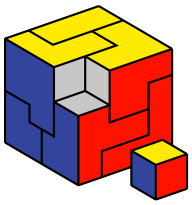
Prologin 2018



Concours National d'Informatique

Correction de la phase de sélection

Correction des questions d'algorithmique

Thibault Allançon, Guillaume Aubian, Arthur Remaud*

18 février 2018

Table des matières

*Pour l'équipe des correcteurs 2018 : Kaci Adjou, Thibault Allançon, Eloi Charpentier, Clément Galland, Paul Guénézan, Baptiste Raabe, Arthur Remaud.

1 Bataille de crêpes

1.1 Énoncé

C'est enfin l'été ! Joseph Marchand arrive sur sa plage préférée pour profiter des vacances en cette belle journée ensoleillée. La plage s'étend sur N mètres de long d'ouest en est. Il peut voir deux marchands de crêpes situés respectivement à M_1 et M_2 mètres depuis l'extrémité ouest de la plage (la position 0 de la plage). Il sait que les bronzeurs sont aussi faineants que lui : un bronzeur situé à x mètres ira toujours chez le marchand le plus proche. On dit que la position x est sous l'influence de ce marchand.

Joseph s'intéresse à la zone d'influence de chaque marchand : le segment formé par l'ensemble des positions (pas nécessairement entières) sous l'influence de chacun des marchands.

Attention : la dernière position de la plage se situe à N mètres du début de la plage.

1.2 Solution

Le sujet nous demande de comparer deux distances, qui représentent les zones d'influence de chaque marchand.

On va commencer par calculer où est le milieu de la plage, pour pouvoir obtenir ensuite l'éloignement par rapport à ce milieu de chaque marchand. Si les marchands sont autant éloignés, on affichera 0, sinon on affichera le marchand le plus proche du milieu, donc avec la plus petite distance par rapport à ce milieu.

Une autre solution possible serait de calculer pour chaque marchand l'étendue de sa zone d'influence, et de les comparer ensuite.

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

```
1  n = 2 * int(input())
2  m1 = 2 * int(input())
3  m2 = 2 * int(input())
4
5  mid = n // 2
6  d1 = abs(m1 - mid)
7  d2 = abs(m2 - mid)
8
9  if d1 == d2:
10     print(0)
11 elif d1 < d2:
12     print(1)
13 else:
14     print(2)
```

2 Faites place !

2.1 Énoncé

Aujourd'hui, Joseph Marchand a décidé de vendre des crêpes lui aussi. Malheureusement, il est arrivé un peu tard : tous les autres marchands ont déjà positionné leur stand, et Joseph doit se faire une place parmi eux, en choisissant une des positions restantes.

On considère encore la plage comme un segment de la position 0 (le plus à l'ouest), à la position N (le plus à l'est de la plage).

Les clients vont toujours acheter leurs crêpes auprès du marchand le plus proche d'eux, sans regarder les différences de prix ou de garnitures (c'est les vacances après tout, le moindre effort est épuisant). Joseph Marchand veut se placer afin d'avoir la plus grande zone d'influence possible sur les clients. Il doit se placer sur une position entière où un marchand n'est pas déjà installé. On rappelle que la zone d'influence d'un marchand est l'ensemble des points (pas forcément entiers) de la plage dont il est plus proche que n'importe quel autre marchand.

Joseph cherche à savoir quelle serait la taille de la plus grande zone d'influence qu'il peut avoir, arrondie à l'entier inférieur.

2.2 Solution

Il y a trois types de places que Joseph Marchand peut prendre pour son stand : soit juste avant le premier marchand, soit juste après le dernier marchand, soit entre deux marchands.

On commencera par définir la taille de la zone d'influence maximale comme étant celle si l'on se place juste avant le premier marchand. Ensuite, on va calculer la distance entre chaque marchand et vérifier si la zone d'influence en s'installant entre eux est plus étendue que celle qu'on a actuellement. La zone d'influence est simplement la distance entre les marchands divisée par deux. Enfin on vérifie si la zone juste après le dernier marchand n'est pas plus étendue, et on affiche la taille de la zone la plus importante que l'on a trouvée.

Nous aurions pu aussi garder toutes les valeurs des étendues des zones d'influence multipliée par 2 afin de ne pas avoir de nombres flottants (les $0,5$) à gérer.

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

```python
from math import floor

def faites_place(N, Ci):
    pos = Ci[0] - 1
    maxi = pos + 0.5
    for i in range(0, len(Ci) - 1):
        if (Ci[i + 1] - Ci[i]) / 2 > maxi:
            pos = Ci[i] + (Ci[i + 1] - Ci[i]) // 2
            maxi = (Ci[i + 1] - Ci[i]) / 2
    if N - Ci[len(Ci) - 1] - 1.5 > maxi:
        pos = Ci[-1] + 1
        maxi = N - Ci[-1] - 0.5
    return floor(maxi)

N = int(input())
M = int(input())
Ci = [int(i) for i in input().split()]

print(faites_place(N, Ci))
```

3 Organisation des vacances

3.1 Énoncé

Joseph Marchand est incapable de vendre ses crêpes tout seul sur la plage, et se fait donc aider par ses deux enfants. Cependant, ces derniers aimeraient bien visiter leurs grands-parents en guise de vacances, mais Joseph désire absolument faire grandir son stand de crêpes. Il accepte finalement de leur accorder des vacances, sous quelques conditions.

Tout d'abord, puisque Joseph a besoin de leur aide, il est nécessaire qu'au moins l'un d'eux soit présent à tout moment pour l'assister avec les crêpes (Joseph Marchand tolère cependant que l'un parte et l'autre revienne le même jour). De plus, vu qu'au sein de la famille les crêpes sont sacrées, tout l'argent de Joseph est investi dans le stand, ce qui signifie qu'il souhaite que la somme totale des deux voyages soit minimum.

Malheureusement Joseph est débordé par toutes les possibilités de voyage, et a besoin de votre aide pour faire son choix.

3.2 Solution

On nous demande de trouver exactement deux intervalles distincts ne s'intersectant pas, de telle sorte que la somme des pondérations des deux intervalles soit minimale.

La solution naïve et évidente consiste à tester chaque paire d'intervalle et de ne retenir que la meilleure. Cependant, la **complexité en temps** de cet algorithme est de l'ordre de $\mathcal{O}(N^2)$ ce qui ne passe pas les tests de performance où N peut aller jusqu'à 10^5.

Afin d'éviter de tester inutilement toutes les possibilités, regardons ce qu'il se passe si on teste un intervalle A. Le choix optimal du deuxième intervalle, noté B est en réalité unique, car c'est celui ayant une pondération minimale et se trouvant strictement avant A (puisque s'il se trouve après, cela revient au même que de tester l'intervalle B en choisissant l'intervalle A placé avant). Si l'on connaît pour chaque intervalle A, le choix correspondant B, on ne teste plus que N valeurs au total.

Pour cela, on peut définir un **événement** comme étant soit le début d'un intervalle, soit sa fin. On va alors trier dans l'ordre croissant tous les événements donnés en entrée et appliquer un *algorithme de balayage*. Ceci nous permet de toujours garder la valeur de l'intervalle de pondération minimale étant placé strictement avant l'intervalle qu'on teste, en suivant les deux cas possibles pour chaque événement à traiter :

- Si on est sur la fin d'un intervalle et que la pondération de l'intervalle actuel est la plus petite rencontrée jusqu'à présent, on la stocke en mémoire.
- Si c'est le début d'un intervalle, on teste si on peut obtenir une meilleure réponse en combinant l'intervalle actuel, et celui de coût minimal qu'on a précédemment gardé en mémoire.

Complexité Pour trier les événements, nous pouvons utiliser la fonction de la librairie standard, résultant en un tri en $\mathcal{O}(N \log N)$. Le reste de l'algorithme ne constitue qu'un parcours des événements ce qui est de l'ordre de $\mathcal{O}(N)$.

Listing 3 – Une solution de l'exercice 3 en C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

const int INF = 1e9;

int main(void)
{
    int n;
    std::vector<std::pair<int, int>> e;

    std::cin >> n;
```

```
13      for(int i = 0; i < n; ++i)
14      {
15          int d, f, c;
16          std::cin >> d >> f >> c;
17          e.push_back(std::make_pair(d, c));
18          e.push_back(std::make_pair(f, -c));
19      }
20
21      std::sort(e.begin(), e.end());
22
23      int rep = INF;
24      int mini = INF;
25      for(auto v : e)
26      {
27          if(v.second < 0)
28              mini = std::min(mini, -v.second);
29          else
30              rep = std::min(rep, v.second + mini);
31      }
32
33      if(rep == INF)
34          rep = -1;
35      std::cout << rep << "\n";
36
37      return 0;
38  }
```

Afin de facilement différencier le début de la fin d'un intervalle, on peut se dire qu'un coût positif représente le début et qu'un coût négatif la fin. De plus pour distinguer le cas où il n'y a pas de solution au problème, on initialise notre réponse finale à $+\infty$ (en pratique on utilisera un nombre suffisamment grand, 10^9 ici), et si à la suite de notre algorithme cette variable contient toujours $+\infty$ ceci signifie qu'on n'a pas trouvé une seule solution possible, on affiche donc -1.

4 Restauration rapide

4.1 Énoncé

Pour atteindre une clientèle plus importante, Joseph Marchand (de crêpes) envisage d'offrir un service de livraison sur la plage. Avant de lancer son nouveau modèle, il fait appel à vous pour évaluer la rentabilité de l'opération.

Joseph a tracé dans le sable des routes qui relient les clients. Pour gagner du temps, il a tracé le minimum possible de routes (mais il s'est tout de même assuré de relier tous les clients à son réseau de distribution).

En empruntant ses routes (dans un sens ou dans l'autre), Joseph se demande quelle est la distance moyenne à parcourir pour aller d'un client à un autre.

4.2 Solution

Le sujet consiste à calculer la longueur moyenne des chemins entre les sommets d'un arbre.

Encore une fois, une solution naïve serait de simplement faire un parcours depuis chaque sommet vers les autres afin de calculer la longueur moyenne. Mais à nouveau cette solution a une complexité en temps trop élevée de $\mathcal{O}(M^2)$, qui ne passe pas tous les tests de performance.

Pour obtenir une complexité linéaire, il est important de remarquer que la solution naïve retraverse de nombreuses fois les mêmes arêtes de l'arbre. On peut alors se contenter de calculer pour chaque arête le nombre de chemins d passant par cette dernière. On évite ainsi de nombreuses passes inutiles, car il suffit de multiplier la pondération de l'arête en question par d pour connaître la contribution totale de l'arête dans la somme finale.

En appliquant ce principe pour toutes les arêtes lors d'un parcours de l'arbre, on peut alors calculer la somme totale des pondérations de tous les chemins, que l'on divise par le nombre de chemins afin de connaître la longueur moyenne recherchée.

Complexité Cette version ne requiert qu'un seul parcours de l'arbre ce qui donne une complexité en temps linéaire : $\mathcal{O}(M)$.

Attention lors de l'implémentation à utiliser des types appropriés pour gérer les **grands nombres** (par exemple `long long` en C et C++).

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

```python
total = 0


def dfs(source, parent, neighbors, l):
    global total
    nbleft = 1
    n = len(neighbors)
    for nei, dist in neighbors[source]:
        if nei != parent:
            nbleft += dfs(nei, source, neighbors, dist)
    nbright = n - nbleft
    total += l * nbleft * nbright
    return nbleft


m = int(input())
n = m + 1
neighbors = [[] for _ in range(n)]
for i in range(m):
    a, b, l = [int(s) for s in input().split()]
    neighbors[a].append((b, l))
```

```
22      neighbors[b].append((a, l))
23
24  dfs(0, 0, neighbors, 0)
25  nb = n * (n - 1) // 2
26  print(int(total / nb))
```

5 Crêpes parfaites

5.1 Énoncé

Joseph Marchand se lance un défi, il doit produire une crêpe tout en se promenant sur la plage, promenade qui doit satisfaire certaines contraintes qu'il a définies à l'avance.

Il a commencé par dessiner des disques à certains endroits sur le sol qu'il numérote de 0 à N - 1, le disque 0 correspondant à l'emplacement de son stand à crêpes. Sur chacun d'entre eux, il plante une pancarte indiquant 2 directions pointant chacune vers respectivement 2 autres emplacements.

Sa promenade doit commencer à l'instant 0 au niveau de son stand et à chaque disque rencontré il doit prendre une des deux directions indiquées selon le critère suivant :

— la première si le nombre de 1 dans l'écriture binaire du nombre de disques déjà rencontrés est pair

— la seconde sinon

Joseph tient à ce que sa crêpe soit parfaite, pour cela il faut impérativement qu'elle soit retournée pile au milieu du temps de cuisson. Pour pouvoir réaliser une telle crêpe il faut donc qu'il se trouve au niveau de son stand aux instants T, $T + t$ puis $T + 2t$ où T est l'instant de début de cuisson, et t la durée de mi-cuisson, cela sachant qu'il met exactement une seconde (il est très rapide) pour se déplacer d'un disque à l'autre.

À l'instant 0 il met en marche sa poêle, et il ne peut pas lancer directement la cuisson d'une crêpe, il lui faudra attendre au moins son prochain retour sur son stand. Par ailleurs, comme il n'est pas infatigable, Joseph arrêtera sa promenade après la seconde L (il peut encore arrêter la cuisson d'une crêpe à cet instant).

Combien Joseph a-t-il de manières différentes de réaliser une crêpe parfaite ?

5.2 Solution

5.2.1 Débarassons-nous de ce problème de graphe

Tout d'abord, on va commencer par simuler le parcours de Joseph sur les L instants successifs. Le faire naïvement suffit.

On va noter tous les instants durant lesquels Joseph peut intéragir avec son stand. Plus exactement, on va avoir un tableau A tel que $A[i] = 1$ si $i > 0$ et Joseph est sur son stand à l'instant i, $A[i] = 0$ sinon.

5.2.2 Se ramener à un problème combinatoire

Soit T un instant où Joseph peut rajouter une crêpe parfaite sur son stand, et t une durée de demi-cuisson correspondante. Alors, Joseph est à son stand aux instants T, $T + t$ et $T + 2t$. C'est à dire que $A[T] = A[T + t] = A[T + 2t] = 1$. En notant $a = T$, $b = T + t$ et $c = T + 2t$, on a donc $A[a] = A[b] = A[c] = 1$, et on remarque que $a + c = 2b$ et $a < c$. Inversement, si on trouve a, b et c tels que $A[a] = A[b] = A[c] = 1$, $a + c = 2b$ et $a < c$, cela correspond à une crêpe parfaite pour $T = a$ et $t = b - a$.

Donc notre problème revient à compter le nombre de triplets (a, b, c) tels que $A[a] = A[b] = A[c] = 1$, $a + c = 2b$ et $a < c$.

Supposons que l'on ait un tableau B tel que $B[i]$ est le nombre de couples (a, c) tels que $a < c$, $A[a] = A[c] = 1$ et $a + c = i$. On verra par la suite comment obtenir un tel tableau.

Alors il est facile de résoudre notre problème ! Il suffit de commencer avec un résultat nul, et d'itérer sur les cases de A : chaque fois qu'on trouve un b tel que $A[b] = 1$, on rajoute $B[2b]$ au résultat.

5.2.3 Polynômes et compagnie

Comment donc obtenir un tel tableau B ? On peut le faire très facilement en temps quadratique, mais c'est trop lent. Voyons le problème différemment, et considérons le polynôme $A[0] + A[1]X + A[2]X^2 + A[3]X^3 + ... + A[L]X^L$ Appellons P ce polynôme.

Soit maintenant le polynôme $Q = P * P$

Q est évidemment un polynôme, et on peut s'intéresser à ses coefficients :

$Q = A[0]A[0] + (A[0]A[1] + A[1]A[0])X + (A[0]A[2] + A[1]A[1] + A[2][0])X^2 + ... + (A[0]A[L] + A[1]A[L-1] + ... + A[L]A[0])X^L + ... + (A[L]A[L])X^{2L}$

On voit que le coefficient associé à X^i dans Q compte la somme des $A[a]A[c]$ tels que $a + c = i$. Or A est à valeur dans $\{0, 1\}$, et donc $A[a]A[c]$ vaut 1 si et seulement si $A[a] = A[c] = 1$, et 0 sinon.

Donc le coefficient associé à X^i dans Q compte le nombre de couples (a, c) tels que $A[a] = A[c] = 1$ et $a + c = i$.

Si dans les cas où i est pair, on enlève $A[i/2]$ du coefficient de X^i, on obtient un polynôme R tel que le coefficient de X^i compte le nombre de couples (a, c) tels que $A[a] = A[c] = 1$, $a + c = i$, et $a! = c$ (i.e. on a enlevé les cas où $a = c$).

Si on divise tous les coefficients de R par 2, on a enfin un polynôme dont les coefficients sont donnés par le tableau B, c'est-à-dire tel que le coefficient de X^i compte le nombre de couples (a, c) tels que $A[a] = A[c] = 1$, $a + c = i$ et $a < c$.

On a alors résolu notre problème... à condition de savoir calculer Q rapidement !

5.2.4 Multiplication de polynômes

Il y a beaucoup d'algorithmes pour multiplier rapidement des polynômes. Le plus connu est basé sur la transformée de Fourier rapide (que l'on note souvent **FFT**, pour **F**ast **F**ourier **T**ransform).

Comment multiplier rapidement deux polynômes ?

Soit P et Q les deux polynômes, et R leur produit que l'on cherche à expliciter.

Tout d'abord, deux petites remarques :

— **Théorème d'interpolation de Lagrange :** Si on a N couples distincts (x_k, y_k), il existe un unique polynôme S de la forme $S[0] + S[1]X + ... + S[N-1]X^{N-1}$, tel que $S(x_k) = y_k$.

— On va supposer, quitte à rajouter des X^k avec des coefficients nuls, que P et Q sont de la forme $P[0] + P[1]X + P[2]X^2 + ... + P[M'-1]X^{M'-1}$ et $Q[0] + Q[1]X + Q[2]X^2 + ... + Q[M-1]X^{M-1}$ où $M + M' + 1$ est une puissance de deux. On remarque que R est de la forme $R[0] + R[1]X + ... + R[N-1]X^{N-1}$, où N est une puissance de deux.

L'idée est d'évaluer P et Q en N points x_k. On va donc avoir avoir N couples distincts $(x_k, y_{P,k})$ tels que $P(x_k) = y_{P,k}$ et N couples distincts $(x_k, y_{Q,k})$ tels que $Q(x_k) = y_{Q,k}$. Alors, en notant $y_k = y_{P,k} * y_{Q,k}$ on a N couples distincts (x_k, y_k), tels que $R(x_k) = y_k$. Et donc on peut reconstituer R par le théorème d'interpolation de Lagrange.

5.2.5 L'algorithme de Cooley-Tukey

Mais comment évaluer un polynôme T de la forme $T[0] + T[1]X + ... + T[N-1]X^{N-1}$ en N points en $\mathcal{O}(N \log N)$? On va choisir $x_k = e^{(\frac{2ik\pi}{N})}$.

L'algorithme suivant est celui de Cooley-Tukey.

— Si $N = 1$, c'est facile, il suffit de renvoyer $T[0]$.

— Sinon, on remarque que T est de la forme $\Sigma_{kpair} T[k]X^k + \Sigma_{kimpair} T[k]X^k = \Sigma_{k<N/2} T[2k](X^2)^k + X * \Sigma_{k<N/2} T[2k+1](X^2)^k$.

En appliquant recursivement la FFT à $T_{pair} = \Sigma_{k<N/2} T[2k]X^k$ et à $T_{impair} = \Sigma_{k<N/2} T[2k+1]X^k$, on obtient deux suites $(x_{T_{pair},k}, y_{T_{pair},k})$ et $(x_{T_{impair},k}, y_{T_{impair},k})$ telles que $T_{pair}(x_{T_{pair},k}) = y_{T_{pair},k}$ et $T_{impair}(x_{T_{impair},k}) = y_{T_{impair},k}$. On remarque alors que pour $x_k = e_{\frac{2ik\pi}{N}}$, on peut définir y_k ainsi : $y_k = x_{T_{pair},k} + e_{\frac{-2ik\pi}{N}} x_{T_{impair},k}$ si $k < N/2$ $y_k = x_{T_{pair},k-N/2} + e_{\frac{-2ik\pi}{N}} x_{T_{impair},k-N/2}$ sinon

Il reste un dernier petit détail : on a N couples (x_k, y_k) tels que $R(x_k) = y_k$. Mais comment appliquer (rapidement) le théorème d'interpolation de Lagrange ?

5.2.6 Application du théorème d'interpolation de Lagrange

En fait, il se trouve que quand $x_k = e_{\frac{2ik\pi}{N}}$, on peut considérer le polynôme $(y_0/N) + (y_1/N)X + (y_2/N)X^2 + ... + (y_{N-1}/N)X^{N-1}$, et lui appliquer la FFT afin d'obtenir N complexes r_k telles que $R(e_{\frac{2ik\pi}{N}}) = r_k$. Alors, magiquement, $R = r_{N-1} + Xr_{N-2} + ... + r_0 X^{N-1}$

5.2.7 Résumé de la solution

Résumons donc comment on résout ce problème :

1. On simule naivement les L mouvements de Joseph Marchand, et on a un tableau de longueur $L+1$ tel que $A[i] = 1$ si il est au stand à l'instant i, 0 sinon.
2. On considère $P = A[0] + A[1]X + ... + A[L]X^L$. Soit N une puissance de 2 supérieure à $2(L+1)$
3. On calcule la FFT de P par récurrence avec l'algorithme ci-dessus
4. On obtient alors N complexes y_k tels que $P(e_{\frac{2ik\pi}{N}}) = y_k$.
5. On considère le polynôme $P' = (y_0^2/N) + (y_1^2/N)X + ... + (y_{N-1}^2/N)X^{N-1}$
6. On calcule la FFT de P' par récurrence avec l'algorithme ci-dessus.
7. On obtient alors N complexes z_k (qui sont en fait des entiers) tels que $P'(e_{\frac{2ik\pi}{N}}) = z_k$
8. Soit B le tableau tel que $B[k] = z_{N-1-k}$.
9. On initialise un compteur *cpt* à 0
10. Pour tout $k \leq L$ tel que $A[k] = 1$, on rajoute $\frac{B[2k]-1}{2}$ à *cpt*
11. Le résultat est alors *cpt*

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

```
from cmath import exp, pi


def fft(x):
    N = len(x)
    if N <= 1:
        return x
    even = fft(x[0::2])
    odd = fft(x[1::2])
    T = [exp(-2j*pi*k/N) * odd[k] for k in range(N // 2)]
    return [even[k] + T[k] for k in range(N // 2)] + \
           [even[k] - T[k] for k in range(N // 2)]


def ifft(x):
    N = len(x)
    if N <= 1:
        return x
    even = ifft(x[0::2])
    odd = ifft(x[1::2])
    T = [exp(2j*pi*k/N) * odd[k] for k in range(N // 2)]
    return [(even[k] + T[k]) for k in range(N // 2)] + \
           [(even[k] - T[k]) for k in range(N // 2)]


n, l = map(int, input().split())
g = []
for i in range(l):
    x, y = map(int, input().split())
    g.append((x, y))

s = []
cur = 0
for i in range(n):
```

```
35      cur = g[cur][str(bin(i)).count('1') % 2]
36      if cur == 0:
37          s.append(1)
38      else:
39          s.append(0)
40
41  s += [0] * len(s)
42  p = 2
43  while p < len(s):
44      p *= 2
45  s += [0] * (p - len(s))
46
47  p = [round(abs(i / len(s))) for i in ifft([x * x for x in fft(s)])]
48
49  nb = 0
50  for i in range(len(s)):
51      if s[i] == 1:
52          nb += p[2 * i] // 2
53
54  print(nb)
```

5.2.8 D'autres ressources sur les FFT

La transformée de Fourier rapide n'est pas un concept facile à appréhender et nous avons essayé d'être le plus clair possible dans cette correction. Cependant, nous ne pouvons que vous recommander de lire davantage d'explications diverses sur le sujet afin de comprendre au mieux cette méthode de multiplication rapide de deux polynômes. Voici quelques liens pouvant vous être très utiles :

— Lecture 2 : Fast Fourier Transforms *Algorithms* par Jeff Erickson
— Tutorial on FFT/NTT — The tough made simple. (Part 1) Codeforces
— On Fast Fourier Transform Codeforces

⁂

Félicitations à tous les participants !

Si après avoir lu cette correction vous avez toujours des questions, n'hésitez pas à nous contacter à l'adresse info@prologin.org.