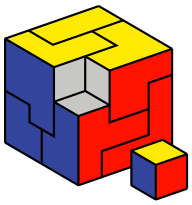
Prologin 2020



Concours National d'Informatique

Correction de la phase de sélection

Correction des qualifications Prologin 2020

Thibault Allançon, Rémi Dupré, Garance Gourdel, Tanguy Segarra*

5 janvier 2020

Table des matières

*Pour l'équipe des correcteurs 2020 : Thibault Allançon, Étienne Benrey, Amélie Bertin, Maya El-Gemayel, Valérian Fayt, Paul Guénézan.

1 Fuel

1.1 Énoncé

Avant de décoller, l'équipage du vaisseau Nigolorp doit prévoir la quantité de carburant à charger afin de tenir tout le voyage. Le capitaine estime qu'il faut prévoir 60kg de carburant par personne.

Cependant, comme certains membres du vaisseau sont plus ou moins enrobés, il s'inquiète et décide de rajouter 20kg de carburant supplémentaire pour chaque personne pesant **strictement** plus de 90kg.

Le but de l'équipage du Nigolorp est d'écrire un programme capable de calculer la quantité de carburant à emporter en fonction des poids des différents membres de l'équipage et l'afficher.

1.2 Solution

La solution la plus simple de ce problème est d'incrémenter régulièrement la quantité de fuel nécessaire, en fonction du poids du passager. On va d'abord récupérer en entrée la liste des poids des passagers. On itère ensuite sur cette liste, pour chaque personne, on ajoute 60 à la quantité de fuel nécessaire, mais nous devons tester si le poids en question est strictement supérieur à 90 ; si c'est le cas on doit ajouter 20 en plus à la quantité de fuel. On affiche finalement la quantité totale de fuel nécessaire.

Cette solution s'effectue en temps linéaire, en fonction du nombre de passagers (soit le nombre de poids passés en entrée) :

```
1  def fuel_compute():
2      nb = int(input())
3      p = list(map(int, input().split()))
4      fuel = 0
5      for v in p:
6          if v > 90:
7              fuel += 20
8          fuel += 60
9      return fuel
10
11 print(fuel_compute())
```

2 Interférences

2.1 Énoncé

L'équipage du vaisseau Nigolorp vient de perdre le contrôle du mastodonte. Très rapidement ils envoient un signal de détresse mais avec tous ces astéroïdes dans l'espace, le message devient rapidement pollué par des interférences.

Par chance, le vaisseau Prolo220 qui passait non loin capte le message mais doit d'abord se débarrasser des caractères ajoutés dus aux interférences.

Tout ‘.‘ dans le message doit être supprimé et lorsqu'un ‘*‘ est lu dans le texte, tous les caractères qui suivent sont ignoré jusqu'à ce qu'on lise à nouveau un ‘*‘.

Le but de l'équipage du Prolo220 est d'écrire un programme capable de retirer les caractères polluants d'un message et d'afficher le message correct.

2.2 Solution

Pour résoudre ce problème, on initialise d'abord une chaîne de caractères vide. Le principe est d'ajouter au fur et à mesure les caractères du message initial dans la nouvelle chaîne, si et seulement si le caractère ne représente pas un caractère corrompu. Cependant, nous devons faire attention au caractère '*' qui représente le début d'une **zone** corrompue du message. On ne veut rien garder du texte initial contenu entre deux '*'. Pour cela nous utiliserons une variable supplémentaire contenant une valeur booléenne (vrai ou faux) qui indique si on se trouve ou non dans une zone corrompue.

Cet algorithme s'effectue en temps linéaire, on parcourt simplement la chaîne de caractères une seule fois, en avançant caractère par caractère :

```
1  def remove_interference(s):
2      fs = ""
3      interference = False
4      for c in s:
5          if c == '*':
6              interference = not interference
7          if c != '*' and c != '.' and not interference:
8              fs += c
9      return fs
10
11 length = int(input())
12 text = input()
13 print(remove_interference(text))
```

3 Marché nocturne

3.1 Énoncé

Entre certaines missions vous prenez du temps pour vous balader dans les marchés extra-terrestres, un événement qui vous fascine tout particulièrement, pour son incroyable mélange de culture et de curiosité venu de toutes les galaxies. Cela sera l'occasion de ramener un souvenir de votre dernière mission P.A.N.D.A (Parachutage Auto Négociant la Descente Astrale).

Vous avez repéré un marchand de minerai rare et inconnu de la planète Terre. En tant que fin connaisseur des cultures extra-terrestres, vous êtes évidemment au courant de la coutume locale, et de l'importance de la **contigüité**. Pour faire plaisir au vendeur (qui semble grincheux et légèrement xénophobe), vous vous appliquez donc pour choisir uniquement des minerais **contigus** sur le fil.

Pour éviter que ce souvenir prenne trop de place, vous souhaitez choisir un nombre **minimal** de minerai, toujours contigus sur le fil, tel que la somme de leurs prix soit exactement égale à votre budget cadeau B.

3.2 Solution

Tout d'abord, reformulons le problème d'un point de vue purement algorithmique. On nous demande de trouver la taille minimale d'un sous tableau tel que la somme des éléments de ce sous tableau soit égale à B.

Une première solution naïve et évidente consiste simplement à tester chaque sous tableau. Dès lors que la somme des éléments dépasse B, ce sous tableau ne peut pas satisfaire la contrainte de l'énoncé et nous pouvons passer au suivant. En revanche, si la contrainte est vérifiée (et donc que la somme est égale à B), on garde en mémoire la taille si cette dernière est la plus petite rencontrée jusqu'à présent. La **complexité en temps** de cet algorithme est de l'ordre de $\mathcal{O}(N^2)$, ce qui n'est malheureusement pas assez rapide pour passer les tests de performance où $N \leq 10^6$.

En observant de plus près cet algorithme non optimal, on se rend compte assez vite qu'en passant d'un sous tableau à un autre, nous recommençons entièrement le calcul de la somme des éléments alors qu'elle était déjà partiellement calculée ! Une façon plus intelligente de résoudre le problème consisterait donc à mettre à jour un sous tableau, en ajoutant des éléments mais aussi en les **retirant**. Au lieu de recommencer le calcul de somme à 0 en passant au prochain sous tableau à tester, on se contente d'enlever la valeur du premier élément du sous tableau précédent, ce qui nous permet de commencer nos tests avec un sous tableau contenant déjà des éléments (que nous allions ajouter de toute façon).

```python
INF = 10**10

n = int(input())
values = list(map(int, input().split()))
b = int(input())

min_len = INF
cur_sum = 0
iright = 0
for idx in range(n):
    while iright < n and cur_sum < b:
        cur_sum += values[iright]
        iright += 1
    if cur_sum == b:
        min_len = min(min_len, iright - idx)
    cur_sum -= values[idx]

if min_len == INF:
    print(-1)
```

```
20 else:
21     print(min_len)
```

Complexité Le sous tableau qui est mis à jour en ajoutant/enlevant des éléments est ce qu'on appelle une **fenêtre glissante**. La boucle imbriquée dans le code pourrait vous faire penser à première vue à une complexité en temps quadratique, mais si l'on réfléchit à la manière dont cette boucle fonctionne dans sa totalité (on parle alors d'**analyse amortie**), on remarque qu'elle ne prend qu'un temps linéaire, car elle va dans le pire des cas parcourir tous les éléments du tableau une seule fois (on ne revient jamais en arrière, on ne fait qu'avancer !).

4 Capitalisme Interplanétaire

4.1 Énoncé

Suite à la pénurie d'hypercarburant de 8153, le conseil suprême de la corporation a voté des coupes de budget. Pour faire toujours plus d'économies les différents vaisseaux de la flotte vont devoir subir un emploi du temps bien plus chargé.

La corporation décide d'assigner des ordres de missions à ses équipages, chaque ordre de mission est constitué d'un emploi du temps d'une durée de `d` jours. Le `i`$^{\text{ème}}$ jour, l'équipage doit remplir un objectif qui ne peut être exécuté que sur un certain type de planète `mission[i]`. Pour rappel, les planètes du système ont été catégorisées en `t` types différents, numérotés de `0` à `t-1`. À noter qu'il y a au moins une planète de chaque type dans le système. Aussi, grâce à OpenSpaceMap il est facile de lister le type de chacune des `n` planètes et leurs coordonnées précises.

La corporation s'est rendu compte que les équipages profitaient de leurs missions pour aller voir leurs proches ou se dorer la pilule sur ZT637, et engendrent des surcouts à cause de leurs détours. Elle a donc besoin d'un moyen de calculer la distance minimale à parcourir lors d'une mission pour imposer des budgets réalistes aux équipages. Elle en profite aussi pour mieux répartir ses ordres de missions, donc vous pouvez supposer qu'une mission commence sur n'importe la planète qui arrange le plus la corporation.

Entrée

L'entrée contiendra :

— Sur la première ligne, un entier : `t`, nombre de types de planètes distincts.

— Sur la ligne suivante, un entier : `n`, le nombre de planètes dans le système.

— Sur les lignes suivantes, une liste de `n` éléments : `planetes`, les planètes du système. Une ligne par élément de la liste : séparés par des espaces, un entier `x` (abscisse de la planète), un entier `y` (ordonnée de la planète), et un entier `k` (type de planète).

— Sur la ligne suivante, un entier : `d`, la durée de la mission à traiter.

— Sur la ligne suivante, une liste de `d` entiers séparés par des espaces : `mission`, pour chaque jour de la mission, le type de planète à visiter.

Sortie

Affichez un entier : la distance minimale à parcourir pour remplir la mission décrite. Les distances devront être calculées en utilisant la distance de Manhattan.

Contraintes

— $1 \leq \texttt{t} \leq 500$

— $0 \leq \texttt{n} \leq 500$

— $1 \leq \texttt{d} \leq 500$

— $0 \leq \texttt{mission[i]} \leq 500$

— $0 \leq \texttt{x} \leq 1\ 000\ 000$

— $0 \leq \texttt{y} \leq 1\ 000\ 000$

— $0 \leq \texttt{k} < t$

4.2 Solution

La solution attendue pour cet exercice est basée sur la programmation dynamique. Plus précisément, on va calculer pour chaque jour i et planète k la quantité de carburant minimale nécessaire pour arriver à la planète k le $i^{\text{ème}}$ jour (tout en respectant l'ordre de mission les jours qui précèdent), appelons cette quantité $Q_{i,k}$.

On remarque que $Q_{i,k}$ peut être calculé à partir des quantités de carburant nécessaires pour arriver sur chaque planète le jour $i-1$: pour cela il faut calculer la distance totale que l'on aurait parcouru pour arriver à la planète k depuis n'importe quelle planète k', et de garder le minimum. Si on était à la planète k' le jour $i-1$, on aura parcouru $d(k,k')$ (la distance entre k et k') en plus et donc au total $Q_{i-1,k'} + d(k,k')$. Il faut aussi ne pas oublier de ne s'intéresser qu'aux planètes k' telles que le type de k' soit bien celui ordonné pour le jour $i-1$. On trouve ainsi la formule suivante :

$$\begin{aligned} Q_{i,k} &= \min_{k'=0}^{n-1} \left(\delta_{t_{k'}=\texttt{mission[i-1]}} \cdot Q_{i-1,k'} \cdot d(p_{k'}, p_k)\right) \\ &= \min\left(\delta_{t_0=\texttt{mission[i-1]}} \cdot Q_{i-1,0} \cdot d(p_0,p_k),\ \ldots,\ \delta_{t_{n-1}=\texttt{mission[i-1]}} \cdot Q_{i-1,n-1} \cdot d(p_{n-1},p_k)\right) \end{aligned}$$

où $\delta_{t_{k'}=\texttt{mission[i-1]}}$ vaut 1 si la l'ordre de mission demande à être sur une planète du même type que k' le $(i-1)^{\text{ème}}$ jour, et 0 sinon.

En utilisant du Python compact on peut recopier cette formule presque telle quelle :

```python
# Lecture de l'entrée
t = int(input())
n = int(input())
planet_pos = []
planet_kind = []

for k in range(n):
    x, y, kind = map(int, input().split())
    planet_pos.append((x, y))
    planet_kind.append(kind)

d = int(input())
objectives = list(map(int, input().split()))


def dist(source, target):
    """Calcule la distance entre deux planètes"""
    x1, y1 = planet_pos[source]
    x2, y2 = planet_pos[target]
    return abs(x1 - x2) + abs(y1 - y2)


# Initialisation du problème: ça coûte 0 de commencer sur n'importe laquelle
# des planètes.
costs = {
    planet: 0
    for planet, kind in enumerate(planet_kind)
    if kind == objectives[0]
}

# Calcul des coûts envisageables Q_{i,k} pour chaque jour
for target_kind in objectives[1:]:
    costs = {
        # Calcul de la distance à parcourir pour arriver à la planète
        # `target_planet`: on prend le minimum des distances qu'on aurait
        # parcouru en partant de chaque planète possible le jour précédent.
        target_planet: min(
            prev_cost + dist(source_planet, target_planet)
            for source_planet, prev_cost in costs.items()
```

```
        )
        # On calcule la distance à parcourir pour chaque planète qui a le type
        # demandé par l'ordre de mission.
        for target_planet, kind in enumerate(planet_kind)
        if kind == target_kind
    }

# Finalement, on retourne la distance minimale pour accéder à une planète
# le dernier jour.
print(min(costs.values()))
```

4.3 Complexité

temps Dans le code qui précède on trouve trois boucles for imbriquées, la première (l. 32) compte au maximum d itérations et les deux autres (l. 39, l. 43) comptent toutes deux au maximum n itérations. Ainsi on obtient une complexité asymptotique en $O\left(d \cdot n^2\right)$. Cela signifie que le programme n'exécutera jamais plus que de l'ordre de $d \times n^2 \leq 500 \times 500^2 = 125\ 000\ 000$ opérations. Sur un ordinateur récent, cadencé à plusieurs gigahertz, cent millions d'opérations peuvent être exécutées en temps raisonnable !

mémoire À tout instant de l'algorithme, en plus de l'entrée, on ne conserve rien d'autre de couteux en mémoire que le dictionnaire `cost`. Il garde en mémoire la distance d'accès à chaque planète candidate pour remplir l'ordre de mission, et donc jamais plus de 500 éléments, ce qui est très faible. Plus formellement, on peut dire que la complexité en mémoire de cet algorithme est en $O(n)$.

4.4 À propos de Dijkstra

À première lecture il est raisonnable d'envisager de résoudre cet exercice en utilisant un algorithme de Dijkstra. Pour cela on pouvait considérer le graphe dont les noeuds représentent chacun une planète à un jour donné (soit de l'ordre de $n \times d$ noeuds), et des arêtes reliant chaque paire de noeuds entre deux jours consécutifs, avec pour poids la distance entre les deux planètes représentées (soit de l'ordre de $d \times n^2$ arêtes).

Bien qu'il n'était pas nécessaire de représenter explicitement le graphe décrit ci-dessus en mémoire, l'algorithme de Dijkstra a quand même une complexité mémoire asymptotique linéaire en le nombre d'arêtes données en entrée. Cela signifie qu'il serait nécessaire de consommer de l'ordre de $d \times n^2 \approx 10^6$ unités de mémoire. Si on regarde de près l'algorithme de Dijkstra, il faudrait plus de mémoire que les 10 Mo autorisés pour faire tenir cette implémentation en mémoire.

... CEPEEENDANT !!! ...

En bidouillant un peu l'ordre utilisé pour comparer les éléments de la file de priorité dans l'algo de Dijkstra, on peut avoir un algorithme similaire à celui précédent. Il suffit de sortir en priorité les éléments de la file qui ont le jour le plus faible, ainsi on va les traiter jour par jour comme dans l'algorithme dynamique. L'algorithme qui en résulte n'est plus vraiment l'algorithme de Dijkstra mais sera toujours correct pour les mêmes raisons que l'algorithme dynamique. Pratique pour adapter son code en ne changeant qu'une ligne si on avait implémenté Dijkstra sans anticiper le problème de mémoire !

Voici un exemple de solution alternative qui est accepté par le correcteur :

```
import heapq

# Lecture de l'entrée
t = int(input())
n = int(input())
planet_pos = []
planets_kind = [[] for _ in range(t)]
```

```python

for k in range(n):
    x, y, kind = map(int, input().split())
    planets_kind[kind].append(len(planet_pos))
    planet_pos.append((x, y))

d = int(input())
objectives = list(map(int, input().split()))


def dist(source, target):
    """Calcule la distance entre deux planètes"""
    x1, y1 = planet_pos[source]
    x2, y2 = planet_pos[target]
    return abs(x1 - x2) + abs(y1 - y2)


# Initialise Dijkstra à partir de tous les points du jour 1
distances = [float("inf") for _ in range(n)]
jours = [0 for _ in range(n)]
heap = []

for s in planets_kind[objectives[0]]:
    heapq.heappush(heap, (1, 0, s))

# Algorithme inspiré de Dijkstra
while True:
    day, dist_s, s = heapq.heappop(heap)

    if day == d:
        print(dist_s)
        break

    if dist_s <= distances[s]:
        for t in planets_kind[objectives[day]]:
            dist_t = dist_s + dist(s, t)
            if jours[t] <= day or distances[t] > dist_t:
                jours[t] = day + 1
                distances[t] = dist_t
                heapq.heappush(heap, (day + 1, dist_t, t))
```

5 La force contre la flotte

5.1 Énoncé

Joseph Marchand, dernier jedi de Prologin, se prépare pour la plus grande bataille qu'il ait jamais mené, il a un plan mais a besoin de votre aide.

Il va affronter aux côtés des rebelles une flotte de vaisseaux de l'empire. Sa stratégie est d'immobiliser les vaisseaux pour que les rebelles puissent aller attaquer leurs points faibles.

La flotte est répartie sur un plan 2D. Pour immobiliser un vaisseau il faut couvrir toute sa surface de ligne de force. Joseph Marchand n'est pas certain de pouvoir contenir tous les vaisseaux, il vous demande donc à vous, ingénieur · e algorithmique de la rébellion, de calculer le nombre minimal de lignes de force à déployer pour recouvrir toute la flotte.

Les lignes de force peuvent être uniquement verticales ou horizontales, mais jamais en diagonales. On peut profiter que deux vaisseaux soient côte à côte pour les couvrir par une même ligne de force mais si du vide les séparent c'est impossible, il faudra deux lignes de force distinctes.

La flotte se présente sous une carte à deux dimension de largeur l et de hauteur h, où les N vaisseaux sont rectangulaires et représentés par deux points, un en haut à gauche et l'autre en bas à droite, À noter que les vaisseaux peuvent se superposer, et on notera S la surface prise par l'ensemble des vaisseaux de la flotte.

Entrée

L'entrée contiendra :

— Sur la première ligne, un entier : l, la largeur de la zone.

— Sur la ligne suivante, un entier : h, la hauteur de la zone.

— Sur la ligne suivante, un entier : N, le nombre de vaisseaux dans la flotte.

— Sur les lignes suivantes, une liste de N éléments : $flotte$, une liste de vaisseaux. Une ligne par élément de la liste : séparés par des espaces, un entier x (abscisse du coin haut-gauche), un entier y (ordonnée du coin haut-gauche), un entier u (abscisse du coin bas-droit), et un entier v (ordonnée du coin bas-droit).

Sortie

Afficher le nombre minimal de lignes de force à déployer pour couvrir toute la flotte.

Contraintes

— $1 \leq l \leq 1000000$

— $1 \leq h \leq 1000000$

— $1 \leq N \leq 1000000$

— $1 \leq S \leq 10000$

— $0 \leq x \leq l$

— $0 \leq y \leq h$

— $x < u \leq l$

— $y < v \leq h$

5.2 Solution

Dans ce problème il y a plusieurs parties, mais le principal enjeu est d'arriver à conceptualiser le problème de sorte à pouvoir le résoudre.

Ici il faut comprendre que le problème revient à trouver la taille d'une couverture minimale par les sommets sur un graphe biparti.

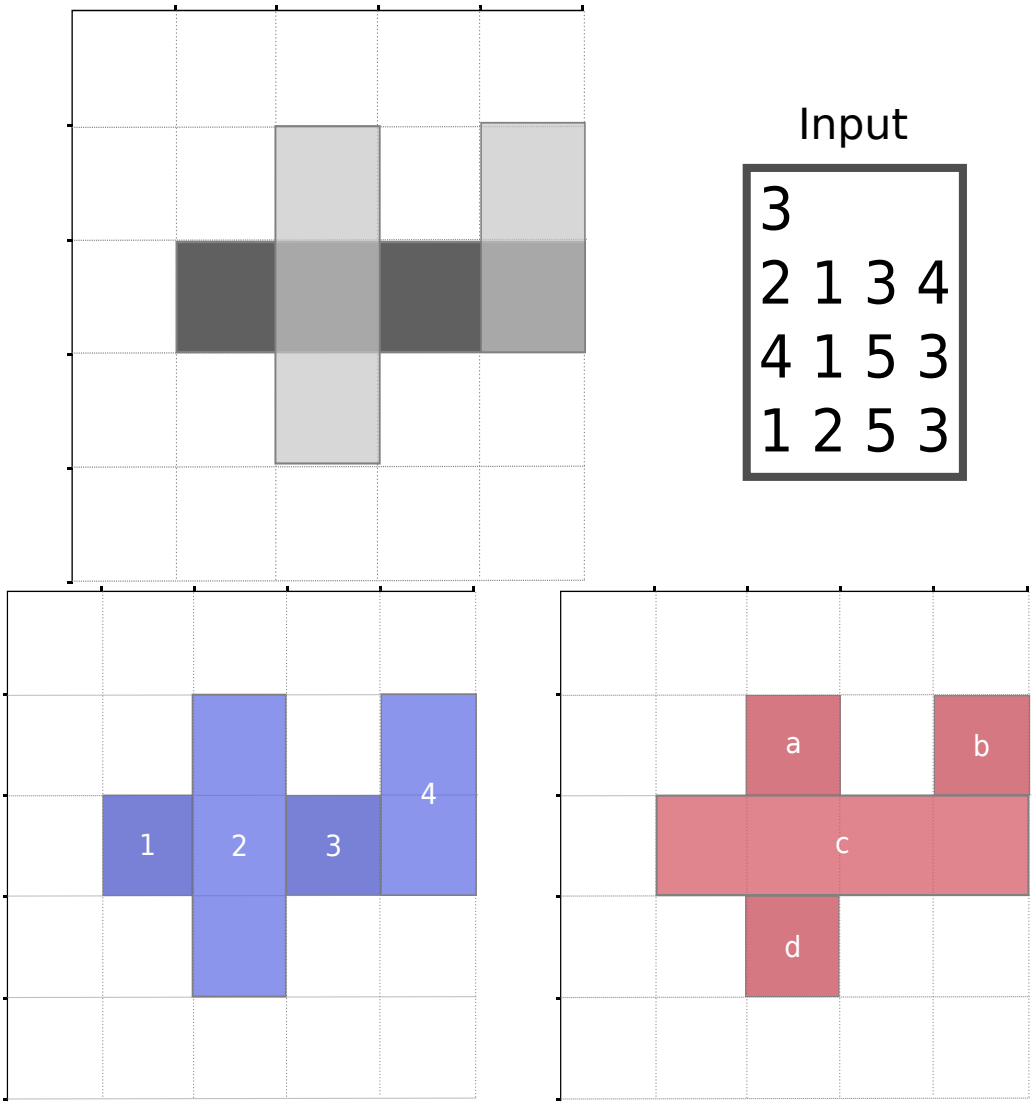
Comment construire le graphe ?

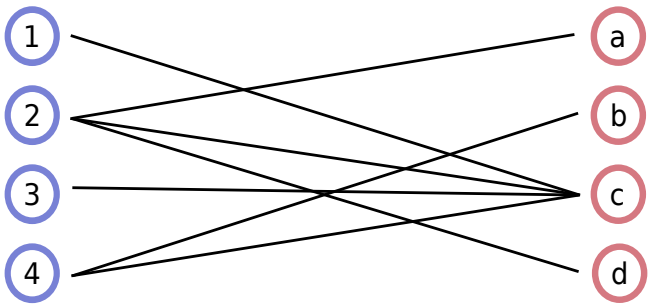
On va construire un graphe avec pour sommets les tirs horizontaux et verticaux les plus longs qu'on puisse faire.

Il faut déjà arriver à calculer quelles sont ces lignes. Ici vient alors le fait que les vaisseaux puissent se superposer. Il y a plusieurs façons de procéder mais avec la limitation sur la surface totale prise par la flotte, on peut se contenter de découper tous les vaisseaux en petit carré de taille 1×1 puis de supprimer les carrés redondants et enfin de les trier pour former les plus longues lignes horizontales et verticales. La complexité de cette étape est $\mathcal{O}(S \log S)$.

Ensuite on ajoute les arêtes ainsi : deux sommets u et v sont liés si les tirs auxquels ils correspondent se croisent. On peut voir chaque arête comme la représentation d'une case 1×1 des vaisseaux, aussi comme chacune des cases doit être couverte par une ligne de force.

Si l'arête (u, v) existe, u ou v doivent faire partie des lignes de force misent en place (la case doit être recouverte). L'enjeu est donc bien de trouver la taille d'une couverture minimale par les sommets (minimal vertex cover en anglais) : chacune des arêtes du graphe doit être couverte par un de ces deux sommets. Ce problème est en général NP-complet (soit à priori pas solvable en temps polynomial) sauf pour les graphes bipartis. Dans notre cas le graphe est bien biparti car les lignes peuvent être divisées entre horizontales et verticales et deux lignes horizontales (ou deux verticales) ne se croisent pas. Pour mieux comprendre la construction du graphe, je vous invite à regarder les dessins d'exemples donnés ci-dessous.





La ligne c est liée à toutes les lignes verticales car c les croisent toutes.

Comment trouver la taille de la couverture minimale ?

Comme mentionné plus haut pour un graphe quelconque le problème d'une couverture des sommets est NP-complet mais dans le cas précis des graphes bipartis le Theoreme de Köenig (1931) nous énonce que la taille de la couverture minimale est égale à la taille du couplage maximal. Or la taille du couplage maximal peut être calculée par l'agorithme de Ford-Fulkerson qui pour tous de sommets x cherche un chemin augmentant partant de x et met à jour le couplage. Pour un graphe avec V sommets et E arêtes la complexité de cet algorithme est $\mathcal{O}(V \times E)$ ce qui dans notre cas correspond à du $\mathcal{O}(S^2)$.

Cela donne le code suivant comme possible implémentation :

```
# Largeur de la carte
max_x = int(input())
# Hauteur de la carte
max_y = int(input())
# Nombre de vaisseaux
N = int(input())


# STRUCTURE
class Block:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        if not isinstance(other, type(self)):
            return NotImplemented
        return self.x == other.x and self.y == other.y

class Shot:
    def __init__(self,x=0,y=0,ex=0,ey=0):
        self.start_x = x
        self.start_y = y
        self.end_x = ex
        self.end_y = ey

def intersect(a,b):
    if a.start_x >= b.end_x or b.start_x >= a.end_x:
        return False
    if a.start_y >= b.end_y or b.start_y >= a.end_y:
```

```
34          return False
35      return True
36
37  blocks = set()
38  # Traitement de l'entrée
39  for i in range(N):
40      x,y,ex,ey = map(int,input().split())
41      for i in range(x, ex):
42          for j in range(y, ey):
43              blocks.add(Block(i,j))
44
45  blocks = list(blocks)
46  # Calcul de toutes les lignes de force maximale possibles (Shots)
47  shot_list = []
48  i = 0
49  blocks.sort(key=lambda s: (s.x, s.y))
50  while i < len(blocks):
51      x = blocks[i].x
52      y_start = blocks[i].y
53      i += 1
54      while i < len(blocks) and x == blocks[i].x:
55          if blocks[i-1].y == blocks[i].y -1 :
56              pass
57          else :
58              # Création d'une nouvelle ligne
59              shot_list.append(Shot(x,y_start,x+1,blocks[i-1].y+1))
60              # Enregistre le début de cette nouvelle ligne
61              y_start = blocks[i].y
62          i += 1
63      # Ajout de la dernière ligne
64      shot_list.append(Shot(x,y_start,x+1,blocks[i-1].y+1))
65
66  i = 0
67  blocks.sort(key=lambda s: (s.y, s.x))
68  while i < len(blocks) :
69      y = blocks[i].y
70      x_start = blocks[i].x
71      i +=1
72      while i < len(blocks) and y == blocks[i].y:
73          if blocks[i-1].x == blocks[i].x -1 :
74              pass
75          else :
76              # Création d'une nouvelle ligne
77              shot_list.append(Shot(x_start,y,blocks[i-1].x+1,y+1))
78              # Enregistre le début de cette nouvelle ligne
79              x_start = blocks[i].x
80          i += 1
81      # Ajout de la dernière ligne
82      shot_list.append(Shot(x_start,y,blocks[i-1].x+1,y+1))
83
84  # Calcul de la matrice d'adjacence du graphe biparti
85  n = len(shot_list)
86  adjacency = [[] for _ in range(n)]
```

```python
for i in range(n):
    for j in range(i+1,n):
        if intersect(shot_list[i],shot_list[j]) and i not in adjacency[j]:
            adjacency[i].append(j)
            adjacency[j].append(i)

# Algorithme de calcul du couplage maximal
def augment(u,adjacency,seen,match):
    for v in adjacency[u]:
        if not seen[v]:
            seen[v] = True
            if match[v] is None or augment(match[v],adjacency,seen,match):
                match[v] = u
                return True
    return False

match = [None]*n
for u in range(n):
    augment(u, adjacency, [False]*n, match)

res = 0
for u in range(n):
    if match[u] is not None :
        res+=1

# res contient le nombre de sommets pris, la taille du couplage est le nombre
# d'arêtes donc res/2
print(res//2)
```

*
**

Félicitations à tous les participantes et 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.