PULSARS

CORRECTION INDÉPENDANTE

Centrale-Supélec 2026 — Option informatique (MP)

SESSION	NIVEAU	DURÉE	FILIÈRES
2026	CPGE	—	MP

Partie A · I — Automates et analyse lexicale (Q1 à Q6)

On analyse lexicalement un source OCaml pour en colorer la syntaxe.

- **Q1, Q2** — expression régulière des écritures binaires d'entiers, puis un automate déterministe ; l'adapter à la base 10.
- **Q3** — le langage reconnu par l'automate de la figure 1, et un déterministe équivalent.

> **Q3.** L'automate de la figure 1 a une boucle sur son état initial, lit ensuite `l`, `e`, `t`, puis boucle sur son état final. Il reconnaît donc les mots **contenant le facteur `let`** :
>
> `Σ*·let·Σ*`.
>
> Le déterminiser revient à construire l'automate de recherche de motif, dont l'état mémorise **le plus long préfixe de `let` reconnu en suffixe** de ce qui a été lu :
>
> - `p_0` : `p_0 --l--> p_1`, sinon `p_0` ;
> - `p_1` : `--e--> p_2`, `--l--> p_1`, sinon `p_0` ;
> - `p_2` : `--t--> p_3`, `--l--> p_1`, sinon `p_0` ;
> - `p_3` **acceptant et absorbant**.
>
> *Le retour de `p_2` vers `p_1` sur la lettre `l` est le point à ne pas manquer : après `le`, lire `l` n'annule pas tout, il amorce un nouveau motif.*

- **Q4** — pourquoi un automate déterministe reconnaît l'ensemble des lexèmes.

> **Q4.** Chaque famille de lexèmes est décrite par une expression régulière : les mots clés par eux-mêmes, les noms de variables par `[a-z]+`, les entiers par l'expression de **Q1** transposée en base 10, les symboles par des mots d'une lettre.
>
> Chacun de ces langages est donc **régulier**. Or la classe des langages réguliers est **stable par union finie** : leur réunion est reconnue par un automate non déterministe, que la construction des sous-ensembles rend déterministe.

- **Q5** — un automate pour `in`, `let` et les noms de variables.

> **Q5.** Sur `Γ = {a, …, z}`, les états mémorisent le préfixe lu :
>
> | État | Atteint après | Acceptant, constructeur |
> |---|---|---|
> | `q_0` | rien | non |
> | `q_l` | `l` | oui — **Var "l"** |
> | `q_le` | `le` | oui — **Var "le"** |
> | `q_let` | `let` | oui — **Let** |
> | `q_i` | `i` | oui — **Var "i"** |
> | `q_in` | `in` | oui — **In** |
> | `q_v` | tout autre mot | oui — **Var** |
>
> Les transitions manquantes mènent à `q_v`, lui-même absorbant. Par exemple `δ(q_l, e) = q_le` mais `δ(q_l, a) = q_v`, et `δ(q_let, ·) = q_v` — car `lets` est un nom de variable, pas le mot clé suivi d'un `s`.
>
> *Les états `q_l`, `q_le`, `q_i` sont acceptants : `l`, `le` et `i` sont des noms de variables parfaitement valides. La priorité « mot clé avant variable » est déjà codée par le fait que `q_let` porte `Let` et non `Var`.*

- **Q6** — la complexité de la phase d'analyse lexicale.

> **Q6.** L'automate consomme un caractère par transition, en temps constant. Un caractère est lu au plus deux fois : une fois à l'intérieur du lexème en cours, une fois éventuellement comme caractère ayant provoqué l'échec — auquel cas la lecture reprend à partir de lui.
>
> La complexité est donc **linéaire en la taille du source**, soit `O(n)`.

Q1. Un entier s'écrit soit `0`, soit une suite de chiffres commençant par `1` :

`0 | 1(0|1)*`

L'alternative est nécessaire : sans elle, `0` serait exclu ; et `(0|1)*` seul accepterait les écritures à zéros initiaux comme `007`.

Q2. Quatre états suffisent :

- `q_0` initial ;
- `q_Z`, atteint après un `0` initial — **acceptant** ;
- `q_1`, atteint après un `1` puis n'importe quels chiffres — **acceptant** ;
- `q_p`, état puits.

Les transitions : `δ(q_0, 0) = q_Z`, `δ(q_0, 1) = q_1`, `δ(q_Z, ·) = q_p`, `δ(q_1, ·) = q_1`, `δ(q_p, ·) = q_p`.

L'automate est bien **déterministe et complet**. Pour la base 10, il suffit de changer l'alphabet : `δ(q_0, 0) = q_Z`, `δ(q_0, d) = q_1` pour `d ∈ {1, …, 9}`, et `δ(q_1, d) = q_1` pour tout chiffre. La structure est inchangée.

Partie A · II — Un langage non régulier (Q7, Q8)

`L` est l'ensemble des mots comportant autant de parenthèses ouvrantes que de fermantes.

- **Q7** — `L` n'est pas régulier.

> **Q7.** Raisonnons par l'absurde et appliquons le **lemme de l'étoile**. Si `L` était régulier, il existerait un entier `N` tel que tout mot de `L` de longueur $\geqslant N$ se décompose en `x = uvw` avec $|uv| \leqslant N$, $|v| \geqslant 1$ et `uv^kw ∈ L` pour tout `k`.
>
> Prenons `x = (^N )^N`, qui appartient bien à `L`. La contrainte $|uv| \leqslant N$ impose que `u` et `v` ne contiennent que des parenthèses **ouvrantes** : `v = (^j` avec $j \geqslant 1$.
>
> Alors uv^2w `= (^{N+j} )^N` compte `N + j` ouvrantes pour `N` fermantes : il n'appartient pas à `L`. Contradiction, donc **`L` n'est pas régulier**.
>
> *L'intuition : un automate fini a une mémoire bornée, il ne peut pas compter jusqu'à un entier arbitraire. C'est précisément pour cela que les analyseurs syntaxiques utilisent une pile, là où l'analyseur lexical se contente d'un automate.*

- **Q8** — reconnaître en temps linéaire une liste de tokens bien parenthésée.

> **Q8.** Un simple **compteur** suffit, sans pile ni récursion. On parcourt la liste de tokens en maintenant un entier initialisé à `0` :
>
> - `Lpar` : on incrémente ;
> - `Rpar` : on décrémente ; **si le compteur devient négatif, on rejette** — une parenthèse se ferme alors qu'aucune n'est ouverte ;
> - tout autre token : on ignore.
>
> À la fin, la liste est bien parenthésée si et seulement si le compteur vaut `0`.
>
> Chaque token est examiné une fois, en temps constant : la complexité est **linéaire**.
>
> *Le test de négativité en cours de route est indispensable : sans lui, `) (` passerait pour bien parenthésé, son compteur final valant `0`.*

Partie B · I.1 — Graphes bipartis (Q9 à Q11)

- **Q9** — un graphe est 2-colorable si et seulement s'il est biparti.

> **Q9.** Si `c` est une 2-coloration valide, posons `S_0 = c^{-1}(0)` et `S_1 = c^{-1}(1)`. Ces deux ensembles partitionnent `S`, et toute arête `{u, v}` vérifie `c(u) ≠ c(v)` : ses extrémités sont dans deux parts différentes. Le graphe est donc **biparti**.
>
> Réciproquement, une bipartition `S = S_0 ⊔ S_1` où toute arête joint les deux parts fournit directement la coloration `c(x) = 0` si `x ∈ S_0`, `1` sinon, qui est valide.

- **Q10** — un cycle impair empêche la 2-coloration.

> **Q10.** Soit un cycle `s_0, s_1, …, s_{ℓ-1}, s_0` de longueur **impaire** `ℓ`, et supposons le graphe 2-colorable.
>
> Le long du cycle, deux sommets consécutifs ont des couleurs différentes : la couleur alterne, donc `c(s_i) = c(s_0)` si et seulement si `i` est **pair**.
>
> Or `s_{ℓ-1}` est adjacent à `s_0`, et `ℓ - 1` est pair puisque `ℓ` est impair : on aurait `c(s_{ℓ-1}) = c(s_0)` sur une arête. **Contradiction.**

- **Q11** — réciproquement, l'absence de cycle impair la garantit.

> **Q11.** Supposons `G` sans cycle impair. Traitons chaque composante connexe séparément, en y choisissant une racine `r`, et posons
>
> `c(x) = d(r, x) mod 2`,
>
> où `d` est la distance dans le graphe — bien définie par connexité.
>
> Supposons qu'une arête `{u, v}` vérifie `c(u) = c(v)`. Les distances `d(r,u)` et `d(r,v)` ont alors même parité. En concaténant un plus court chemin de `r` à `u`, l'arête `{u, v}`, puis un plus court chemin de `v` à `r`, on obtient un cycle de longueur
>
> `d(r,u) + 1 + d(r,v)`,
>
> somme de deux entiers de même parité et de `1` : elle est **impaire**. Un tel cycle, même non élémentaire, contient toujours un cycle **élémentaire** de longueur impaire — ce qui contredit l'hypothèse.
>
> Donc toute arête joint deux couleurs différentes : la coloration est valide.

Partie B · I.2 — Une solution naïve (Q12 à Q15)

Une bicoloration est vue comme un entier écrit en base 2 dans un tableau.

- **Q12** — la fonction `suivante`, qui passe à la bicoloration suivante.

> **Q12.** Passer à l'entier suivant, c'est propager une retenue depuis le bit de poids faible : on remet à `0` les `1` initiaux, puis on met à `1` le premier `0` rencontré. S'il n'y en a pas, c'est que la coloration était `1…1` : il n'y a pas de suivante.
>
> ```
> let suivante (c : col) : bool =
>   let n = Array.length c in
>   let i = ref 0 in
>   while !i < n && c.(!i) = 1 do
>     c.(!i) <- 0;
>     incr i
>   done;
>   if !i = n then false
>   else begin c.(!i) <- 1; true end
> ```
>
> Dans le **meilleur cas** — `c.(0) = 0` —, la boucle ne s'exécute pas : le coût est constant, comme demandé. C'est d'ailleurs le cas d'une coloration sur deux.

- **Q13** — la fonction `valide`.

> **Q13.** Il suffit de vérifier chaque arête. Le parcours des listes d'adjacence examine chaque arête deux fois, une par extrémité, ce qui est sans importance.
>
> ```
> let valide (g : graphe) (c : col) : bool =
>   let ok = ref true in
>   for i = 0 to g.nbr - 1 do
>     List.iter (fun j -> if c.(i) = c.(j) then ok := false) g.adj.(i)
>   done;
>   !ok
> ```

- **Q14** — la recherche exhaustive `bicolnaif`.

> **Q14.** On énumère les colorations dans l'ordre croissant de l'entier associé, en s'arrêtant à la première valide.
>
> ```
> let bicolnaif (g : graphe) : bool * col =
>   let c = Array.make g.nbr 0 in
>   let trouve = ref (valide g c) in
>   let reste = ref true in
>   while not !trouve && !reste do
>     reste := suivante c;
>     if !reste then trouve := valide g c
>   done;
>   (!trouve, c)
> ```
>
> Le drapeau `reste` distingue « on a épuisé les `2^n` colorations » de « on en a trouvé une valide ».

- **Q15** — sa complexité dans le pire cas.

> **Q15.** Dans le pire cas — le graphe n'est pas bicolorable — les `2^n` colorations sont toutes engendrées et toutes testées.
>
> Chaque appel à `valide` coûte `O(n + |A|)` : il parcourt les `n` listes d'adjacence, dont la somme des longueurs vaut `2|A|`. La complexité totale est
>
> `O(2^n·(n + |A|))`,
>
> **exponentielle**. C'est ce que la partie I.4 corrigera en descendant à `O(n + |A|)`.

Partie B · I.3 — Une file avec deux piles (Q16 à Q21)

- **Q16** — la liste OCaml comme structure de pile.

> **Q16.** La liste OCaml offre exactement les deux opérations d'une pile en temps **constant** : `x :: l` empile, et le filtrage `x :: reste` dépile en lisant la tête. Aucune recopie n'a lieu, la queue étant partagée entre l'ancienne et la nouvelle liste.
>
> Elle est de plus **persistante** : empiler ne détruit pas la pile précédente, ce qui convient au style fonctionnel demandé.

- **Q17** — les quatre opérations de la file : `initFile`, `estVideFile`, `enfile`, `defile`.
- **Q18, Q19** — l'ordre de sortie, et le coût d'un `defile` dans le pire cas.
- **Q20, Q21** — le coût **amorti**, par un argument de comptage.

> **Q17.**
>
> ```
> let initFile () : 'a file = { entree = [] ; sortie = [] }
>
> let estVideFile (f : 'a file) : bool = f.entree = [] && f.sortie = []
>
> let enfile ((x, f) : 'a * 'a file) : 'a file =
>   { f with entree = x :: f.entree }
>
> let defile (f : 'a file) : 'a * 'a file =
>   match f.sortie with
>   | x :: reste -> (x, { f with sortie = reste })
>   | [] ->
>       match List.rev f.entree with
>       | [] -> failwith "file vide"
>       | x :: reste -> (x, { entree = [] ; sortie = reste })
> ```
>
> Le **basculement** n'a lieu que lorsque la pile de sortie est vide : on renverse alors toute la pile d'entrée d'un coup, ce qui remet les éléments dans l'ordre d'arrivée.

Q18. La file logique est la liste `sortie @ List.rev entree`.

`enfile` ajoute `x` en tête de `entree`, donc **en queue** de cette liste ; `defile` prend la tête de `sortie`, donc la **tête** de cette liste. Le basculement, lui, ne fait que réécrire `[] @ List.rev entree` en `List.rev entree @ []` : la liste logique est inchangée.

L'ordre de sortie est donc exactement l'ordre d'entrée : la structure est bien une **file**.

Q19. Dans le pire cas, `sortie` est vide et `entree` contient les `n` éléments : `List.rev` les parcourt tous. Le coût est `O(n)`.

Q20. Suivons un élément particulier. Il est empilé une fois sur `entree`. S'il finit par sortir, il est dépilé de `entree`, empilé sur `sortie`, puis dépilé de `sortie`.

Il subit donc **au moins 1 et au plus 4** opérations élémentaires — et jamais davantage : une fois passé sur `sortie`, il n'y revient pas.

Q21. Sur `n` opérations `enfile` ou `defile` à partir d'une file vide, au plus `n` éléments entrent, donc le nombre total d'opérations élémentaires est compris entre `0` et `4n`.

Le coût **amorti** par opération est donc **constant** : bien qu'un `defile` isolé puisse coûter `O(n)`, une suite de `n` opérations coûte `O(n)` au total.

C'est l'intérêt de cette implémentation : le coût du basculement est « payé d'avance » par les empilements qui l'ont rendu nécessaire.

Partie B · I.4 — Bicoloration par parcours (Q22 à Q27)

- **Q22** — la fonction `bicol`, par parcours en largeur.
- **Q23 à Q25** — terminaison, complexité, correction.

> **Q22.** On colore la racine, puis chaque sommet découvert reçoit la couleur opposée à celle de son prédécesseur. Une arête reliant deux sommets déjà coloriés de la même couleur signe l'échec.

```
let bicol (g : graphe) : bool * col =
  let c = Array.make g.nbr (-1) in
  let f = ref (enfile (0, initFile ())) in
  let ok = ref true in
  c.(0) <- 0;
  while not (estVideFile !f) do
    let (s, reste) = defile !f in
    f := reste;
    List.iter
      (fun v ->
        if c.(v) = -1 then begin
          c.(v) <- 1 - c.(s);
          f := enfile (v, !f)
        end
        else if c.(v) = c.(s) then ok := false)
      g.adj.(s)
  done;
  (!ok, c)
```

- **Q26** — l'adapter aux graphes non connexes.

> **Q26.** Il suffit d'englober le parcours dans une boucle sur tous les sommets :
>
> pour `s` de `0` à `nbr - 1`, si `c.(s) = -1`, colorier `s` en `0` et lancer le parcours depuis `s`.
>
> Chaque itération traite une **composante connexe**, et le graphe est bicolorable si et seulement si chacune l'est — les composantes n'ayant aucune arête entre elles. La complexité reste `O(n + |A|)`.

- **Q27** — la variante en profondeur.

> **Q27.** Il suffit de **remplacer la file par une pile**, le reste du code étant identique : la règle de coloration `c.(v) <- 1 - c.(s)` ne dépend pas de l'ordre de visite.
>
> On peut aussi l'écrire récursivement : une fonction `visite s` qui, pour chaque voisin non colorié, le colorie de la couleur opposée et s'appelle sur lui.
>
> *Le choix du parcours est indifférent ici, ce qui n'est pas le cas de tous les algorithmes de graphe : la propriété exploitée — deux sommets adjacents ont des parités de distance différentes — vaut quel que soit l'arbre couvrant.*

Q23. Un sommet n'est enfilé qu'au moment où sa couleur passe de `-1` à `0` ou `1`, ce qui ne se produit **qu'une fois** pour chaque sommet. Il y a donc au plus `n` enfilements, donc au plus `n` défilements : la boucle s'arrête.

Q24. Chaque sommet est défilé une fois, et sa liste d'adjacence parcourue une fois : la somme de ces parcours vaut `2|A|`. Les opérations de file coûtant `O(1)` en amorti (**Q21**), la complexité est

`O(n + |A|)`,

linéaire en la taille du graphe — à comparer aux `O(2^n)` de **Q15**.

Q25. *Invariant* : à tout instant, un sommet colorié a reçu la parité de sa distance à la racine, puisque chaque découverte inverse la couleur du prédécesseur.

Si l'algorithme renvoie `true`, aucune arête ne joint deux sommets de même couleur : la coloration est **valide** par définition.

S'il renvoie `false`, c'est qu'une arête relie deux sommets de même parité de distance. Le raisonnement de **Q11** construit alors un cycle impair, et **Q10** montre qu'aucune 2-coloration n'existe : la réponse est correcte.

Partie B · II — La 3-coloration par retour sur trace (Q28 à Q30)

- **Q28** — la validation partielle `partial`.

> **Q28.** Il suffit de comparer la couleur de `s` à celle de chacun de ses voisins.
>
> ```
> let partial (g : graphe) (c : col) (s : int) : bool =
>   List.for_all (fun v -> c.(v) <> c.(s)) g.adj.(s)
> ```
>
> Les sommets non encore colorés portent `-1`, valeur que `c.(s)` ne prend pas puisque `s` est supposé coloré : le test les accepte automatiquement, ce qui est bien le comportement voulu.

- **Q29** — la fonction `backtracking`.

> **Q29.** On colorie les sommets dans l'ordre, en essayant les trois couleurs et en **revenant en arrière** dès qu'aucune ne convient.
>
> ```
> let backtracking (g : graphe) : bool * col =
>   let c = Array.make g.nbr (-1) in
>   let rec aux s =
>     if s = g.nbr then true
>     else begin
>       let trouve = ref false in
>       let couleur = ref 0 in
>       while not !trouve && !couleur < 3 do
>         c.(s) <- !couleur;
>         if partial g c s then trouve := aux (s + 1);
>         incr couleur
>       done;
>       if not !trouve then c.(s) <- -1;
>       !trouve
>     end
>   in
>   let r = aux 0 in
>   (r, c)
> ```
>
> La remise à `-1` en cas d'échec est essentielle : sans elle, un sommet abandonné garderait une couleur parasite qui fausserait les tests ultérieurs.

- **Q30** — sa complexité dans le pire cas.

> **Q30.** Dans le pire cas — aucune 3-coloration n'existe — l'arbre de recherche est exploré entièrement : il compte au plus `3^n` feuilles, une par affectation possible.
>
> Chaque nœud demande un appel à `partial`, de coût `O(deg(s))`, donc `O(n)` au plus. La complexité est
>
> `O(3^n·n)`,
>
> **exponentielle**. C'est attendu : la 3-coloration est un problème NP-complet, et aucun algorithme polynomial n'est connu. L'élagage par `partial` réduit l'arbre en pratique, sans changer cette borne.

Partie B · III.1 — La formule d'Euler (Q31 à Q36)

- **Q31** — la somme des degrés des sommets et celle des degrés des faces.
- **Q32, Q33** — la formule d'Euler, d'abord pour les arbres, puis en général.
- **Q34, Q35** — la majoration $|A| \leqslant 3|S| - 6$, la non-planarité de K_5, et l'existence d'un sommet de petit degré.

> **Q31.** Chaque arête possède **deux extrémités**, donc contribue 1 au degré de chacune :
>
> $\Sigma_{s \in S} deg(s) = 2|A|$.
>
> Chaque arête borde par ailleurs **deux faces** — ou la même face deux fois, si elle est un isthme, auquel cas elle est parcourue deux fois par le même parcours de frontière. Dans les deux cas elle compte pour 2 :
>
> $\Sigma_{f \in F} deg(f) = 2|A|$.

- **Q36** — le cas biparti et $K_{3,3}$.

> **Q36.** Un graphe biparti n'a **aucun cycle impair** (**Q10**), donc aucune face n'est bordée par exactement 3 arêtes : $deg(f) \geqslant 4$. Le même calcul donne
>
> $2|A| \geqslant 4|F|$, puis $2 - |S| + |A| \leqslant |A|/2$, soit $|A| \leqslant 2|S| - 4$.
>
> Pour $K_{3,3}$: $|S| = 6$ et $|A| = 9$, alors que $2\times6 - 4 = 8$. Donc **$K_{3,3}$ n'est pas planaire**.
>
> *Ces deux graphes sont exactement les obstructions du théorème de Kuratowski : un graphe est planaire si et seulement s'il n'en contient aucun.*

Q32. Un arbre est connexe et sans cycle : il vérifie $|A| = |S| - 1$, ce qui se montre par récurrence en retirant une feuille — tout arbre fini à au moins deux sommets en possède une.

Dessiné dans le plan, il ne délimite **aucune région bornée** : il n'a qu'une seule face, la face infinie. Donc

$|S| - |A| + |F| = |S| - (|S| - 1) + 1 = 2$.

Q33. Récurrence sur le nombre d'arêtes, à $|S|$ fixé.

Si G est sans cycle, c'est un arbre : **Q32** conclut.

Sinon, G contient un cycle ; choisissons-y une arête e. Le graphe $G - e$ reste **connexe** — un cycle offre un chemin de contournement — et reste planaire.

Le retrait de e fait **fusionner les deux faces** qu'elle séparait, distinctes puisque e appartient à un cycle. Donc $|A|$ et $|F|$ diminuent chacun de 1, et la quantité $|S| - |A| + |F|$ est **inchangée**. L'hypothèse de récurrence appliquée à $G - e$ conclut.

Q34. Si $|S| \geqslant 3$ et que le graphe est simple, toute face est bordée par au moins **trois** arêtes : $\deg(f) \geqslant 3$. En sommant et en utilisant **Q31** :

$2|A| = \Sigma_f \deg(f) \geqslant 3|F|$, donc $|F| \leqslant 2|A|/3$.

La formule d'Euler donne $|F| = 2 - |S| + |A|$, d'où

$2 - |S| + |A| \leqslant 2|A|/3$, soit $|A|/3 \leqslant |S| - 2$ et $|A| \leqslant 3|S| - 6$.

Pour K_5 : $|S| = 5$ et $|A| = C(5,2) = 10$, alors que $3\times5 - 6 = 9$. La majoration est violée : **K_5 n'est pas planaire**.

Q35. Supposons tous les degrés $\geqslant 6$. Alors $2|A| = \Sigma \deg(s) \geqslant 6|S|$, donc $|A| \geqslant 3|S|$, ce qui contredit $|A| \leqslant 3|S| - 6$ dès que $|S| \geqslant 3$.

Les cas $|S| \leqslant 2$ sont immédiats : les degrés y valent au plus 1.

Tout graphe planaire connexe possède donc un sommet de **degré au plus 5** — c'est la clé de la récurrence du théorème des cinq couleurs.

Partie B · III.2 à III.4 — Planar-k-COL et les cinq couleurs (Q37 à Q43)

- **Q37** — résoudre Planar-2-COL en temps linéaire.
- **Q38, Q39** — les graphes de la figure 5, puis trois exemples à construire.
- **Q40 à Q43** — la récurrence du théorème des cinq couleurs, par échange de couleurs le long des chaînes de Kempe.

> **Q37.** Planar-2-COL n'est rien d'autre que la bicoloration : l'algorithme de **Q22**, étendu aux graphes non connexes par **Q26**, le résout en $O(n + |A|)$.
>
> L'hypothèse de planarité apporte une chose : d'après **Q34**, $|A| \leqslant 3|S| - 6$, donc $|A| = O(n)$. La complexité devient
>
> $O(n)$,
>
> **linéaire en le seul nombre de sommets**.

Q38. La méthode est la suivante, et elle se conduit sans dessin :

1. **Compter les arêtes** et appliquer les majorations de **Q34** et **Q36**. Un graphe à 6 sommets et plus de 12 arêtes n'est pas planaire ; s'il est de plus biparti, la limite tombe à 8.
2. **Chercher K_5 ou $K_{3,3}$** comme sous-graphe, ce que le théorème de Kuratowski rend décisif.
3. Pour la 3-colorabilité, exhiber une coloration, ou invoquer **Q10** pour l'exclure.

Sur la figure, le troisième graphe joint chacun des sommets $4, 3, 5$ à chacun des sommets $0, 2, 1$: c'est $K_{3,3}$, donc **non planaire** d'après **Q36** — mais il est biparti, donc 2-colorable, a fortiori 3-colorable. Les deux premiers, qui en sont des sous-graphes stricts, restent sous la barre des 8 arêtes et sont planaires.

Cette lecture dépend de la figure, que ce document ne reproduit pas : à vérifier sur le sujet officiel. La méthode, elle, ne dépend d'aucun dessin.

Q39. Trois exemples à quatre sommets suffisent :

- **2-colorable** : le cycle C_4, de longueur paire — ou simplement un chemin.
- **3-colorable mais pas 2** : le triangle K_3, éventuellement augmenté d'un quatrième sommet pendant. Il contient un cycle impair, donc **Q10** interdit la bicoloration, et trois couleurs suffisent visiblement.
- **4-colorable mais ni 2 ni 3** : le graphe complet K_4. Il est planaire — on le dessine comme un triangle avec un sommet au centre — et ses quatre sommets étant deux à deux adjacents, il faut quatre couleurs.

Q40. Si $\deg(v) \leqslant 4$, le graphe $G' = G - v$ a n sommets et est planaire : l'hypothèse de récurrence lui donne une 5-coloration.

Les voisins de v sont au plus quatre, donc utilisent **au plus quatre couleurs** parmi cinq. Il en reste au moins une libre, que l'on attribue à v.

Q41. L'ensemble E décrit dans l'énoncé est la **composante de Kempe** de s_1 pour le couple de couleurs (a, c) : les sommets atteignables depuis s_1 par un chemin dont les couleurs alternent entre a et c.

Échanger les couleurs a et c **à l'intérieur de E seulement** produit encore une coloration valide : à la frontière de E, les voisins extérieurs ne portent ni a ni c, sinon ils appartiendraient à E.

Si $s_3 \notin E$, cet échange donne à s_1 la couleur c sans toucher à s_3, qui la garde. La couleur a n'est alors plus utilisée par aucun voisin de v : on l'attribue à v.

Q42. Dans le cas contraire, $s_3 \in E$: il existe une chaîne de s_1 à s_3 alternant a et c. Fermée par les arêtes $\{v, s_1\}$ et $\{v, s_3\}$, elle forme un **cycle** dans G.

Les voisins $s_1, \ldots, s_5$ étant énumérés dans l'ordre trigonométrique autour de v, les sommets s_2 et s_4 sont **séparés** par ce cycle : l'un lui est intérieur, l'autre extérieur. Ils ne peuvent donc pas être tous deux du même côté — c'est précisément ce qu'interdit la planarité, un dessin le rendant évident.

Q43. Il en résulte qu'aucune chaîne alternant les couleurs b et d ne peut relier s_2 à s_4 : elle devrait traverser le cycle précédent, donc en couper une arête, ce que le dessin planaire interdit.

> On applique alors l'argument de **Q41** au couple `(b, d)` et à la composante de `s_2` : l'échange libère la couleur `b` pour `v`.
>
> Dans tous les cas, `v` reçoit une couleur, et `G` est 5-colorable. La récurrence est achevée : **tout graphe planaire est 5-colorable**.
>
> *Le théorème des quatre couleurs, lui, a résisté un siècle et demi. La démonstration de Kempe de 1879, dont l'argument des chaînes reste au cœur de la preuve ci-dessus, était fausse — l'erreur n'a été relevée qu'en 1890. Il faut attendre 1976 et la première preuve assistée par ordinateur, portant sur 1 478 cas critiques.*