $P[i + 1]$ puisque k est maximum. Supposons qu'on puisse remplacer ce dernier élément par $P[i + 1]$. Cela signifie que $R[-2] > P[i + 1]$. Mais alors $R[-1]$ est une sous-suite de $P[: i + 1]$ de longueur $k + r - 1 \geq k + 1$. Son dernier élément étant plus grand que $P[i + 1]$, c'est aussi le cas d'une sous-suite optimale de $P[: i + 1]$ de longueur $k + r - 1$. Donc, le dernier élément de $S_i[k + r - 1]$ est plus grand que $P[i + 1]$. Mais c'est impossible par maximalité de k .

Montrons que le dernier élément est maximum pour toutes les sous-suites $S_{i+1}[r]$ avec $r \leq k$. On a $S_{i+1}[r] = S_i[r]$.

- Si pour tout $r \leq k$ le dernier élément de $S_i[r]$ est plus grand que $P[i + 1]$, alors on ne peut pas trouver de sous-suite décroissante de $P[: i + 2]$ optimal qui se termine par $P[i + 1]$. Une sous-suite optimale de $P[: i + 2]$ est forcément une sous-suite optimale de $P[: i + 1]$. Or, $S_i[r]$ est optimale pour $P[: i + 1]$ donc $S_{i+1}[r]$ est optimale pour $P[: i + 2]$.
- Supposons qu'existe $r < k$ tel que le dernier élément de $S_i[r]$ est plus petit que $P[i + 1]$. Et prenons r tel que le dernier élément de $S_i[r + 1]$ soit supérieur à $P[i + 1]$ (un tel r existe puisque $k > r$ vérifie ce point). Alors la suite $S_i[r + 1][: -1]$ est meilleure que $S_i[r]$, ce qui est absurde. Ainsi, il n'existe pas de $r < k$ tel que le dernier élément de $S_i[r]$ est plus petit que $P[i + 1]$.

Montrons le point c)

Soit ℓ une longueur licite pour une sous-suite de $P[: i + 1]$. On a $\ell \leq |S_i|$ car, au mieux, la taille de la plus grande sous-suite décroissante de $P[: i + 2]$ est plus grande d'une unité que la taille de la plus grande sous-suite décroissante de $P[: i + 1]$ (en effet, une sous-suite de $P[: i + 2]$ contenant $P[i + 1]$ le possède en dernier élément).

- Si $\ell < |S_i|$ alors $S_{i+1}[\ell]$ est soit $S_i[\ell]$ (donc une sous-suite décroissante) soit une suite décroissante formée avec $S_i[\ell]$ et $P[i + 1]$. Dans tous les cas, c'est une suite décroissante.
- Si $\ell = |S_i|$ alors $S_{i+1}[\ell]$ est $S_i[\ell - 1]$ à laquelle on ajoute $P[i + 1]$ de sorte que la nouvelle suite soit décroissante.

Bref, à toute position ℓ licite dans S_{i+1} , on trouve une sous-suite décroissante.

Correction de l'algorithme : Par invariant, à la fin du tour $n - 1$, toute longueur licite ℓ d'une sous-suite de $P[: n - 1 + 1] = P[: n] = P$, est l'indice d'une sous-suite décroissante de P dans S_n . C'est vrai pour la plus grande longueur possible de sous-suite. Or cette plus grande longueur possible est le cardinal de la clique maximale. $\square$

FIN DU DEVOIR