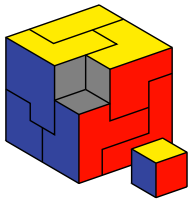
Prologin 2016



Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique

Clément Beauseigneur, Li-yao Xia [*]

20 janvier 2015

Table des matières

[*]Pour l'équipe des correcteurs 2016 : Florian Amsallem, Rémi Audebert, Clément Beauseigneur, Benjamin Bigaré, Adrien Boulay, Victor Collod, Marion Descamps, Alexis Descamps, Nicolas Ding, Guillaume Doré, Quentin Dugaugez, Sarah Fachada-Dury, Stéphane Henriot, Alexandre Macabies, Lê Thành Dũng Nguyễn, Jean-François Nguyen, Antoine Pietri, Héloïsa Vallerio, Li-yao Xia.

1 Le déluge

1.1 Énoncé

Il y a fort longtemps, existait un continent où vivait une espèce rare : le Prolosaure. Ce continent était composé de n montagnes (numérotées de 1 à n) de différentes altitudes dans lesquelles les paisibles dinosaures vivaient en paix. Un jour, une tempête se déchaîna et entraîna une montée globale du niveau de l'eau. Les Prolosaures se demandent si une partie de leur foyer a disparu sous les eaux. On vous donne une liste de nombres représentant les altitudes des différentes montagnes qui composent le continent ainsi que l'altitude globale atteinte par la montée des eaux. Écrivez un programme qui indique si au moins une montagne a été submergée par les flots.

1.2 Solution

On parcourt la liste des montagnes pour vérifier que leurs altitudes sont bien supérieures au niveau de l'eau, en interrompant la boucle si on trouve une montagne submergée.

Complexité On parcourt au maximum une seule fois la liste des sommets, la complexité est donc linéaire, soit en $O(n)$.

Listing 1 – Une solution de l'exercice 1 en Python

```
1 niveau_eau = int(input())
2 nombre_montagnes = int(input())
3 montagnes = [int(i) for i in input().split()]
4 for hauteur in montagnes:
5     if hauteur < niveau_eau:
6         print(1)
7         break
8 else:
9     print(0)
```

2 Montée des eaux

2.1 Énoncé

Le grand conseil des Prolosaures, un groupe plus malin que la plupart de leurs congénères, cherche un moyen de calculer le volume gagné par la mer lors du déluge. On vous donne une liste de nombres représentant les altitudes des différentes montagnes qui composent le continent ainsi que l'altitude globale atteinte par la montée des eaux. Écrivez un programme qui indique le volume total de l'eau au-dessus des zones inondées, sachant que la surface du sommet de chaque montagne vaut 1. Par exemple, si l'altitude d'une montagne est 20 et que l'altitude de l'eau est 23, alors cette montagne contribue pour un volume de 3 unités (voir exemples).

2.2 Solution

On fait la somme des volumes d'eau au dessus de chaque montagne.

En maintenant une variable `volume` initialisée à 0, on parcourt la liste des montagnes, et si une montagne est submergée on ajoute à `volume` le volume de l'eau au dessus d'elle. Ce volume est donné par la différence des altitudes de l'eau et de la montagne ; il est négatif quand la montagne n'est pas submergée, on peut alors écrire le tout brièvement avec la fonction `max`.

Complexité On parcourt une seule fois la liste des sommets, la complexité est donc linéaire, soit en $O(n)$.

Listing 2 – Une solution de l'exercice 2 en Python

```
1 niveau_eau = int(input())
2 nombre_montagnes = int(input())
3 montagnes = [int(i) for i in input().split()]
4 volume = 0
5 for hauteur in montagnes:
6     volume += max(0, niveau_eau - hauteur)
7 print(volume)
```

3 La prophétie

3.1 Énoncé

Le grand conseil des Prolosaures a mis la main sur une tablette très ancienne annonçant le désastre que les dinosaures sont en train de vivre. Sur cette pierre oubliée de tous, il est indiqué le volume d'eau qui est en train de noyer le monde. Pour anticiper le désastre, le conseil cherche à savoir jusqu'à quelle altitude la mer va continuer de monter.

On vous donne une liste de nombres représentant les altitudes des différentes montagnes qui composent le continent ainsi que le volume total d'eau au-dessus des terres submergées. Écrivez un programme qui indique l'altitude (arrondie à l'entier inférieur) que va atteindre l'eau une fois la catastrophe terminée. Plus précisément, étant donné le volume V d'eau déversé par la tempête, on vous demande de trouver l'entier h tel que :

— si le niveau de la mer est h, alors le volume de l'eau sur les zones inondées est au plus V ;
— si le niveau de la mer est $h+1$, alors le volume d'eau sur les terres est strictement supérieur à V.

3.2 Solution naïve

Ce n'était pas la solution attendue, elle passe les tests de corrections mais pas les tests de performance. Il s'agit de tester toutes les hauteurs d'eau possibles, en calculant le volume d'eau correspondant (en réutilisant votre solution pour l'exercice 2), jusqu'à tomber sur celle qui convient.

Complexité Pour chaque hauteur possible on parcourt la liste des montagnes, la complexité est donc en $O(n \times H)$, où H est la différence entre la hauteur minimum et la hauteur maximum possibles.

Listing 3 – Une solution naïve de l'exercice 3 en Python

```
volume = int(input())
nombre_montagnes = int(input())
montagnes = [int(i) for i in input().split()]


def volume_submergeant(hauteur_eau):
    volume = 0
    for hauteur_montagne in montagnes:
        volume += max(0, hauteur_eau - hauteur_montagne)
    return volume


hauteur_eau = 100000
while volume_submergeant(hauteur_eau) > volume:
    hauteur_eau -= 1

print(hauteur_eau)
```

3.3 Perfectionnement de cette solution

Cette solution pouvait être facilement améliorée en cherchant la bonne valeur de la hauteur d'eau *par dichotomie*. À chaque étape, on divise la taille de l'intervalle de recherche par deux, en comparant le volume prophétisé avec celui atteint par la valeur au centre de cet intervalle.

Cet algorithme repose crucialement sur le fait que plus l'eau monte, plus son volume augmente : le volume est une fonction croissante de la hauteur. Comparer les volumes nous permet ainsi de déduire la moitié de l'intervalle à laquelle la hauteur recherchée appartient.

Complexité On parcourt la liste des montagnes à chacune des $O(\log(H))$ étapes de la dichotomie, la complexité de cette solution est donc en $O(n \times \log(H))$.

Listing 4 – Solution par recherche dichotomique en Python

```python
volume = int(input())
nombre_montagnes = int(input())
montagnes = [int(i) for i in input().split()]


def volume_submergeant(hauteur_eau):
    volume = 0
    for hauteur_montagne in montagnes:
        volume += max(0, hauteur_eau - hauteur_montagne)
    return volume


hauteur_inf = -100000
hauteur_sup = 100000
# À chaque étape la réponse est dans l'intervalle [hauteur_inf, hauteur_sup]
while hauteur_sup > hauteur_inf:
    hauteur_test = (hauteur_inf + hauteur_sup + 1) // 2
    if volume_submergeant(hauteur_test) <= volume:
        hauteur_inf = hauteur_test
    else:
        hauteur_sup = hauteur_test-1

print(hauteur_sup)
```

3.4 Solution par balayage

Beaucoup de candidats ont eu cette autre idée, qui consiste à trier la liste des montagnes par hauteur croissante pour ensuite remplir d'eau progressivement. En effet tant que l'eau n'a pas atteint le sommet de la deuxième plus petite montagne, on peut se contenter de remplir au-dessus de la plus petite, et une fois le niveau de la deuxième atteint on peut remplir au-dessus des deux premières montagnes simultanément jusqu'à atteindre le troisième, et ainsi de suite jusqu'au volume désiré.

Au lieu de faire monter la hauteur de l'eau unité par unité, on la met directement au niveau de la prochaine montagne non submergée. Si on a dépassé le volume d'eau prophétisé, alors on peut revenir à la bonne hauteur avec une simple expression arithmétique, sinon on passe à la montagne suivante.

Complexité L'opération la plus coûteuse ici est le tri, car une fois triée on ne parcourt la liste des montagnes qu'une seule fois. En employant *un algorithme de tri efficace*, implémenté par vous-mêmes ou trouvé dans la bibliothèque standard de votre langage, cette solution est en $O(n \times \log(n))$.

Listing 5 – Solution par balayage en Python

```python
v_restant = int(input())
nb_montagnes = int(input())
montagnes = [int(i) for i in input().split()]
montagnes.sort()
i = 1 # Nombre de montagnes submergées
while i < nb_montagnes and v_restant > i * (montagnes[i] - montagnes[i-1]):
    v_restant -= i * (montagnes[i] - montagnes[i-1])
    i += 1
print(v_restant // i + montagnes[i-1])
```

4 Après le déluge

4.1 Énoncé

Grâce aux prévisions avérées du grand conseil des Prolosaures, un certain nombre de survivants au déluge ont pu se réfugier sur quelques îles émergées. Pour communiquer, un pacte a été conclu avec les Algoptéryx, dinosaures volants, grâce auxquels les bases de la communication ont pu s'établir entre les différentes îles. Malheureusement, ceux-ci commmencent à se lasser de leur rôle trop peu rémunérateur.

En guise de solution, le grand conseil des Prolosaures a imaginé déployer un grand réseau de téléphones yaourt entre les îles, grâce au concours des Algoptéryx.

Malheureusement, le fil nécessaire à leur confection se fait rare en ces temps post-apocalyptiques, et une récompense est prévue pour celui qui en minimisera l'usage.

Écrivez un programme qui permet de déterminer la longueur de fil minimale nécessaire pour rallier les différentes îles : toute île doit pouvoir communiquer, directement ou non, avec toutes les autres îles.

4.2 Solution

On dit d'un algorithme qu'il est *glouton* quand il fait des choix qui sont optimaux sur le court terme. C'est un type général d'algorithme bien utile pour les problèmes d'optimisation. Certains s'avèrent être optimaux globalement, d'autres non, mais ils s'en approchent parfois de très près tout en étant rapides à exécuter.

Pour ce problème, des algorithmes gloutons donnent la bonne solution.

4.2.1 Algorithme de Prim

En commençant avec une seule île arbitrairement choisie, on étend progressivement un ensemble d'îles interconnectées en reliant à chaque fois la paire d'îles les plus proches dont exactement une se trouve déjà dans cet ensemble.

Nous en donnons une implémentation ci-dessous. Elle utilise un tableau `cost` pour garder en mémoire la distance minimale de chaque île à l'ensemble connecté, qui se met à jour en temps linéaire $O(n)$ pour chaque itération.

Complexité en temps $O(n^2)$, en mémoire $O(n)$.

Listing 6 – Une solution de l'exercice 4 en Python

```python
from math import sqrt


def prim(n, points):
    def distance(i, j):
        pi = points[i]
        pj = points[j]
        return sqrt((pi[0] - pj[0]) ** 2 + (pi[1] - pj[1]) ** 2)

    total = 0
    connected = [False] * n
    connected[0] = True

    # Le coût minimum de relier chaque île à l'ensemble d'îles connectées
    cost = [distance(0, i) for i in range(n)]

    # On veut connecter les n-1 autres îles a 0
    for _ in range(n-1):
        # On trouve l'île non connectée la plus proche de l'ensemble
```

```
        min_i, min_cost = 0, float('inf')
        for i in range(n):
            if not connected[i] and cost[i] < min_cost:
                min_i, min_cost = i, cost[i]

        # Et on la connecte
        connected[min_i] = True
        total += min_cost
        for j in range(n):
            cost[j] = min(cost[j], distance(min_i, j))
    return total

n = int(input())
points = [list(map(int, input().split())) for _ in range(n)]
print(int(prim(n, points)))
```

Un autre algorithme glouton, aussi connu, est l'algorithme de Kruskal, qui examine toutes les paires d'îles par distances croissantes, en faisant attention à :

1. ne pas relier des îles déjà indirectement reliées, c'est-à-dire qu'on peut aller de l'une à l'autre en suivant des liaisons déjà établies ;
2. et ne finir qu'une fois que toutes les îles sont bien reliées entre elles.

Cela donne un algorithme de complexité en temps $O(n^2 \times \log(n))$ (pratiquement équivalent à $O(n^2)$), mais aussi $O(n^2)$ en mémoire car il faut ordonner toutes les paires d'îles dès le début. Cela ne devrait pas passer les tests de performance avec la contrainte de 1Mo de mémoire. [1]

Beaucoup de soumissions correspondent à un de ces deux algorithmes en idée, mais n'atteignent pas du tout la complexité attendue car elles n'utilisent pas de structures de données efficaces : le coût minimum est ainsi souvent recalculé à partir de zéro, en réexaminant toutes les paires possibles à chaque itération ; ce qui donne une complexité en au moins $\Omega(n^3)$.

Il faut aussi arrondir à un entier seulement la réponse finale, en évitant donc d'arrondir les distances entre les îles dans le reste du programme ; le calcul devrait être fait avec des nombres flottants, les pertes de précision s'avérant être négligeables.

4.3 Bonus : en temps quasi-linéaire

Malgré les apparences, ce problème peut être résolu encore plus rapidement qu'en temps quadratique. Cela peut sembler a priori impossible, car il y a $\frac{n(n-1)}{2}$ liens possibles à examiner. Pourtant tous n'en valent pas le coup. Nos îles étant des points du plan, on peut réduire à environ $3n$ le nombre de liens potentiels, grâce à la *triangulation de Delaunay* [2], qui se calcule en temps $O(n \log n)$.

On peut ensuite appliquer, entre autres, l'algorithme de Prim ou de Kruskal dont les vraies complexités [3] sont $O(m + n \times \log n)$ et $O(m \times \log(m))$ respectivement, où m est le nombre d'arêtes dans le graphe. Comme $m \approx 3n = O(n)$, on a donc dans tous les cas, la complexité finale en temps $O(n \times \log(n))$ (et $O(n)$ en espace).

4.4 Pour en savoir plus

Ce problème est mieux connu sous le nom d'*arbre couvrant de poids minimal*, plus précisément dans le *cas euclidien*.

1. Il faudrait allouer moins de 2 octets par paire d'îles (il y en a environ 500000 pour $n = 1000$), ce qui n'est a priori pas faisable même en étant très radin.
2. Justification omise.
3. Encore une fois, avec les bonnes structures de données. En particulier, on préférera une représentation par listes d'adjacences, et l'algorithme de Prim nécessite alors une file de priorité plus sophistiquée, dans le but de sélectionner et relier chaque sommet en $O(\log(n))$.

5 Un peu de compagnie

5.1 Énoncé

Une nouvelle problématique vient troubler la quiétude post-apocalyptique des Prolosaures : suite à la solitude qui règne sur leurs îles, les Prolosaures en sont venus à se détester, de telle sorte que la plupart d'entre eux ont au moins un rival à haïr. Le téléphone yaourt, qui ne permet pas les communications simultanées, sert alors de carburant, alimentant la haine entre rivaux.

Le grand conseil, préoccupé par la situation, pense avoir trouvé une solution : sur l'une des îles se trouvait une fabrique de talkie-walkies, fonctionnant par paires, dont la portée permet de couvrir toute l'étendue de l'archipel Prolosaure.

Il suffirait que ne soient mis en contact que des couples de Prolosaures entretenant de bonnes relations, en prenant garde à ne laisser personne dans la solitude. Il n'est par contre pas nécessaire de rallier un même individu à plusieurs autres, à moins que leurs rivalités ne l'imposent.

Toutefois, les talkie-walkies se font rares, et le conseil en vient à se demander s'il sera un jour possible de mettre ce plan à exécution.

À partir du nombre d'îles et d'une liste de couples d'îles dont les habitants ne souhaitent pas communiquer avec ceux de l'autre île, écrivez un programme qui indique le nombre minimal de couples de talkie-walkies nécessaires. On vous garantit qu'aucun Prolosaure n'est détesté de tous.

Remarque Contrairement à l'exercice précédent, on ne demande pas ici que toute paire d'îles soit reliée, même indirectement.

Notations On désignera une liaison par la paire de ses extrémités $\{i, j\}$, (égale à $\{j, i\}$) et on dira aussi qu'elle *couvre* ses extrémités. Un ensemble de liaisons qui couvrent tout l'archipel est appelé une *couverture*, dont nous cherchons ici la taille minimum.

5.2 Mauvaise solution

Pour ce problème, les algorithmes gloutons ne sont pas corrects. Plus précisément, il s'agit des stratégies consistant à distribuer des talkie-walkie jusqu'à couvrir l'archipel, parfois à l'aide d'heuristiques telles que privilégier les îles qui ne sont pas encore couvertes, et ayant le plus de rivaux.

Contre-exemple Considérez le graphe donné par Fig. 1a. Un algorithme glouton choisirait la liaison $\{0, 1\}$ en premier (si ce n'est pas le cas, on peut adapter le graphe pour contrer les heuristiques employées). Cela débouche nécessairement sur une couverture utilisant cinq paires de talkie-walkies, tandis que la seule couverture optimale en utilise quatre, illustrée en Fig. 1c.

5.3 Problème équivalent : le couplage maximum

Lorsqu'on construit une couverture, il est naturel de choisir des liaison qui contribuent à couvrir deux îles d'un seul coup tant que possible.

Dans une couverture donnée, nous pouvons rendre explicite cette idée de « contribution » de chaque liaison en désignant [4], pour chaque île, une liaison qui la couvre. Ainsi une liaison peut être désignée une ou deux fois [5]. Nous parlerons respectivement de liaisons de type 1 et de type 2. Voir par exemple Fig. 1.

Les liaisons de type 2 forment un *couplage* : c'est un ensemble de liaisons toutes disjointes. C'est-à-dire qu'il n'y en a pas deux avec une île en commun.

La procédure de « désignation » décrite ci-dessus produit un couplage à partir d'une couverture. Réciproquement, si nous avons un couplage de m_2 liaisons, il couvre exactement $2m_2$ îles, donc il reste $n - 2m_2$ îles non appariées que nous pouvons simplement couvrir en ajoutant une liaison pour

4. Une telle désignation n'est généralement pas unique.

5. Une liaison désignée zéro fois est de type inutile : l'enlever nous économise une paire de talkie-walkies. Nous supposons sans perte de généralité qu'il n'y en a pas.

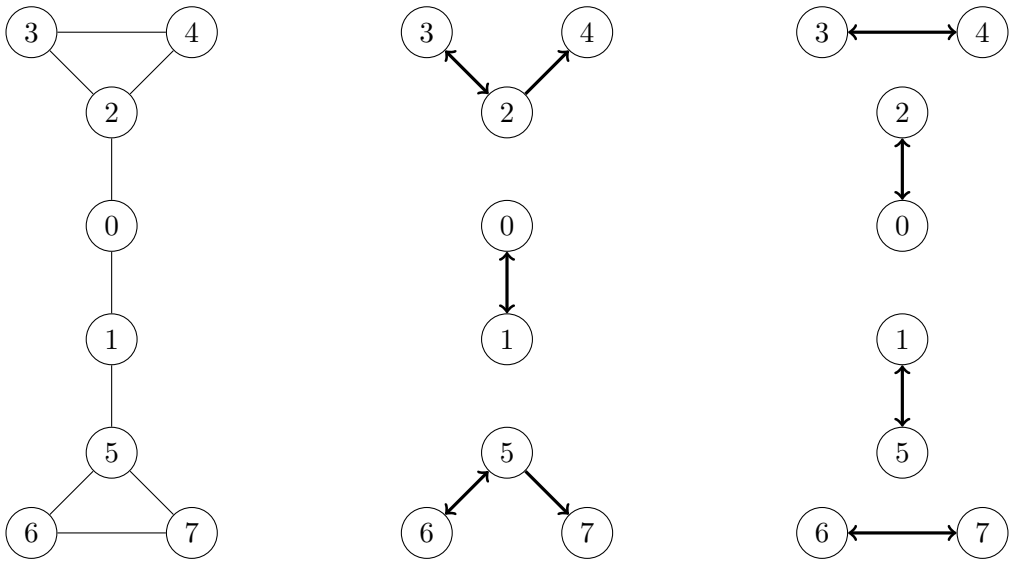


FIGURE 1 – Exemples de couvertures.

Dans les deux couvertures, chaque île désigne par une pointe de flèche incidente une liaison qui la couvre.

1a Sur ce dessin (un *graphe*) chaque île est représentée par un cercle (un *sommet*) étiquetté par son identifiant, et deux îles sont reliées si et seulement si elles ne sont *pas* rivales. (Ces deux îles forment une *arête*, correspondant à une liaison possible.)

1b Couverture sous-optimale, forcée par la sélection gloutonne de $\{0, 1\}$. Avec deux liaisons de type 1 et trois de type 2.

1c Couverture optimale. Avec quatre liaisons de type 2.

chacune[6] ; nous obtenons une couverture de $n - m_2$ liaisons. Nous cherchons à minimiser ce nombre, donc à maximiser le nombre de liaisons du couplage, m_2.

5.4 Solution

Nous allons partir d'un couplage quelconque (vide par exemple) et l'améliorer progressivement jusqu'à ce qu'il soit (de taille) maximum. Mais à la différence d'un simple algorithme glouton qui ne fait que rajouter des liaisons sans jamais en enlever, nous allons plutôt remplacer des groupes de liaisons par des groupes plus gros.

5.4.1 Chemin augmentant

Un *chemin* de longueur w est une suite finie d'îles $(i_0, i_1, \ldots, i_{w-1}, i_w)$ telles que deux îles consécutives ne sont jamais rivales, et toute île y figure au plus une fois (un chemin ne se recroise donc pas). Il y a $w + 1$ îles dans un chemin de longueur w, qui est en fait le nombre de liaisons potentielles successives entre les îles : $\{i_0, i_1\}, \ldots, \{i_{w-1}, i_w\}$.

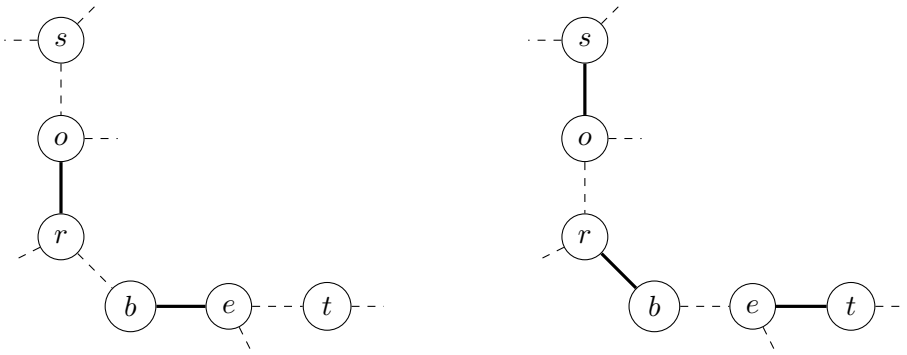
Étant donné un couplage C, un chemin $(i_0, i_1, \ldots, i_{w-1}, i_w)$ entre deux îles $s = i_0$ (*source*) et $t = i_w$ (*target*[7]) est *augmentant*[8] si :

1. les extremités s et t sont *exposées* : elles ne sont pas couvertes par une liaison de C ;
2. le chemin est *alternant* : une liaison sur deux appartient à C.

6. Nous rappellons qu'il n'y a pas de Prolosaure haï de tous.
7. Ou « la lettre après s ».
8. Sous-entendu, il augmente *ce* couplage C, mais il ne sera question que d'un couplage à la fois.

Nécessairement, w est impair, et on a $\{i_z, i_{z+1}\} \in C$ pour z impair et $z < w$.

Comme son nom l'indique, un chemin augmentant permet d'augmenter la taille du couplage C : il suffit d'échanger les liaisons du chemin appartenant à C avec celles qui n'y appartiennent pas ; Fig. 3 donne un exemple.



(a) Avant augmentation. (b) Après augmentation.

FIGURE 3 – Amélioration d'un couplage par un chemin augmentant.
Les traits pleins sont les liaisons du couplage, et les traits pointillés indiquent les liaisons potentielles, c'est-à-dire les paires d'îles non rivales.

Il s'avère [9] que tout couplage non-maximum a au moins un chemin augmentant, nous pouvons donc procéder comme suit :

Squelette d'algorithme pour résoudre l'exercice 5

procédure COUPLAGEMAX(G)
  $C \leftarrow \emptyset$
  tant que il existe un chemin augmentant c **faire**
    Augmenter C à l'aide de c
  renvoyer C ▷ Il n'y a plus de chemin augmentant : C est maximum.

5.4.2 Algorithme d'Edmonds

Il reste à décrire l'algorithme de recherche d'un chemin augmentant, où réside maintenant toute la difficulté du problème.

Partons d'une île r exposée. S'il n'y en a pas, alors il ne peut pas non plus y avoir de chemin augmentant, donc le couplage C est maximum.

Une île i est *paire* s'il y a un chemin alternant (rappel : une liaison sur deux appartient à C) de longueur paire entre r et i. Quelques remarques :

— il peut y avoir d'autres chemins alternants de longueurs impaires entre r et i ;
— puisque r est exposé, l'alternance commence avec une liaison qui n'est pas dans C, et la dernière liaison du chemin alternant est donc l'unique liaison appartenant à C et couvrant i ;
— trouver un chemin augmentant de r vers une île exposée s revient à trouver une île paire parmi les amies (i.e., îles non rivales) de s ;
— si on enlève la dernière île i et l'avant-dernière, un chemin alternant $(r, i_1, \ldots, i_{w-2}, i_{w-1}, i)$ de longueur non-nulle et paire w se raccourcit en un chemin alternant $(r, i_1, \ldots, i_{w-2})$ de longueur paire $w - 2$: la nouvelle extrémité est ainsi paire.

Imaginons un chemin augmentant issu de r, vers une île exposée s. Chaque île à l'intérieur du chemin est couverte par exactement une liaison du couplage C. On peut donc le décrire avec juste une île sur deux, disons celles d'indices pairs, les autres se déduisant en suivant les uniques liaisons de C associées.

Cela suggère d'utiliser un *arbre* pour représenter un ensemble d'îles paires et leurs chemins alternants associés issus de r, qui en sera la *racine* : à chaque île i distincte de la racine, cet arbre

9. Preuve omise.

associe une autre île, appelé *prédécesseur* de i, de telle sorte qu'en allant de prédécesseur en prédécesseur, on arrive toujours à la racine. Pour illustrer, l'arbre Fig. 5a représente l'ensemble d'îles paires $\{1, 3, 4, 5, 10\}$ ($r = 3$) dans le graphe Fig. 4, où par exemple, le prédécesseur de 10 est 5.

Ce type d'arbre se code très simplement avec un tableau de prédécesseurs, Fig. 5b en donne une idée. Chaque île i est associée à une case, qui est :

— noire, si l'île n'est pas paire (ou plus précisément, si on n'a pas encore trouvé de chemin alternant de longueur paire) ;
— grise, si l'île est la racine r ;
— blanche, et contenant l'identifiant de l'île paire qui la précède dans un chemin alternant de longueur paire, si i est elle-même paire.

L'intérêt de cet structure est que pour chaque île i distincte de r, cet arbre induit un chemin alternant entre les îles r et i, en partant de la fin :

1. la dernière île est bien-sûr i ;
2. l'avant-dernière est l'autre extrémité t de l'unique liaison du couplage C qui couvre i ($t = 6$ pour $i = 4$; voir Fig. 4) ;
3. l'antépénultième [10] île est le prédécesseur p de i ($p = 5$ pour $i = 4$; voir Fig. 5a) et si $p \neq r$ on réapplique les règles ci-dessus pour construire le chemin restant entre r et p.

Pour $i = 4$ on obtient $(3, 2, 5, 6, 4)$.

Notre algorithme commence avec l'arbre-racine $A = \{r\}$, et agrandit A progressivement avec des îles paires. Tant que possible, on applique une des opérations ci-dessous à une liaison $\{i, j\}$ *adjacente* à A, c'est-à-dire qu'elle n'appartient pas à C, et i est une île paire déjà dans l'arbre.

Croissance Si l'île j n'appartient pas à A, si elle est couverte par une liaison $\{j, k\}$ de C, et si k n'est pas non plus dans A, alors k est une nouvelle île paire, qu'on ajoute à A avec pour prédécesseur l'île i.

Floraison Si comme i, l'île j est paire et appartient à A, alors en étendant le chemin alternant de r à j avec le chemin alternant inversé de i à r induit par A, certaines îles peuvent se révéler être paires, quand elles ne sont pas déjà dans A.

Terminaison Si l'île j est exposée et distincte de r, alors on étend le chemin alternant de r à i en un chemin augmentant d'extrémités r et j, et on a fini.

Sinon, quand plus aucune des règles ci-dessus ne s'applique, c'est qu'il n'y a pas de chemin augmentant issu de r. On peut dans ce cas continuer à chercher un chemin augmentant à partir d'une autre île exposée. Une optimisation facultative consiste à enlever les îles paires et celles accessibles à partir de celles-là en suivant juste une liaison de C.

Petit exemple Par applications répétées de la règle de *croissance*, on peut produire l'arbre Fig. 5a. En l'appliquant encore à la liaison adjacente $\{4, 7\}$, on peut prolonger le chemin alternant $(3, 2, 5, 6, 4)$ avec $(7, 11)$. L'île 11 est donc une île paire. On obtient le résultat Fig. 6a.

Grâce à la liaison adjacente $\{10, 11\}$, on peut prolonger par *floraison* le chemin $(3, 2, 5, 9, 10)$ en remontant l'autre chemin à l'envers $(11, 7, 4, 6, \dots)$. Ainsi 6 et 7 sont des îles paires ; pour ajouter 9, on fait de même en inversant les rôles des deux chemins. Il faut faire attention à s'arrêter avant la dernière île en commun entre les deux chemins, ici 5. En effet, 2 n'est pas une île paire. On obtient le résultat Fig. 6b.

Ensuite, l'île 7 est reliée à 8, qui est exposée ; on en déduit le chemin $(3, 2, 5, 9, 10, 11, 7, 8)$ qui est augmentant (*terminaison*).

Complexité Calculons la complexité de cet algorithme en fonction du nombre d'îles n et de liaisons possibles m. Attention : il s'agit du nombre de paires d'îles *non* rivales ; $m = \frac{n(n-1)}{2} - m'$, où m' est le nombre de paires données en entrée. Nos exemples avaient une valeur m de l'ordre de n plutôt que n^2, et inversement pour m'. Pour cette raison, l'analyse suivante sépare ces paramètres au lieu de simplement majorer m par n^2.

10. Avant-avant-dernière.

On effectue une recherche de chemin augmentant à partir de chaque île. Chacune examine une liaison au plus deux fois [11]. Pour chaque liaison, on est dans un des cas : rien à faire, terminaison, croissance, tous en $O(1)$, et floraison en $O(n)$ [12]. Au pire, toutes les m liaisons donnent lieu à des floraisons (une situation très pessimiste à vrai dire), pour un total de $O(n \times m)$ opérations par recherche. Finalement, le total est de $O(n^2 \times m)$ en temps, $O(n^2 - m)$ en mémoire [13], les autres structures de données utilisées étant pour l'essentiel des tableaux de longueur n (on a $\frac{n}{2} \leq m$).

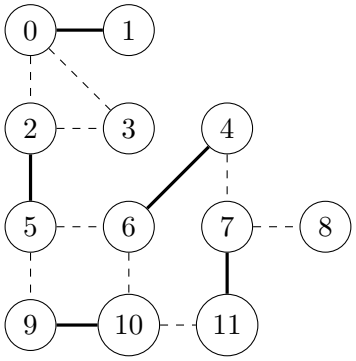


FIGURE 4 – Un graphe avec un couplage non-maximum, en traits pleins.

(a) (b) Visualisation compacte de l'arbre en Fig. 5a.

FIGURE 5 – Représentation d'un ensemble d'îles paires et de chemins alternants dans le graphe donné en Fig. 4, par un arbre enraciné en 3.

(a) Opération de croissance à partir de la Fig. 5a (b) Opération de floraison à partir de la Fig. 6a

FIGURE 6 – Agrandissement de l'arbre. Les pointillés indiquent les derniers ajouts.

11. Il est suffisant de le faire juste quand une extrémité est ajoutée.

12. On frôle l'arnaque sur un point subtil : une partie des opérations qui ont lieu lors d'une croissance ou d'une floraison doit être comptée séparément par analyse amortie. En particulier il s'agit, lorsqu'on rencontre une nouvelle île paire, d'insérer les liaisons couvrantes possibles dans une pile ; plus exactement, la ligne 53 du code ci-dessous. Cela coûte $O(m)$ *par recherche*, ce qui est négligeable à côté du total $O(nm)$ des autres opérations.

13. À cause de la lecture de l'entrée. Grâce à une représentation par listes d'adjacence, le graphe passé à la fonction `max_matching_size` a une taille $O(m)$.

5.5 Bonus : algorithme randomisé

Une soumission s'est basée sur un algorithme original, simple à mettre en œuvre, mais complexe à justifier. On génère une matrice antisymétrique aléatoire M de la manière suivante :

— pour deux îles i et j, supposons $i < j$, et si elles ne sont pas rivales, alors $M_{i,j}$ vaut un entier aléatoire entre 1 et R (un gros nombre arbitraire), et $M_{j,i} = -M_{i,j}$;

— si elles sont rivales, on pose $M_{i,j} = M_{j,i} = 0$.

Ensuite, pour des raisons qui sortent largement du cadre de ce document, on calcule le rang r de cette matrice et, avec probabilité $1 - \frac{n}{R}$, le nombre minimum de liaisons pour former une couverture vaut exactement $n - \lceil \frac{r}{2} \rceil$. Sinon ce nombre est strictement plus grand que la valeur attendue, ce qui nous permet de répéter ce calcul plusieurs fois pour améliorer la précision du résultat. Pour $R = 4n$, en réessayant 4 fois, on a le bon résultat avec probabilité 0.996. Ainsi, cet algorithme n'est pas *toujours* correct, mais il l'est *très souvent*, et ce pour n'importe quelle entrée, ce qui s'avère être suffisant pour passer des tests simples.

La complexité de cet algorithme est dominée par le calcul du rang, qui peut se faire par pivot de Gauss en $O(n^3)$. D'autres algorithmes peuvent calculer le rang d'une matrice en $O(n^{2.38})$, mais sont des curiosités plus théoriques que pratiques, surtout pour nos cas de tests relativement petits ($n \leq 200$).

5.6 Pour en savoir plus

Le premier algorithme connu pour résoudre ce problème est dû à J. Edmonds (*algorithme d'Edmonds*, ou *blossom algorithm*) qui emploie une terminologie très imagée dans son article *Paths, Trees, and Flowers* (1965). Élégant à présenter, cet algorithme est toutefois pénible à coder tel quel. Nous avons décrit ici et implémenté ci-dessous une légère variante publiée par C. Witzgall et C. T. Zahn Jr. (*Modification of Edmonds' Maximum Matching Algorithm*, 1965).

Ce dernier algorithme randomisé pour calculer la *taille* minimum d'une couverture (et de manière équivalente, la taille maximum d'un couplage) date de 1979 (*On Determinants, Matchings, and Random Algorithms*, L. Lovász), et un résultat de 2004 s'est basé dessus pour trouver explicitement une telle couverture avec la même complexité (*Maximum Matchings via Gaussian Elimination*, M. Mucha, P. Sankowski).

Listing 7 – Une solution de l'exercice 5 en Python

```python
# graph[i] est la liste des îles amies de i.
def max_matching_size(n, graph):
    # Un couplage. Une liaison (i, j) y appartient si matching[i] = j et
    # matching[j] = i. Les îles non-couvertes k ont matching[k] = -1.
    matching = [-1] * n
    matching_size = 0

    # Cherche un chemin augmentant partant de r, et effectue l'augmentation
    # associée s'il y en a un.
    def plant_tree(r):
        # L'arbre A des îles paires est donne par le tableau de prédécesseurs
        # On a pred[i] = -1 si i n'est pas dans l'arbre, ou si i == r.
        pred = [-1] * n
        queue = [(r, j) for j in graph[r]] # File de liaisons adjacentes

        # Si on trouve une île paire i dont le voisin j est exposé,
        # il y a un chemin augmentant. On augmente alors le couplage.
        # (a b-c d-e...x-y z) devient (a-b c-d e...x y-z)
        def augment(i, j):
            nonlocal matching_size
            matching_size += 1
            while i != -1:
                mi = matching[i]
                matching[i] = j
                matching[j] = i
                i, j = pred[i], mi

        # Liste des îles d'indices impairs sur
        # le chemin de i vers r le long de A
        def path_from(i):
            p = []
            while i != r:
                p.append(matching[i])
                i = pred[i]
            return p

        # On enlève la partie commune des deux chemins
        def strip_common_suffix(p, q):
            while p and q and p[-1] == q[-1]:
                p.pop()
                q.pop()

        # On forme un chemin p d'îles paires partant de i,
        # ajoutant de nouvelles îles a A.
        def backtrace(i, p):
            nonlocal queue
            # Il faut en fait s'arrêter à la dernière île
            # qui n'est pas déjà dans A.
            while p and pred[p[-1]] != -1:
                p.pop()
            for j in p:
                if pred[j] == -1:
```

```
53                  queue += [(j, k) for k in graph[j]]
54              pred[j] = i
55              i = j
56
57          while queue:
58              i, j = queue.pop()
59              if j == r or matching[j] == i: # Liaison non adjacente
60                  continue
61              if matching[j] == -1: # Terminaison
62                  augment(i, j)
63                  return
64              elif pred[j] != -1: # Floraison
65                  pi = path_from(i)
66                  pj = path_from(j)
67                  strip_common_suffix(pi, pj)
68                  backtrace(j, pi)
69                  backtrace(i, pj)
70              elif pred[matching[j]] == -1: # Croissance
71                  backtrace(i, [matching[j]])
72
73      for r in range(n):
74          if matching[r] == -1:
75              plant_tree(r)
76      return matching_size
77
78
79  def read_graph():
80      n = int(input())
81      m = int(input())
82      graph = [[i] for i in range(n)]
83      for i in range(m):
84          ra, rb = list(map(int, input().split()))
85          graph[ra].append(rb)
86          graph[rb].append(ra)
87      for i in range(n):
88          coedges = []
89          e0 = 0
90          graph[i].sort()
91          graph[i].append(n)
92          for j in graph[i]:
93              coedges += list(range(e0, j))
94              e0 = j+1
95          graph[i] = coedges
96      return n, graph
97
98
99  n, graph = read_graph()
100 print(n - max_matching_size(n, graph))
```

Félicitations à tous les participants !

Nous avons tenté de rédiger une correction aussi claire que possible. Néanmoins, si vous avez des questions, n'hésitez pas à nous contacter à l'adresse info@prologin.org.