Solution de l'Exercice 1 – *OMG ça a (un peu) changé!*

Dans cet exercice, et dans tout ce sujet, on note sans prime (e.g. `i`) la valeur d'une quantité au début d'une itération, et avec prime (e.g. `i'`) la valeur en fin de cette itération (c'est à dire au début de l'itération suivante, si elle existe).

1. La fonction n'a pas d'effets secondaires, comme attendu. Montrons qu'elle renvoie la bonne valeur, c'est à dire qu'elle renvoie $\sum_{i \text{ pair} \in [\![0;n]\!]} i$.
Pour cela, on utilise les deux invariants suivants :

$$(S) : \quad \texttt{somme} = \sum_{k \text{ pair} \in [\![0;\texttt{i}-1]\!]} k$$

$$(P) : \quad \texttt{i est pair}$$

Je vais prouver les deux invariants à la fois.

- Initialisation : juste avant la première itération de la boucle, on a $\texttt{somme} = 0$. On a également $\texttt{i} - 1 < 0$, d'où la somme de (S) est vide et vaut donc 0. On a bien $0 = 0$: l'invariant (S) est initialisé.
  De plus, $\texttt{i} = 0$ est pair donc (P) est également initialisé.
- Conservation : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération.
  D'après (P), `i` est pair. Or $\texttt{i}' = \texttt{i} + 2$ d'où (P').
  On en déduit également que :

$$\{k \in [\![0; \texttt{i}' \ t.q.\ k \text{ pair}\}] = \{k \in [\![0; \texttt{i} \ t.q.\ k \text{ pair}\}] \cup \{\texttt{i}\}$$

  Or, on a $\texttt{somme}' = \texttt{somme} + \texttt{i}$. D'après l'égalité ensemble ci-dessus, on a donc (S')
  Les deux invariants se conservent.
- Conclusion : On a prouvé que (S) et (P) sont des invariants.

Or, à la fin de la dernière itération, `i` vaut le plus petit multiple de 2 strictement supérieur à `n`. Donc `i-1` est le plus petit impair strictement supérieur à `n`, donc en appliquant (S) en sortie de boucle on obtient que :

$$\texttt{somme} = \sum_{k \text{ pair} \in [\![0;n]\!]} k$$

Il s'ensuit que :

`somme_entiers` est partiellement correcte.

---

2. **a.** En notant $A(n)$ la complexité en additions

$$\boxed{A(n) = \Theta(n)}$$

NB : plus précisement, en notant p tel que $n = 2p$ ou $n = 2p+1$ on a $A(n) = 2p + 2$.

---

b. En notant $m = \left\lfloor \frac{n}{2} \right\rfloor$, remarquons tout d'abord que :

$$\sum_{k \text{ pair} \in [\![0;n]\!]} k = \sum_{p=0}^{m} 2p = m(m+1)$$

Il reste à renvoyer cette valeur. D'où :

somme-pairs.c

```c
int somme_pairs_faster(int n) {
  int m = n/2;
  return m * (m+1);
}
```

Solution de l'Exercice 2 – *Milieux dans un nuage de points*

1. Pour que le milieu soit à coordonnées entières, il faut et suffit que son abscisse et ordonnée le soient. Celles-ci sont une fraction de dénominateur 2, il suffit donc de tester que le numérateur est pair :

milieux.c

```c
/** Renvoie true si et seulement si le milieu de [a;b]
  * est à coordonnées entières.
  */
bool milieu_est_entier(point a, point b) {
  return    (a.x + b.x) % 2 == 0  // l'abs du milieu est paire
         && (a.y + b.y) % 2 == 0; // l'ord du milieu est paire
}
```

En anticipation de la prochaine question, on écrit également :

milieux.c

```c
/** Renvoie le point milieu de [a;b] (à coo entières) */
point milieu(point a, point b) {
  point milieu = {.x = (a.x + b.x)/2, .y = (a.y + b.y)/2};
  return milieu;
}
```

2. Les deux solutions proposées correspondent au pseudo-code suivant :

Entrées : *nuage* le nuage
Sorties : le nombre de paires
1 $total \leftarrow 0$
2 **pour chaque** *paire* (a, b) *du nuage avec* $a \neq b$ **faire**
3 | **si** *le milieu de* $[a;b]$ *est à coo entières* **alors**
4 | | **si** *ce milieu est dans le nuage* **alors**
5 | | | $total \leftarrow total + 1$
// On divise total par 2 car on a compté chaque segment 2 fois
6 **renvoyer** $total/2$

Il faut donc une fonction qui teste si un point est dans le nuage (pour la ligne 4). La voici :

milieux.c

```c
/** Renvoie true ssi x est présent dans une des len cases de nuage.
  * Complexité : O(len)
  */
bool mem(point p, point const nuage[], int len) {
  for (int i = 0; i < len; i = i+1) {
    if (cmp(p, nuage[i]) == 0) { return true; }
  }
  return false;
}
```

Cette fonction `mem` est en $O(len)$.

Voici maintenant une première version de `nb_milieux` en $O(len^3)$. La subtilité est qu'il faut penser à vérifier que $a \neq b$:

milieux.c

```c
/** Compte le nombre de milieux dans le nuage.
  * Complexité : O(len^2 * C_mem(len)) = O(len^3)
  */
int nb_milieux(point nuage[], int len) {
  int total = 0;
  for (int i = 0; i < len; i = i+1) {
    point a = nuage[i];
    for (int j = 0; j < len; j = j+1) {
      point b = nuage[j];
      if (cmp(a, b) != 0                    // a != b
          && milieu_est_entier(a, b)        // le milieu est entier
          && mem(milieu(a, b), nuage, len) // le milieu est dans le nuage
         )
      {
        total = total +1;
      }
    }
  }
```

Pour accélérer cette fonction, on peut remplacer `mem` par une recherche dichotomique qui s'exécute en $O(\log_2 len)$:

milieux.c

```c
/** Renvoie true ssi x est présent dans une des len cases de nuage.
  * Pré-conditions : le nuage doit être trié par ordre croissant.
  */
bool mem_opt(point p, point const nuage[], int len) {
  int lo = 0;
  int hi = len;

  // Invariant : si p est présent, il est dans nuage[lo:hi[
  while (lo < hi) {
    int mid = (lo+hi)/2;
    int comparaison = cmp(p, nuage[mid]);
    if (comparaison == 0) {
      // nuage[mid] == p
      return true;
    }
    else if (comparaison < 0) {
      // p < nuage[mid]
      hi = mid;
    }
    else {
      // p > nuage[mid]
      lo = mid+1;
    }
  }

  return false;
}
```

Et il n'y a plus qu'à modifier `nb_milieux` pour qu'elle utilise cette recherche améliorée (sans oublier de débuter par trier le tout). Notez que j'optimise légèrement en ne parcourant que les paires (a, b) où $a \leq b$. La complexité devient un $O\left(\texttt{len} \log_2 \texttt{len} + \texttt{len}^2 \log_2 \texttt{len}\right) = O\left(\texttt{len}^2 \log_2 \texttt{len}\right)$:

milieux.c

```c
/** Compte le nombre de milieux dans le nuage.
  * Complexité : O(len^2 * C(mem)) = O(len^3)
  */
int nb_milieux_opt(point nuage[], int len) {
  sort(nuage, len);

  int total = 0;
  for (int i = 0; i < len; i = i+1) {
    point a = nuage[i];
    for (int j = i+1; j < len; j = j+1) {
      point b = nuage[j];
      if (cmp(a,b) < 0
          && milieu_est_entier(a, b)
          && mem(milieu(a, b), nuage, len)
          )
      {
        total = total +1;
      }
    }
  }

  return total;
}
```

3. Pour certains de ces calculs, on utilise l'astuce $10^3 \simeq 2^{10}$. Je me sers également de $\log_2(100)$, pour cela j'utilise $2^7 = 128 \simeq 100$. Demandez-moi si vous avez des questions.

complexité \ len	100	1000	10^4	10^5
$\texttt{len}^3$	1ms	1s	15min	10 jours
$\texttt{len}^2 \log_2 \texttt{len}$	70µs	10ms	1s	3min

NB : si l'ordre de grandeur était à peu près bon (facteur d'erreur 10 à 100 toléré selon les cases), j'ai mis les points.

Solution de l'Exercice 3 – *Traversée de rivière (adapté depuis CCINP MPI 2023)*

0. Voici les états successifs du chemin. Je note « G » un randonneur venu de la gauche, « D » venu de la droite, et « . » le caillou vide. L'état final est celui décrit en question 6.

$$\text{G G G . D D} \xrightarrow{\text{avanceG}} \text{G G . G D D} \xrightarrow{\text{sauteD}} \text{G G D G . D} \xrightarrow{\text{avanceD}} \text{G G D G D .}$$

$$\xrightarrow{\text{sauteG}} \text{G G D . D G} \xrightarrow{\text{sauteG}} \text{G . D G D G} \xrightarrow{\text{avanceG}} \text{. G D G D G} \xrightarrow{\text{sauteD}} \text{D G . G D G}$$

$$\xrightarrow{\text{sauteD}} \text{D G D G . G} \xrightarrow{\text{avanceG}} \text{D G D . G G} \xrightarrow{\text{sauteG}} \text{D . D G G G} \xrightarrow{\text{avanceD}} \text{D D . G G G}$$

1. Il suffit de parcourir tous les cailloux jusqu'à trouver le caillou vide. J'ajoute `const` dans le prototype pour garantir que les données ne seront pas modifiées.

riviere.c
```c
/** Calcule l'indice du caillou vide */
int caillou_vide(chemin_caillou const* ch) {
  for (int i = 0; i < ch->len; i = i+1) {
    if (ch->cailloux[i] == VIDE) {
      return i;
    }
  }

  // Sinon : on n'a pas trouvé de caillou vide, impossible.
  assert(false);
}
```

Remarque. Pour signaler qu'une zone du code est inatteignable, on peut utilise `assert(false)` . C'est ce que je fais ici (mais ce n'était pas attendu !).

---

2. Il faut juste faire attention à ne pas « écraser » et perdre définitivement une des deux valeurs concernées :

riviere.c
```c
/** Échange le contenu des cailloux i et j */
void echange(chemin_caillou* ch, int i, int j) {
  etat tmp = ch->cailloux[i];
  ch->cailloux[i] = ch->cailloux[j];
  ch->cailloux[j] = tmp;

  if (ch->cailloux[i] == VIDE) { ch->pos_vide = i; }
  if (ch->cailloux[j] == VIDE) { ch->pos_vide = j; }
  return;
}
```

---

3. Je me base sur l'hypothèse de l'énoncé : il y a un seul caillou vide. L'énoncé n'explique jamais comment faire sortir des randonneurs du chemin, aussi je ne considère pas ce cas de figure. Ainsi, pour qu'un randonneur gauche avance, il doit avancer sur le caillou vide : autrement dit, un randonneur gauche peut avancer si et seulement si le randonneur à gauche du caillou vide est un randonneur gauche.

Il faut penser à vérifier que la gauche de la position vide existe.

riviere.c
```c
/** Teste si un randonneur gauche peut avancer */
bool randonneurG_avance(chemin_caillou const* ch) {
  return ch->pos_vide > 0 && ch->cailloux[ch->pos_vide-1] == GAUCHE;
}
```

---

4. On effectue le même raisonnement qu'à la question précédente.

riviere.c
```c
/** Teste si un randonneur gauche peut sauter */
bool randonneurG_saute(chemin_caillou const* ch) {
  return ch->pos_vide > 1 && ch->cailloux[ch->pos_vide-2] == GAUCHE;
}
```

---

5. La fonction `echange` ne contient ni boucle ni appels de fonctions ; elle effectue uniquement 6 lignes de codes toutes en temps constant. Donc :

`echange` s'exécute en $O(1)$.

Les deux autres fonctions sont encore plus clairement en $O(1)$:

`randonneurG_avance` et `randonneurG_saute` s'exécutent en $O(1)$.

6. On essaye de reconnaitre dans le chemin la situation finale : que des `DROITE`, puis un `VIDE`, puis que des `GAUCHE`. Si ce n'est pas le cas, on renvoie `false`. Notez que ce code fonctionne s'il n'y a aucun `DROITE` ou aucun `GAUCHE` : l'énoncé ne garantit jamais la présence d'une des deux sortes de randonneurs (voir des deux) !

riviere.c

```c
/** Teste si la traversée est terminée */
bool fini(chemin_caillou const* ch) {
  // On va lire tous les cailloux de gauche à droite.
  int ind = 0;

  // Lecture des randonneurs droite
  while (ind < ch->len && ch->cailloux[ind] == DROITE) {
    ind = ind +1;
  }

  // Après les droite, il doit y avoir un caillou vide
  if (ind >= ch->len || ch->cailloux[ind] != VIDE) {
    return false;
  }

  // Après le vide, il doit y avoir des gauche jusqu'à la fin
  ind = ind +1;
  while (ind < ch->len && ch->cailloux[ind] == GAUCHE) {
    ind = ind +1;
  }
  return ind == ch->len;
}
```

7. Il s'agit d'écrire un retour sur trace. Nous n'avions pas encore fait le cours ; aussi je n'ai pas pénalisé le fait de ne pas penser à annuler le mouvement d'un randonneur lorsque vous devez en essayer un autre (j'ai par contre valorisé le fait d'y avoir pensé). De plus, comme les 4 cas sont similaires, je n'ai pas pénalisé quelqu'un qui pressé par le temps écrit « de même » dans le code C (mais j'ai valorisé le fait de l'écrire en entier).

   L'écriture de ce code utilise le fait qu'en situation finale, aucun mouvement n'est possible : chaque appel peut donc d'abord tester la faisabilité les mouvements, et ensuite seulement vérifier si la situation actuelle est finale. Notez également que si elle existe, une suite de mouvements corrects est affichée (du dernier mouvement au premier).

riviere.c

```c
/** Retour sur trace pour trouver un passage */
bool passage(chemin_caillou* ch) {
  int pos_vide = ch->pos_vide;

  if (randonneurG_avance(ch)) {
    // avancer le randonneur gauche
    echange(ch, pos_vide-1, pos_vide);
    // essayer récursivement de voir si ça marche
    if (passage(ch)) {
      affiche(pos_vide-1, pos_vide);
      return true;
    }

    // Si ça n'a pas marché : annuler le déplacement
    echange(ch, pos_vide, pos_vide-1);
  }

  if (randonneurG_saute(ch)) {
    echange(ch, pos_vide-2, pos_vide);
    if (passage(ch)) {
      affiche(pos_vide-2, pos_vide);
      return true;
    }
    echange(ch, pos_vide, pos_vide-2);
  }

  if (randonneurD_avance(ch)) {
    echange(ch, pos_vide+1, pos_vide);
    if (passage(ch)) {
      affiche(pos_vide+1, pos_vide);
      return true;
    }
    echange(ch, pos_vide, pos_vide+1);
  }

  if (randonneurD_saute(ch)) {
    echange(ch, pos_vide+2, pos_vide);
    if (passage(ch)) {
      affiche(pos_vide+2, pos_vide);
      return true;
    }
    echange(ch, pos_vide, pos_vide+2);
  }

  // Sinon : personne ne peut faire de mouvement utile,
  // on vérifie si la traversée est finie ou non
  return fini(ch);
}
```

---

8. Question difficile si le variant V n'est pas fourni. En fait, ce variant est pensé pour quefaire avancer/sauter dans la bonne direction un randonneur fasse croitre le variant.
Prouvons la terminaison de la fonction `passage` . Elle effectue des appels à d'autres fonctions. Les fonctions déjà prouvées en $O(1)$ terminent trivialement. La fonction `fini` car elle ne contient pas d'appels de fonction et que dans ses deux boucles `ind` est un variant (entier strictement croissant majoré par `len`).
Reste à prouver que la suite d'appels récursifs de `passage` termine, à l'aide du variant proposé. Or :

- Pour tout chemin ch, $V(ch) \in \mathbb{Z}$ puisque c'est une somme d'entiers.

- Pour tout chemin ch, $V(ch) < \texttt{len}^2$ puisque chaque randonneur est éloigné d'au plus `len` de sa rive d'origine et qu'il y a au plus `len` randonneurs.
- Si l'on note ch le chemin au début d'un appel, le chemin ch' au début d'un appel récursif vérifie $V(ch') > V(ch)$ car :
  - soit ch' est ch avec un randonneur gauche qui a avancé (lignes 148-152). Dans ce cas, ce randonneur s'est éloigné de 1 de sa rive d'origine, et les autres n'ont pas bougé. Donc $V(ch') = 1+V(ch) > V(ch)$.
  - Similaire dans les trois autres cas.

Ainsi, la suite d'appels récursifs admet un variant donc termine, et il s'ensuit que :

`passage` termine.

---

9. Au lieu d'afficher le mouvement, on peut l'ajouter à une liste de mouvements. Cette liste pourrait par exemple être un tableau dynamique, qui serait partagé entre les appels et que les appels modifieraient (en ajouter le mouvement à la fin du tableau) en cas de succès du retour sur trace.

Ainsi, à la fin du retour sur trace, si une solution existe elle serait stockée dans le tableau dynamique (de la fin au début).

Solution de l'Exercice 4 – *Tri par files parallèles (2nd concours ENS 2011)*

Ce corrigé est adapté d'un corrigé de G. Héméry.

1. Chaque élément doit être déplacé exactement une fois vers les files intermédiaires puis exactement une fois les la file résultat depuis une file intermédiaire. Au total :

Un scénario de tri se compose de $2n$ déplacements.

---

2. On considère $\mathcal{T}$ un scénario de tri. Puisque $\mathcal{T}$ trie correctement, il n'essaye jamais de défiler une file vide (NB : c'est un point que l'énoncé met sous le tapis ; mais il me semblait important de le mentionner. Numérotons les enfilages successifs : le premier enfilage a lieu dans la file i_1, le second dans i_2, etc. De même, on note $o_1, o_2, \ldots$ les files des défilages successifs.

D'après la propriété FIFO, l'ordre de sortie des éléments d'une file est entièrement déterminé par l'ordre d'entrée ; autrement dit, le $k$ème Out(i) défile le $k$ème élément enfilé dans F_i. Ainsi, on peut faire tous les enfilages d'une file avant tous ses défilages sans que cela ne change les éléments défilés.

On peut alors conclure : les intercalages des Out o_j entre les In i_j ne changent pas les éléments défilés. On peut donc faire tous les enfilages avant tous les défilages ; tant que l'on intervertit pas deux In ou deux Out.

NB : cela ne devrait pas être une question... on devrait vous demander de faire cette transfo sur un exemple et voilà. Ou alors, donner un formalisme précis qui permet une preuve rigoureuse. Un entre deux (comme ce que j'ai fait ici) n'est pas satisfaisant.

---

3. L'élément en tête d'une file F_i sera défilé de F_i et enfilé dans `resultat` avant que les autres éléments de F_i ne le soient (FIFO). En particulier, il est enfilé dans `resultat` *avant* les autres éléments de F_i : par correction d'un tri, la tête de F_i est donc inférieure aux autres éléments de F_i. De même, le second élément de F_i est inférieur aux éléments suivants, etc : F_i est triée.

Ce raisonnement peut-être fait à n'importe quelle étape de l'algorithme. On en déduit que :

À chaque étape d'un scénario de tri, les files intermédiaires sont triées.

---

4. Si x_i et x_j (avec $i < j$) sont inversés par S, on a $x_i > x_j$. On ne peut donc pas les enfiler dans la même file intermédiaire sans rompre l'invariant prouvé à la question précédente.

Plus généralement, si $x_{i_1} > x_{i_2} > \ldots > x_{i_k}$ est une sous-séquence strictement décroissante de S, alors pour tous $j < k$, x_{i_j} et x_{i_k} ne peuvent pas aller dans la même file. Il s'ensuit qu'il faut (au moins) k files distinctes.

---

5. Si l'on place tous les éléments de S dans les files intermédiaire de manière triée, on peut alors appliquer l'algorithme ci-dessous pour effectuer les *Out* :

Algorithme 3 : Déplacements de sortie

```
1 tant que il existe une file intermédiaire non-vide faire
2     i ←—indice de la file non-vide ayant la tête la plus petite
3     x ←—Défiler(F_i)
4     Enfiler(x, resultat)
```

Cet algorithme est correct car puisque les files intermédiaires sont triées il respecte l'invariant suivant : « le plus petit élément des files intermédiaires est en tête d'une des files intermédiaires ».

NB : nous avons donc montré qu'il est nécessaire (q4) et suffisant (q5) qu'un scénario normal débute par placer tous les éléments dans les files intermédiaires de sorte à ce qu'elles soient triées.

6. D'après la q4, il faut au moins 3 files. On peut montrer que c'est suffisant à l'aide du scénario ayant l'état intermédiaire ci-dessous :
(TODO joli dessin)

7. D'après le théorème, trier avec 2 files en parallèle signifie que la plus longue sous-séquence décroissante est de longueur 2.
Je n'ai pas le temps d'en trouver une. Disons que ça fera un bon exercice de backtrack ?

8. On propose l'algorithme suivant :

Algorithme 4 : Entrée avec infinité de files

```
1 tant que donnee est non-vide faire
2     x ←—Défiler(donnee)
3     i ←—rechercher le plus petit i tel que Q(F_i) < x
        // Par définition de i, pour tout j < i on a Q(F_i) > x
4     Enfiler(x, F_i)
```

La recherche de i peut-être effectuée par une recherche linéaire comme ci-dessous :

Fonction recherche-lineaire(x)

```
1 i ←—0
2 tant que x < Q(F_i) faire i ← i + 1
3 renvoyer i
```

Le fait que le premier invariant est vérifié s'obtient via le fait que l'on insère dans une file où x est plus grand que la queue ; le second invariant s'obtient à l'aide du commentaire en ligne 4.
Montrons plus précisément la conservation de ce second invariant (ce n'était pas attendu !) : supposons que $\big(Q(F_j)\big)$ soit décroissante avant l'insertion de x dans F_i.
Comme on a uniquement modifié F_i, pour $j \neq i$ on a $Q'(F_j) = Q(F_j)$ et donc d'après l'invariant $Q'(F_1), ..., Q'(F_{i-1})$ est encore une suite décroissante et $Q'(F_{i+1}), ...Q'(F_\infty)$ aussi.
Par transitivité, il suffit donc de montrer que $Q'(F_{i-1}) > Q'(F_i) = x > Q'(F_{i+1})$. La première inégalité s'obtient par définition de i. Mais comme $x > Q(F_i)$ et que d'après l'invariant $Q(F_i) > Q(F_{i+1})$, on obtient la seconde inégalité. D'où la conservation de l'invariant.

9. Montrons le résultat par récurrence forte sur $p \in \mathbb{N}^*$.

- Pour $p = 1$, l'élément seul est bien une sous-séquence décroissante de longueur 1.
- Soit $p > 1$, on suppose le résultat vrai pour tout p' tel que $p' \leq p - 1$.
  Considérons le p-ème déplacement d'entrée, et notons i_p le numéro de la file où il enfile. Si i_p n'était pas une file vide, alors il n'y a pas de nouvelle file vide et on peut simplement appliquer l'hypothèse de récurrence (une sous-séquence de $[s_1, ..., s_{p-1}]$ étant une sous-séquence de $[s_1, ..., s_p]$).
  Si i_p était une file vide : alors $F_{i_p - 1}$ était non-vide. En appliquant l'hypothèse de récurrence au rang $i_p - 1$, on obtient une sous-séquence décroissante D de $[s_1, ..., s_{p-1}]$ de longueur $i_p - 1$. Comme tous ces éléments ont été enfilés avant s_p, ils apparaissaient avant s_p dans la séquence d'origine et donc D rallongée de s_p est une sous-séquence de $[s_1, ..., s_p]$
  De plus, d'après le second invariant de la q8, pour tout $j < i_p$, $s_p < Q(F_j)$. Il s'ensuit que l'on peut ajouter s_p à la sous-séquence décroissante D sans rompre la décroissance : on obtient une sous-séquence décroissante de la longueur demandée. D'où l'hérédité.

Nous avons montré en q4 qu'il faut au moins k files pour trier. Réciproquement, on vient de montrer que si l files sont utilisées alors $l \leq k$. Donc :

Il faut et suffit de k files pour trier.

10. D'après la q9, la recherche linéaire parcourt au plus k files. Elle effectue des opérations en temps constant sur chacune d'entre elles (accès à la queue et incrémentation de i), et un temps constant hors de sa boucle : elle est donc en $O(k)$.

La fonction principale a une boucle qui itère n fois (elle défile un élément de donnée à chaque itération et s'arrête quand cette file est vide) et le coût de chaque itération est dominé par le coût de la recherche donc en $O(k)$. Il n'y a pas d'opérations hors de la boucle.

Soit $O(nk)$ pour les entrées.

De même, les sorties font n fois une recherche du minimum des têtes de k files (cf q5), donc sont en $O(nk)$. Au total :

Mon code s'exécute en $O(nk)$.

Pour l'implémenter, il faut avoir une structure qui permettre de stocker toutes les files, et de rajouter facilement une file dans la structure :

Un tableau dynamique de files est adapté pour stocker le réseau.

---

11. On peut accélérer la recherche en utilisant une recherche dichotomique parmi les $Q(F_i)$ au lieu de la recherche linéaire. Les files étant stockées dans un tableau dynamique, cela est une recherche dichotomique classique.

NB : Cela demande de connaitre non pas n, mais le nombre de files non-vides. On peut le mémoriser au fur et à mesure.

Puisqu'il y a au plus k files, la recherche dichotomique s'exécutera en $O(\log k)$ et on obtient alors les déplacements d'entrées en $O(n \log k)$.

---

12. Pour accélérer les déplacements de sortie, on peut stocker les indices des F_i dans une file de priorité ordonnée par $Q(F_i)$. Si on utilise un tas-min à cette fin, Chaque itération de la boucle des déplacements de sortie se fait alors en $O(\log k)$ d'où le résultat.

---

13. J'avoue ne pas savoir ce qui était attendu.

On peut insérer s_i dans F_{s_i} et communiquer $\min S$ à la partie gérant les sorties ; mais fera que le tableau des files commencera par $\min S - 1$ files vides... ce qui prend uu temps $O(\min S)$ à créer.

De plus, si on s'autorise à manipuler $\min S$, alors autant ne pas utiliser de réseau de files et simplement calculer le min et le max de S et renvoyer $[\![\min S; \max S]\!]$...