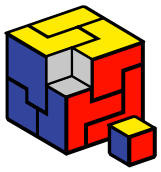
Prologin 2015



Concours National d'Informatique

Questionnaire de sélection

Correction des questions d'algorithmique

Lê Thành Dũng Nguyễn Jill-Jênn Vie [*]

27 janvier 2015

Table des matières

[*] Pour l'équipe des correcteurs 2015 : Kaci Adjou, Alexandre Bonnetain, ~~Simon~~ Sacha Delanoue, Paul Hervot, Stéphane Lefebvre, Marin Hannache, Stéphane Henriot, Lê Thành Dũng Nguyễn, Antoine Pietri, Yvan Sraka, Jill-Jênn Vie.

1 Triangles

1.1 Énoncé

On veut savoir le nombre de points requis pour former un triangle de N lignes. Il y a un point sur la première ligne et chaque ligne est constituée d'un point de plus que la précédente. Écrivez une fonction prenant en argument le nombre de lignes du triangle et renvoyant le nombre de points nécessaires pour former ce triangle.

1.2 Solution naïve

Comme on a un point sur la première ligne, deux points sur la deuxième ligne, etc. jusqu'à n points sur la n-ième ligne, la réponse attendue est le résultat du calcul de $1+2+\ldots+n$, qu'on peut effectuer via une boucle :

Listing 1 – Une solution de l'exercice 1 en Python 3

```
1 def triangles(n):
2     nb_points = 0
3     for i in range(1, n + 1):
4         nb_points += i
5     return nb_points
6
7 print(triangles(int(input())))
```

ou bien avec la fonction standard `sum`, qui détermine la somme d'une liste :

Listing 2 – Une autre solution de l'exercice 1 en Python 3

```
1 n = int(input())
2 print(sum(range(1, n + 1)))
```

Notez, pour les puristes, que cette approche a une complexité non linéaire mais exponentielle car la taille des données est ici $\log n$ (et non n, car l'entrée n'est composée que d'un entier sur $\log n$ bits).

1.3 Solution mathématique

Ici, on utilise la formule [1] : $1+\ldots+n = \dfrac{n(n+1)}{2}$.

Démonstration. Si S est la somme à calculer, alors si on l'écrit deux fois, une fois à l'endroit et une fois à l'envers :

$$\begin{array}{ccccccccc} & 1 & + & 2 & + & \ldots & + & n \\ + & n & + & n-1 & + & \ldots & + & 1 \end{array}$$

alors en regroupant les termes en colonnes, on obtient une somme de n termes tous égaux à $n+1$, d'où $2S = n(n+1)$ et $S = \frac{n(n+1)}{2}$. □

En Python 3, cela donne le code suivant :

Listing 3 – Une autre solution de l'exercice 1 en Python 3

```
1 def triangles(n):
2     return (n * (n + 1)) / 2
3
4 print(triangles(int(input())))
```

1. La légende raconte qu'elle aurait été découverte par Carl Friedrich Gauss à l'école primaire, alors que l'instituteur avait demandé de calculer $1+2+\ldots+100$.

2 Syracuse

2.1 Énoncé

Dans cet exercice, il vous est demandé d'afficher le k-ième terme d'une suite de Syracuse. À partir d'un nombre x strictement positif, on obtient une suite de Syracuse comme suit :

— on part de x comme terme numéro zéro ;
— ensuite, un terme est obtenu à partir du précédent :
  – soit en divisant par 2, s'il était pair ;
  – soit en multipliant par 3 puis en ajoutant 1, s'il était impair.

2.2 Solution

L'énoncé nous explique comment obtenir, à partir d'un terme, le suivant. Ainsi, on peut partir du terme n° 0, égal à x, puis calculer successivement le terme n° 1, le terme n° 2, etc. jusqu'au k-ième. Pour cela, on répète k fois les mêmes actions sur une variable, ce que l'on peut faire avec une boucle `for`.

Le code suivant réalise cela :

Listing 4 – Une solution de l'exercice 2 en Python 3

```
1  def syracuse(x, k):
2      u = x
3      for i in range(k):
4          if u % 2 == 0:
5              u = u / 2
6          else:
7              u = 3 * u + 1
8      return u
9
10 x = int(input())
11 k = int(input())
12 print(syracuse(x, k))
```

2.3 Solution d'un candidat

L'usage de fonctions d'ordre supérieur permet d'exprimer l'idée de faire k itérations successives de la même transformation sans avoir recours explicitement à une boucle, voyons-en une illustration avec le code suivant en Haskell :

Listing 5 – Une solution alambiquée de l'exercice 2 en Haskell par Rafbill

```
1  {-# LANGUAGE ViewPatterns #-}
2
3  snext x
4    | x `mod` 2 == 0 = x `div` 2
5    | otherwise = 3 * x + 1
6
7  main = interact $ \(fmap read . words -> [u, k]) ->
8                  show $ iterate snext u !! k
```

3 Rond-point

3.1 Énoncé

Pour des raisons absurdes, Joseph Marchand doit prendre la première sortie à droite à chaque fois que sa voiture arrive sur un rond-point.

On vous donne les coordonnées du rond-point, et une liste de coordonnées représentant les différentes destinations des sorties du rond-point. La voiture arrive sur le rond-point depuis la première de ces destinations dans la liste. Écrivez une fonction qui renvoie les coordonnées de la destination de la sortie qui correspond, du point de vue de Joseph Marchand, à la première sortie du rond-point.

3.2 Précisions

L'énoncé de cet exercice semble avoir causé beaucoup de confusion parmi les candidats. Ainsi, certains ont cru que la configuration du rond-point était fixée à celle décrite par le dessin ; d'autres n'ont pas compris ce que signifiait « la première à droite ».

En fait, il fallait comprendre que le problème posé concernait une figure géométrique en « étoile », qui contenait :

— un point central R (le rond-point), un point d'entrée E, et des points de sortie P_i ; ces points sont donnés en entrée et varient d'un test à l'autre ;

— les segments $[RE]$ et $[RP_i]$ (où i est compris entre 1 et le nombre de sorties).

Il fallait trouver le point P_i tel que le segment de sortie $[RP_i]$ fût le plus proche à droite du segment d'entrée $[RE]$, c'est-à-dire qu'il fallait minimiser l'angle orienté $\widehat{ERP_i}$.

3.3 Solution élémentaire

Ceci étant un problème de géométrie portant sur des angles, on pouvait s'attendre à ce qu'il soit nécessaire de recourir à la trigonométrie pour calculer des angles. Il n'en est rien : la solution présentée ici ne demande que des connaissances de niveau collège (pente d'une droite).

Une première étape pouvait consister à répartir tous les points parmi quatre listes selon leur position par rapport au rond-point (à gauche, en bas, à droite ou au-dessus, cf. Figure 1). Notez qu'il peut y avoir au maximum un seul point exactement au-dessus du rond-point car deux routes ne se chevauchent jamais. Idem pour au-dessous.

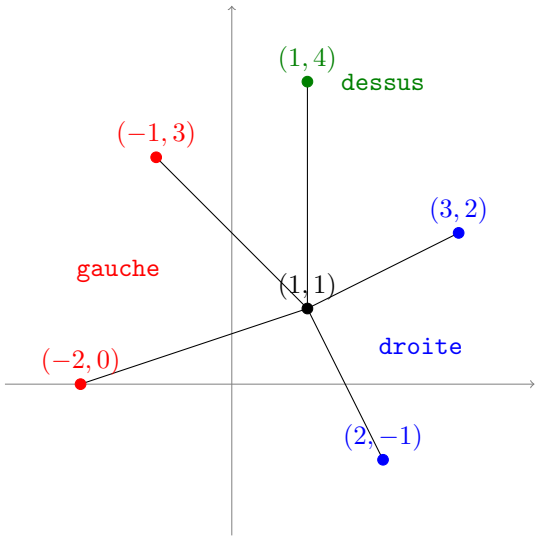


FIGURE 1 – Un exemple de jeu de routes, réparties dans 4 classes. La classe `dessous` est vide.

On remarque alors que pour les énumérer dans l'ordre inverse des aiguilles d'une montre, on peut les considérer par pente croissante au sein d'une même classe. Ainsi, dans la classe `droite` de la Figure 1 ci-contre, la pente de la droite passant par le rond-point et $(3, 2)$ est $\frac{2-1}{3-1} = \frac{1}{2}$ tandis que la pente de la droite passant par le rond-point et $(2, -1)$ est $\frac{-1-1}{2-1} = -2$. Donc $(2, -1)$ se situe avant $(3, 2)$ dans l'ordre inverse des aiguilles d'une montre. Un raisonnement similaire s'applique à la classe `gauche`.

Une fois les classes `gauche` et `droite` ainsi triées (les deux autres n'ont pas besoin d'être triées car elles contiennent au plus un seul élément), on obtient tous les points dans l'ordre en considérant les points à gauche, puis au-dessous, puis à droite, puis au-dessus. Il ne reste plus qu'à déterminer où se trouve l'entrée et renvoyer le sommet suivant (donc plus 1 modulo $n+1$ au cas où l'entrée se trouverait à la fin de la liste de $n+1$ points).

Listing 6 – Une solution de l'exercice 3 en Python

```
def rondpoint(x0, y0, n, coords):
    xd, yd = coords[0] # Départ
    gauche = []
    droite = []
    dessus = []
    dessous = []
    for x, y in coords:
        if x < x0:
            gauche.append((x, y))
        elif x > x0:
            droite.append((x, y))
        elif y > y0:
            dessus.append((x, y))
        else:
            dessous.append((x, y))
    gauche.sort(key=lambda (x, y): float(y - y0) / (x - x0)) # Tri par pente croissante
    droite.sort(key=lambda (x, y): float(y - y0) / (x - x0))
    points_ordonnes = gauche + dessous + droite + dessus
    for i in range(n):
        if points_ordonnes[i] == (xd, yd): # On a trouvé l'entrée
            x, y = points_ordonnes[(i + 1) % (n + 1)]
            return '%d %d' % (x, y)

x0, y0 = map(int, raw_input().split())
n = int(raw_input())
coords = []
for _ in range(n):
    coords.append(map(int, raw_input().split()))
print(rondpoint(x0, y0, n, coords))
```

3.4 Solution trigonométrique

Une autre idée naturelle, lorsqu'on connaît les coordonnées polaires en géométrie, est d'essayer de calculer les angles polaires des points de sortie relativement à l'axe $[RE)$ (R : rond-point ; E : entrée), et de choisir la sortie avec l'angle polaire le plus petit (en prenant les angles dans $[0, 2\pi[$. Pour cela, la plupart des langages de programmation fournissent la fonction `atan2` : pour un point P de coordonnées (x, y), `atan2(y,x)` renvoie l'angle polaire de P dans le repère usuel (où l'axe $[Ox)$ est choisi comme axe polaire). Voyons comment le code procède :

Listing 7 – Une autre solution de l'exercice 2 en Lua

```lua
function read_int_list(s)
    -- Lit une liste d'entiers dans une chaîne de caractères
    local t = {}
    for number in string.gmatch(s, "-?%d+") do
        table.insert(t, number)
    end
    return t
end

function angle_avec(point, rond_point)
    -- Détermine l'angle avec l'axe des abscisses
    return math.atan2(point[2] - rond_point[2], point[1] - rond_point[1])
end

rond_point = read_int_list(io.read())
n = tonumber(io.read())
depart = read_int_list(io.read())

angle_min = 2 * math.pi
angle_depart = angle_avec(depart, rond_point)
for i = 1, n - 1 do
    point = read_int_list(io.read())
    angle_point = (angle_avec(point, rond_point) - angle_depart) % (2 * math.pi)
    if angle_point < angle_min then
        angle_min = angle_point
        reponse = point
    end
end
print(string.format('%d %d', reponse[1], reponse[2]))
```

Cette solution, qui calcule le résultat d'une fonction trigonométrique, a le défaut de faire intervenir de façon incontournable les nombres à virgule flottante. (Dans la solution précédente, les pentes étaient des nombres rationnels et auraient pu être représentées par des couples d'entiers numérateur/dénominateur.) En conséquence, les calculs deviennent plus lents et surtout approximatifs, alors qu'on peut résoudre cet exercice en ne faisant que des calculs exacts (arithmétique entière), comme le montre bien la solution suivante.

3.5 Solution astucieuse d'un candidat

Voici une solution d'un candidat calculant le déterminant de 2 vecteurs :

Listing 8 – Une solution de l'exercice 3 en C++ par @uguste

```
#include <cstdio>
#include <cmath>

int main()
{
   int n, x, y, x1, y1, xd, yd;
   scanf("%d%d%d%d%d%d%d", &x, &y, &n, &x1, &y1, &xd, &yd);
   for (int i = 0; i < n - 2; ++i)
   {
      int xi, yi;
      scanf("%d%d", &xi, &yi);
      int a = (x1 - x) * (yi - y) - (y1 - y) * (xi - x);
      int b = (xd - x) * (yi - y) - (yd - y) * (xi - x);
      int c = (x1 - x) * (yd - y) - (y1 - y) * (xd - x);
      if ((a >= 0 && (c < 0 || b < 0)) || (a < 0 && c < 0 && b < 0))
         xd = xi, yd = yi;
   }
   printf("%d %d\n", xd, yd);
}
```

Ce programme présente deux bonnes propriétés :
- il ne fait intervenir que de l'arithmétique sur des entiers ;
- il lit ses entrées une à la fois, ne retenant en mémoire qu'un petit nombre de variables entières (ainsi il fonctionne en espace constant).

Au début d'un tour de boucle, `xd` et `yd` contiennent les coordonnées du point de sortie le plus proche à droite parmi ceux déjà rencontrés. On va leur affecter les coordonnées du nouveau point si et seulement si celui-ci est encore plus proche, c'est-à-dire s'il se trouve « entre » le vecteur (rond-point, entrée) et le vecteur (rond-point, (`xd`, `yd`)).

Pour tester cette dernière condition, on calcule les déterminants :

$$a = \det\left(\overrightarrow{RE}, \overrightarrow{RP_i}\right) \qquad b = \det\left(\overrightarrow{RD}, \overrightarrow{RP_i}\right) \qquad c = \det\left(\overrightarrow{RE}, \overrightarrow{RD}\right)$$

où on a repris les notations du 3.2 :
- R est l'emplacement du rond-point, de coordonnées (x, y) ;
- E est le point d'entrée, de coordonnées (x_1, y_1) ;
- D est l'ancien candidat de point destination, de coordonnées (x_d, y_d) ;
- P_i est le point qu'on vient de lire, de coordonnées (x_i, y_i).

On rappelle que $\det(\vec{u}, \vec{v})$ est positif lorsque $\widehat{(\vec{u}, \vec{v})}$ est entre 0 et π ($\vec{v}$ est « dans la moitié gauche » relativement à $\vec{u}$), et négatif lorsque l'angle est entre $-\pi$ et 0. Un dessin (laissé en exercice au lecteur) permet de voir que la condition dans le `if` correspond bien à « $\overrightarrow{RP_i}$ est plus à droite de $\overrightarrow{RE}$ que $\overrightarrow{RD}$ ne l'est ».

4 Expert-itinérant

4.1 Énoncé

Vous disposez d'une carte des rues d'une ville, c'est-à-dire aux données suivantes :
— les différentes rues et les emplacements à leurs extrémités ;
— le temps nécessaire pour aller d'un bout à l'autre de chaque rue. Attention, ce temps dépend du sens du parcours. De plus, si on vous indique le temps pour aller d'un point A à un point B, mais pas de B à A, cela signifie que la rue est à sens unique $A \to B$.

On vous donne un ensemble de requêtes d'itinéraire à traiter urgemment ; trouvez, pour chacune d'entre elles, le temps minimum pour la satisfaire. Évidemment, il sera toujours possible de rejoindre l'arrivée depuis le départ.

4.2 Précisions

Il s'agissait du problème classique de *plus court chemin* dans un *graphe orienté pondéré à poids positifs*. Nous allons expliciter ce qui était demandé dans cet exercice.

La donnée passée en argument était composée des emplacements et des rues ayant :
— un emplacement de départ ;
— un emplacement d'arrivée ;
— un temps de parcours.

Si la rue est à double sens, on considère qu'il y a deux rues, chacune dans un sens, qui peuvent nécessiter un temps de parcours différent [2].

Chaque requête demandait l'*itinéraire de durée minimale* reliant deux emplacements. Un itinéraire de A à B consiste en une suite de rues telle que :
— le départ de la première rue est A ;
— l'arrivée de la dernière rue est B ;
— si deux rues se succèdent dans l'itinéraire, l'arrivée de celle d'avant doit coïncider avec le départ de celle d'après.

L'itinéraire décrit les rues que l'on emprunte dans l'ordre pour se rendre de A à B, et on ne peut passer d'une rue à une autre qu'à un carrefour commun. La durée d'un itinéraire est la somme des temps nécessaires pour parcourir chaque rue.

Le problème consistait donc à considérer le graphe des villes et la liste des requêtes, et renvoyer pour chaque requête le temps de l'itinéraire le plus rapide. Comme le nombre de requêtes pouvait être grand (jusqu'à 40 000), on attendait des candidats que leur algorithme les traite plus efficacement qu'en les résolvant une à une indépendamment.

4.3 Solution classique

Une méthode standard pour résoudre ce problème est d'utiliser l'algorithme de Floyd–Warshall [3], qui permet de calculer les plus courts chemins entre toutes les paires de sommets du graphe (c'est-à-dire, entre deux emplacements quelconques de la carte).

Numérotons les emplacements de la carte de 1 à N. L'algorithme calcule le tableau $\texttt{distance}_k$ où $\texttt{distance}_\texttt{k}[\texttt{i}][\texttt{j}]$ représente la longueur du plus court chemin entre `i` et `j` si l'on ne s'autorise qu'à passer par des emplacements de numéro plus petit que `k`. (Il s'agit d'une méthode de *programmation dynamique*.) Ainsi définie, $\texttt{distance}_1[\texttt{i}][\texttt{j}]$ vaut 0 si $\texttt{i} = \texttt{j}$, $+\infty$ sinon (tous les emplacements sont interdits lorsque $\texttt{k} = 0$). Puis pour un certain $\texttt{k} \in \{1, \ldots, N\}$, deux cas sont possibles :
— si permettre de passer par le sommet `k` ne donne pas de chemin plus court entre `i` et `j`, on a : $\texttt{distance}_{\texttt{k}+1}[\texttt{i}][\texttt{j}] = \texttt{distance}_\texttt{k}[\texttt{i}][\texttt{j}]$

2. Par exemple, une piste cyclable avec un dénivelé.
3. Pour les curieux : cet algorithme se généralise en fait en l'algorithme de Kleene (parfois aussi appelé algorithme de McNaughton–Yamada) qui effectue ce calcul dans une algèbre de Kleene quelconque, ce qui permet notamment de déterminer les langages rationnels reconnus entre des paires d'états d'automates.

— sinon, un chemin optimal est d'aller de `i` à `k`, puis de `k` à `j` (sans repasser par `k`, ce qui constituerait un boucle de durée positive donc ne raccourcirait pas le chemin), donc $\texttt{distance}_{\texttt{k+1}}[\texttt{i}][\texttt{j}] = \texttt{distance}_{\texttt{k}}[\texttt{i}][\texttt{k}] + \texttt{distance}_{\texttt{k}}[\texttt{k}][\texttt{j}]$.

Avec ces notations, la valeur qui nous intéresse serait alors $\texttt{distance}_N[\texttt{i}][\texttt{j}]$ (où N est le nombre d'emplacements), la longueur du plus court chemin entre `i` et `j` si l'on considère tous les emplacements.

Une dernière optimisation de mémoire : pour chaque valeur de $\texttt{k} \in \{1, \dots, N\}$, pour calculer $\texttt{distance}_{\texttt{k+1}}$ on a seulement besoin de connaître $\texttt{distance}_{\texttt{k}}$. Ainsi, il suffit d'allouer un tableau `distance` dont on souhaite que, à l'issue du `k`-ième tour de boucle, $\texttt{distance}[\texttt{i}][\texttt{j}] = \texttt{distance}_{\texttt{k}}[\texttt{i}][\texttt{j}]$ quels que soient $\texttt{i}, \texttt{j} \in \{1, \dots, N\}^2$. Ainsi, la complexité spatiale n'est que de $O(N^2)$, tandis que l'algorithme reste en temps $O(N^3)$. En voici une implémentation en Python :

Listing 9 – Une solution de l'exercice 4 en Python – Algorithme de Floyd–Warshall

```
def expert_itinerant(N, M, R, aretes, requetes):
    distance = [[float('inf')] * N for _ in range(N)]
    for debut, arrivee, temps in aretes:
        distance[debut - 1][arrivee - 1] = temps
    for k in range(N):
        for i in range(N):
            for j in range(N):
                distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
    for debut, arrivee in requetes:
        print(distance[debut - 1][arrivee - 1])

N, M, R = map(int, input().split())
aretes = []
for _ in range(M):
    aretes.append(map(int, input().split()))
requetes = []
for _ in range(R):
    requetes.append(map(int, input().split()))
expert_itinerant(N, M, R, aretes, requetes)
```

4.4 Solution matricielle (niveau expert)

On peut aussi songer à un algorithme de programmation dynamique qui, au lieu de calculer la longueur du plus court chemin entre i et j passant par des sommets plus petits que k, calcule la longueur du plus court chemin *qui utilise au plus ℓ arêtes* notée $D_{ij}^{(\ell)}$. Dans ce cas, on a $D_{ij}^{(0)} = 0$ si $i = j$, $+\infty$ sinon et on obtient une formule de récurrence de la forme suivante, pour tout entier $\ell \geqslant 0$:

$$D_{ij}^{(\ell+1)} = \min_{k=1}^{n} \{D_{ik}^{(\ell)} + A_{kj}\}$$

où on a noté A_{ij} la longueur de la rue entre i et j, avec $A_{ij} = +\infty$ s'il n'existe pas de telle rue.

Cela peut être vu comme un produit de matrices sur le semi anneau $(\mathbb{R} \cup \{+\infty\}, \min, +)$:

$$\text{pour tout } \ell \geqslant 0, \qquad D^{(\ell+1)} = D^{(\ell)} A \qquad \text{soit} \qquad D^{(\ell)} = A^{\ell}.$$

En effet, la définition de la multiplication se généralise pour pouvoir considérer des matrices dont les éléments ne sont pas forcément des nombres, mais simplement des éléments d'un *semi-anneau*[4]. Ici, on travaille encore avec des nombres réels (auxquels nous avons ajouté $+\infty$), mais on les voit comme éléments de *l'algèbre tropicale*[5], dont la loi additive est min et la loi multiplicative est $+$. (En effet, deux structures algébriques différentes peuvent avoir le même ensemble d'éléments.)

4. C'est-à-dire une structure algébrique avec une loi de composition additive et une loi multiplicative, et des éléments neutres, mais pas forcément des inverses. On exige les axiomes usuels : associativité, commutativité, distributivité.

5. Ainsi dénommée pour ses liens avec la géométrie tropicale, elle est aussi appelée « algèbre min-plus ».

Un plus court chemin passant par au plus N arêtes (les poids étant tous positifs), on cherche à calculer $D^{(N)} = A^N$. Cependant, comme les algorithmes de multiplication matricielle rapide que l'on connaît ne marchent que sur des anneaux et non des semi-anneaux, l'algorithme de plus court chemin obtenu utilisant l'exponentiation rapide est en fait plus lent ($O(N^3 \log N)$) que Floyd–Warshall ($O(N^3)$), aussi bien asymptotiquement qu'en pratique. L'existence d'un algorithme en temps $O(n^{3-\varepsilon})$ pour calculer tous les plus courts chemins dans un graphe est un problème ouvert.

Voici comment formaliser tout cela en OCaml, en exploitant le système de modules :

Listing 10 – Une solution de l'exercice 4 en OCaml – Multiplication matricielle

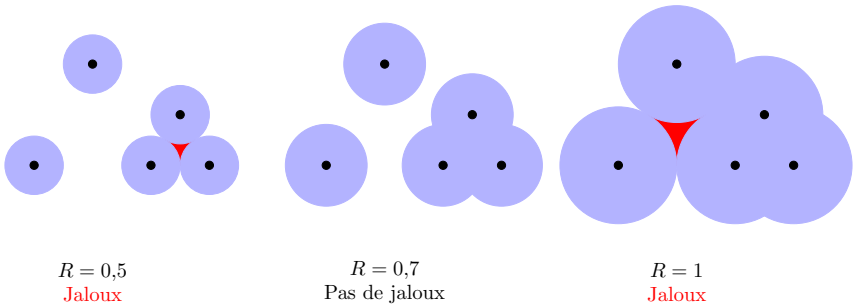
```
(* la signature des semi-anneaux : il faut fournir
   * le type (l'ensemble) des éléments
   * les deux lois de compositions
   * les neutres *)
module type Semiring = sig
    type t
    val (<+>) : t -> t -> t (* loi additive *)
    val (<*>) : t -> t -> t (* loi multiplicative *)
    val zero : t (* neutre pour <+> *)
    val one : t (* neutre pour <*> *)
  end

(* l'anneau Z des entiers relatifs, donné comme exemple
   on voit d'où viennent les noms des opérateurs et constantes *)
module Integers : Semiring = struct
  type t = int
  let (<+>) = ( + )
  let (<*>) = ( * )
  let zero = 0
  let one = 1
end

(* l'algèbre tropicale *)
module Tropical = struct
  type t = Infty | Finite of int
  let (<+>) x y = match x, y with
    | Infty, _ -> y
    | _, Infty -> x
    | Finite x', Finite y' -> Finite (min x' y')
  let (<*>) x y = match x, y with
    | Infty, _ -> Infty
    | _, Infty -> Infty
    | Finite x', Finite y' -> Finite (x' + y')
  let zero = Infty
  let one = Finite 0
end

(* les opérations matricielles sont définies dans un foncteur
   ce qui permet de les paramétrer par le choix du semi-anneau *)
module Matrix (S : Semiring) = struct

  open S

  let zero_matrix dim = Array.make_matrix dim dim zero
  let identity dim =
```

```
46      Array.init dim (fun i -> Array.init dim (fun j -> if i = j then one else zero))
47
48    let prod a b =
49      let n = Array.length a in
50      let c = zero_matrix n in
51      for k = 0 to n - 1 do
52        for i = 0 to n - 1 do
53          for j = 0 to n - 1 do
54            c.(i).(j) <- c.(i).(j) <+> (a.(i).(k) <*> b.(k).(j))
55          done
56        done
57      done;
58      c
59
60    let exp_rapide a n =
61      let dim = Array.length a in
62      let id = identity dim in
63      let rec aux p a n = match n with
64        | 0 -> p
65        | n when n mod 2 == 0 -> aux p (prod a a) (n / 2)
66        | n -> aux (prod p a) (prod a a) (n / 2)
67      in aux id a n
68
69  end
70
71  (* c'est là que commence le code spécifique au problème *)
72
73  type rue = {
74    rue_depart : int;
75    rue_arrivee : int;
76    rue_temps : int;
77  };;
78
79  type requete = {
80    requete_depart : int;
81    requete_arrivee : int;
82  };;
83
84  let expert_itinerant nb_emp nb_rues nb_req rues requetes =
85    let module M = Matrix(Tropical) in
86    let dist = M.identity nb_emp in
87    for i = 0 to nb_rues - 1 do
88      dist.(rues.(i).rue_depart - 1).(rues.(i).rue_arrivee - 1) <-
89        Tropical.Finite(rues.(i).rue_temps)
90    done;
91    let dist = M.exp_rapide dist nb_emp in
92    for i = 0 to nb_req - 1 do
93      match dist
94            .(requetes.(i).requete_depart - 1)
95            .(requetes.(i).requete_arrivee - 1) with
96      | Tropical.Infty -> failwith "êrequte non satisfiable"
97      | Tropical.Finite d -> Printf.printf "%d\n" d
98    done
```

5 Wi-Fi

Ce problème, le plus difficile, se trouvait dans la sélection pour occuper les meilleurs participants. Dans tout ce qui suit, on considère N routeurs, de coordonnées $[P_1, \dots, P_N]$.

Il s'agissait de déterminer un rayon des routeurs à partir duquel il n'y a pas de zones de jaloux. Notez que si l'on considère un rayon croissant, il peut y avoir plusieurs moments où l'on passe d'une présence de jaloux à une absence de jaloux (cf. Figure 2). On veut connaître le dernier de ces moments.



$R = 0{,}5$
Jaloux

$R = 0{,}7$
Pas de jaloux

$R = 1$
Jaloux

FIGURE 2 – Divers exemples de rayons pour une même configuration de routeurs. La réponse pour ce jeu de routeurs est $R = \sqrt{3} \simeq 1,732$.

Intéressons-nous aux positions des zones de jaloux. Première remarque, il faut au moins 3 routeurs pour créer une zone de jaloux. En faisant augmenter le rayon, cette zone va se résorber : à un moment, elle va se condenser en un point. Ce point se retrouve donc touché par la frontière d'au moins 3 disques de routeurs : il est à égale distance de ces routeurs. On cherche donc le rayon d'un cercle circonscrit à (au moins) 3 routeurs. Tous les énumérer serait rédhibitoire ($\binom{N}{3} = O(N^3)$), il faut donc ruser.

Concentrons-nous sur un routeur P_i pour $i \in \{1, \dots, N\}$. Il va participer à la résorption des zones de jaloux les plus proches de lui. Il s'agit donc de savoir de quels points il est le plus proche, les autres zones de jaloux étant résorbées par d'autres routeurs. Il est donc naturel de considérer sa cellule de Voronoi [6], c'est-à-dire l'ensemble des points plus proches de ce routeur P_i que de tout autre routeur :

$$\text{Voronoi}(P_i) = \{M \in \mathbb{R}^2 \text{ tel que pour tout routeur } P_j, \text{ on a } P_iM \leqslant P_jM\}$$

L'algorithme de Fortune permet de calculer toutes les cellules de Voronoi en $O(N \log N)$ où N est le nombre de points considérés. Les frontières des cellules de Voronoi sont des polygones. On peut prouver que tout point est un sommet d'un polygone de Voronoi (dit *sommet de Voronoi*) si et seulement si c'est le centre d'un cercle circonscrit à au moins 3 routeurs, ne contenant aucun autre routeur dans sa surface (susceptible de résorber une zone de jaloux).

Le sommet qui nous intéresse est donc un sommet de Voronoi, mais lequel ? Tous les sommets de Voronoi ne sont pas des points de résorption de zones de jaloux, comme indiqué sur la Figure 3 :

Points non confinés dans un demi-disque
Jaloux

Points confinés dans un demi-disque
Jamais de jaloux

FIGURE 3 – Les centres des cercles ici tracés en rouge, vert et pointillé sont des sommets de Voronoi. À gauche il s'agit d'un point de résorption de zone de jaloux, à droite non.

6. Pour plus d'informations, se reporter aux articles Wikipédia https://fr.wikipedia.org/wiki/Diagramme_de_Voronoï et https://fr.wikipedia.org/wiki/Triangulation_de_Delaunay.

Qu'est-ce qui caractérise donc l'apparition de jaloux ? On considère $Q_1, \ldots, Q_n$ points sur un même cercle de centre C et de rayon R, ordonnés dans le sens des aiguilles d'une montre. Il existe une zone de jaloux si et seulement si pour tout $i \in \{1, \ldots, n\}$, $Q_iQ_{i+1} < 2R$ (avec $Q_{n+1} = Q_1$), c'est-à-dire exactement s'ils ne sont pas confinés dans un demi-disque. Intuitivement, cela correspond à une configuration où les disques des routeurs consécutifs se touchent avant que le sommet de Voronoi n'ait été atteint. Ce test peut se faire par exemple en $O(n \log n)$, coût pour trier les points dans l'ordre croissant.

Algorithme de résolution de l'exercice 5

procédure PortéeMinimale(N, $(P_1, \ldots, P_N)$)
  $\{(C_i, R_i, (p_1, \ldots, p_{n_i})\} \leftarrow$ Voronoi$(N, (P_1, \ldots, P_N))$ ▷ Algorithme de Fortune en $O(N \log N)$
    ▷ où pour tout i, $(p_1, \ldots, p_{n_i})$ sont les routeurs à distance R_i du sommet de Voronoi C_i
  Trier($\{(C_i, R_i, (p_1, \ldots, p_{n_i})\}$) par rayon décroissant
  pour $C, R, (p_1, \ldots, p_n) \in \{(C_i, R_i, (p_1, \ldots, p_{n_i})\}$ **faire**
    Trier($\{p_1, \ldots, p_n\}$) dans le sens des aiguilles d'une montre
    si l'angle $\widehat{p_1Cp_n}$ dépasse 180° **alors** ▷ Les points ne sont pas confinés dans un demi-disque
      renvoyer R ▷ Il y a existence de jaloux jusqu'à ce que la portée de $p_1, \ldots, p_n$ soit R
      ▷ Sinon, on ignore ce sommet de Voronoi et on passe au candidat suivant
  renvoyer 0

Cet algorithme est en $O(N \log N)$. La démonstration est laissée au lecteur.

$\sim$

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.