DEVOIR SURVEILLÉ IV

Informatique

Durée : 3 heures

Consignes :

- La candidate attachera la plus grande importance à la **clarté**, à la **précision** et à la **concision** de la rédaction. *2 points de la note finale sur 20 sont reservés au soin de la copie.*[1]
- Si une candidate est amenée à repérer ce qui peut lui sembler être une erreur d'énoncé, elle le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'elle a été amenée à prendre.
- L'usage du cours, de notes, d'une calculatrice ou de tout autre appareil électronique est interdit. **L'usage de la feuille de bonnes pratiques de programmation, éventuellement annotée de conseils méthodologiques est autorisée.**
- L'usage d'éléments du langage C non-encore vus en cours/TP/TD est interdit. Cela concerne notament l'opérateur `++`, ainsi que `calloc` et `realloc`. *En cas de doute, demandez-moi.* L'utilisation de `else if` est autorisé.
- L'usage d'éléments du langage OCaml non encore vus en cours/TP/TD est interdit. Cela concerne notament tous les aspects impératifs du langage (dont les `;`). *En cas de doute, demandez-moi.* L'usage d'exceptions est autorisé mais jamais attendu.

Pour une bonne présentation de la copie, il est notamment demandé à la candidate de :

- Mettre en valeurs les résultats finaux avec une couleur différente de sa couleur principale.
- Tirer un trait horizontal entre chaque question afin que le début et la fin d'une question soient facilement identifiables.
- Appliquer les bonnes pratiques de programmation. En particulier, si elle écrit un programme non-trivial, la candidate doit le préfixer d'un résumé de son fonctionnement en deux ou trois lignes. *Je m'autorise à mettre 0 à un code compliqué mal écrit, quand bien même il serait correct.*
- Numéroter ses pages en bas à droite sous le format numéro_de_la_page / nombre_total_de_pages.
- Ne pas utiliser de correcteur blanc.[2]

Dans tout le devoir, on supposera que les librairies C suivantes sont déjà incluses : `stdio.h` , `stdlib.h` , `stdbool.h` , `stdint.h` , `limits.h` . Quelques erreurs de syntaxe ponctuelles pourront être tolérées. Lorsque l'on programme en OCaml, on programme pour le compilateur `ocamlopt` et non pour l'interpréteur `utop` : en particulier, les `;;` sont interdits.

Les exercices sont indépendants.

Ne retournez pas la page avant d'y être invitées.

1. Ce n'est pas une lubie personnelle, cela se fait aux concours. Par exemple, CCINP maths réserve 1 point sur 20 pour le soin.
2. Car cela ne passe pas aux scanners des concours, et est en conséquence interdit aux concours.

I - Proche du cours

Exercice 1 – *OMG ça a (un peu) changé!*

On rappelle que par convention, une somme vide vaut 0.
On considère la fonction ci-dessous :

somme-pairs.c

```c
/** Renvoie la somme des entiers pairs de 0 à n (inclus).
  * Pas d'effets secondaires.
  */
int somme_pairs(int n) {
  int somme = 0;

  for (int i = 0; i <= n; i = i + 2) {
    somme = somme + i;
  }

  return somme;
}
```

Elle doit donc calculer :

$$\sum_{i \ pair \in [\![0;n]\!]} i$$

1. Prouver la correction partielle de cette fonction (on ignore les éventuels débordements).
   Attention, cela nécessite **deux** invariants (à ma connaissance). Il est attendu que vous énonciez les deux mais vous pouvez n'en prouver qu'un.
   Indice : si dans votre preuve vous affirmez sans rpouver une propriété sur des variables car "elle est évidente", c'est qu'elle doit faire l'objet d'un invariant.
2. **a.** Sans justification, quelle est la complexité de cette fonction en nombre d'additions ? On exprimera la réponse à l'aide d'un $\Theta()$
   b. Trouver une autre façon d'écrire cette fonction, en $O(1)$.
   On fera attention à éviter les débordements évitables : les calculs intermédiaires ne doivent jamais être plus grands que la valeur renvoyée.

II - Un problème algorithmique

Exercice 2 – *Milieux dans un nuage de points*

Cet exercice est à résoudre en C. Cet exercice est l'exercice « Les bons milieux » de la section « Efficacité Temporelle » du Niveau 3 de France-IOI.

On manipule dans cet exercice des points. On utilise le type ci-dessous pour les représenter :

milieux.c

```c
/** Un point est une paire (x,y) */
struct point_s {
  int x; // abscisse
  int y; // ordonnée
};
typedef struct point_s point;
```

On considère donnée en entrée un entier positif `len` et `nuage` tableau de `len` points deux à deux distincts. L'ensemble des points stockés dans ce tableau est appelé un *nuage de points*.
On voudrait trouver le nombre de points du nuage qui sont le milieu d'un segment entre deux autres points distincts du nuage, avec multiplicité.

Exemple. Considérons les points $a = (0,4); b = (2,4); c = (4,4); d = (4,0); e = (1,2); f = (2,2); g = (3,2)$. Le schéma ci-dessous représente le nuage de points associés Il y a 3 milieux dedans : b (milieu de $[a;c]$), f (milieu de $[a;d]$ et de $[e;g]$) et g (milieu de $[b;d]$). La réponse à renvoyer sur cette entrée est donc 4 (2 points de multiplicité 1, et 1 de multiplicité 2).

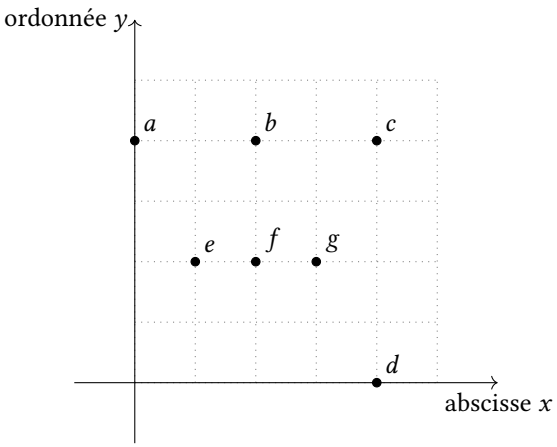

Figure IV.1 – Le nuage P

On rappelle que si $(i,j) \in \mathbb{Z}^2$ et $(k,l) \in \mathbb{Z}^2$, le milieu du segment $[(i,j);(k,l)]$ est $\left(\frac{(i+k)}{2}; \frac{(j+l)}{2}\right)$.

Pour que le milieu de $[a;b]$ soit un point du nuage, il faut déjà que ce soit un point à coordonnées entières.

1. Écrire une fonction `bool milieu_est_entier(point a, point b)` qui renvoie `true` si et seulement si le milieu du segment `[a;b]` est à coordonnées entières.

Définition 1 (Ordre lexicographique sur $\mathbb{Z}^2$).
Pour comparer deux points (i,j) et (k,l) de $\mathbb{Z}^2$:

- On pose $(i,j) < (k,l)$ si et seulement si $i < k$ ou $(i = k \text{ et } j < l)$.
- On pose $(i,j) = (k,l)$ si et seulement si $i = k$ et $j = l$.
- On pose $(i,j) \leq (k,l)$ si et seulement si $(i,j) < (k,l)$ ou $(i,j) = (k,l)$

Cet ordre est appelé l'**ordre lexicographique** sur $\mathbb{Z}^2$.

Exemple. $(2,100) < (3,0)$

Pour utiliser cet ordre, on suppose données :

- une fonction `int cmp(point a, point b)` qui renvoie -1 si $a < b$, 0 si $a = b$ et 1 sinon (si $a > b$). Elle s'exécute en temps $O(1)$.
- une fonction `void sort(point nuage[], int len)` qui permet de trier le nuage de points nuage par ordre lexicographique croissant. Elle s'exécute en temps $O(\texttt{len}\log_2(\texttt{len}))$.

Exemple. Le nuage de point précédent trié est :

$$a = (0,4); e = (1,2); f = (2,2); b = (2,4); g = (3,2); d = (4,0); c = (4,4)$$

2. Écrire une fonction `int nb_milieux(point nuage[], int len)` qui prend en entrée un nuage de points et renvoie le nombre de points de ce nuage qui sont le milieu d'un segment de deux autres points distincts du nuage (avec multiplicité).
Une solution naïve est en $O(\texttt{len}^3)$. Une solution améliorée en $O(\texttt{len}^2\log_2(\texttt{len}))$ est possible.
⚠ Une erreur classique consiste à renvoyer le *double* de la réponse attendue, en comptant chaque segment deux fois (en tant que $[a;b]$ et que $[b;a]$).

3. On considère un processeur à ~1GHz. Donner l'ordre de grandeur du temps d'exécution de la solution naïve et de la solution améliorée pour `len` $= 100$, `len` $= 1000$, `len` $= 10^4$ et `len` $= 10^5$. Commenter.
On rappelle que $1heure = 3600s \simeq 4.10^3 s$ et que $1jour = 86400s \simeq 9.10^4 s$

On suppose désormais que tous les points ont une abscisse et une ordonnée comprise entre 0 et 100 (inclus).

4. Proposer une fonction `int nb_milieux_100(point nuage[], int len)` résolvant le problème en $O(\texttt{len}^2)$.
Si jamais on fait une solution en $O(\texttt{len})$ avec un gros terme constant cachée dans le $O()$, on explicitera ce terme constant.

III - Concours CCINP

Si vous traitez cet exercice, ne traitez pas le suivant (2nd concours ENS)

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

Cet exercice est à traiter en **C**.
Dans une vallée des Alpes, un passage à gué fait de cailloux permet de traverser la rivière. Deux groupes de randonneurs arrivent simultanément sur les berges gauche et droite de cette rivière et veulent la traverser. Le chemin étant très étroit, une seule personne peut se trouver sur chaque caillou de ce chemin (figure ci-dessus). Un randonneur sur la berge de gauche peut avancer d'un caillou (vers la droite sur la figure) et sauter par dessus le randonneur devant lui (un caillou à droite) si le caillou où il atterit est libre. De même, chaque randonneur de la berge de droite peut avancer d'un caillou (vers la gauche sur la figure) et sauter par dessus le randonneur devant lui, dans la mesure où le caillou sur lequel il atterit est libre. Une fois engagés, les randonneurs ne peuvent pas faire marche arrière. De plus, pour simplifier, on suppose qu'une fois tous les randonneurs sur le chemin, il ne reste qu'un caillou de libre.

Figure IV.2 – Les randonneurs et le chemin de cailloux

0. Donnez une suite de déplacements de randonneurs qui leur permette de traverser dans cet exemple.

Un caillou contient donc soit un randonneur venu de la gauche, soit un randonneur venu de la droite, soit est vide. On crée à cette fin le type `etat`. Un élément de ce type ne peut prendre que trois valeurs : `GAUCHE`, `DROITE` et `VIDE`.

riviere.c

```c
enum etat_s {GAUCHE, DROITE, VIDE};
typedef enum etat_s etat;
```

Le chemin de cailloux est défini par une structrure qui contient un tableau d'états et sa longueur, ainsi que l'indice du caillou vide. On définit une structure contenant ces 3 données :

riviere.c

```c
struct chemin_caillou_s {
  etat* cailloux;  // contenu des cailloux
  int len;         // nombre de cailloux
  int pos_vide;    // indice du caillou vide
};
typedef struct chemin_caillou_s chemin_caillou;
```

On manipulera des `chemin_caillou*`. On rappelle que si `ch` est un `chemin_caillou*`, on peut accéder à ses 3 champs via respectivement `ch->cailloux`, `ch->len`, `ch->pos_vide`

Exemple. Voici par exemple une fonction qui renvoie un pointeur vers le chemin de cailloux correspondant à la figure :

riviere.c

```c
chemin_caillou* demo(void) {
  int len = 6;
  etat* cailloux = (etat*) malloc(6*sizeof(etat));
  // Cf fig : G G G vide D D
  cailloux[0] = GAUCHE;
  cailloux[1] = GAUCHE;
  cailloux[2] = GAUCHE;
  cailloux[3] = VIDE;
  cailloux[4] = DROITE;
  cailloux[5] = DROITE;

  chemin_caillou* ch = (chemin_caillou*) malloc(sizeof(chemin_caillou));
  ch->cailloux = cailloux;
  ch->len = len;
  ch->pos_vide = 3;

  return ch;
}
```

1. On considère dans cette question un chemin où le champ `pos_vide` est invalide (par exemple, il contient `-1`); et on doit donc recalculer cette position. Écrire une fonction de prototype `int caillou_vide(chemin_caillou* ch)` qui renvoie l'indice du caillou vide du chemin.

Dans toute la suite, les chemins ont bel et bien le champ `pos_vide` correctement rempli.

2. Écrire une fonction de prototype `void echange(chemin_caillou* ch, int i, int j)` qui modifie le chemin pointé pour y échanger le contenu des cailloux d'indice `i` et `j`.
   On n'oubliera pas de mettre à jour `pos_vide` !
3. Écrire une fonction de prototype `bool randonneurG_avance(chemin_caillou* ch)` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse avancer (vers la droite).
4. Écrire une fonction de prototype `bool randonneurG_saute(chemin_caillou* ch)` qui teste si, parmi les randonneurs venant de la berge de gauche, il en existe un qui puisse sauter (vers la droite) au-dessus d'un randonnenneur.

On supposera dans la suite les fonctions de prototypes `bool randonneurD_avance(chemin_caillou* ch)` et `bool randonneurD_saute(chemin_caillou* ch)` écrites de manière similaire pour les randonneurs venant de la berge de droite.

À partir d'ici, les questions diffèrent du sujet CCINP d'origine, mais l'idée reste la même.

5. Prouvez les complexités des fonctions des questions 2 3 et 4.
6. Écrire une fonction `bool fini(chemin_cailloux* ch)` qui renvoie `true` si et seulement si les randonneurs ont fini de traverser, c'est à dire si le chemin est de la forme :

   `DROITE ... DROITE  VIDE  GAUCHE ... GAUCHE`

On suppose donnée une fonction de prototype `void affiche(int src, int dst)` qui affiche un déplacement d'un randonneur du caillou d'indice `src` au caillou d'indice `dst`.

7. Écrire une fonction récursive de prototype `bool passage(chemin_cailloux* ch)` qui renvoie `true` si et seulement si il est possible de faire traverser les randonneurs. De plus, si cela est possible elle affichera (dans l'ordre ou à l'envers) les mouvements successifs à l'aide de `affiche`.
   Indication : essayez un mouvement, et demandez à vos camarades intelligents si le nouvel état est résoluble. Si oui, bravo, sinon annulez le mouvement que vous venez de faire et essayez en un autre.
8. Prouvez la terminaison de votre fonction. Pour cela, vous pourrez noter V la fonction qui à un chemin de cailloux associe :
   - La somme de pour chaque randonneur de gauche sa distance à la rive gauche, à laquelle s'ajoute :
   - la somme de pour chaque randonneur de droite sa distance à la rive droite.
9. On voudrait renvoyer la suite des mouvements au lieu de l'afficher. Expliquer comment cela pourrait se faire, et quelle(s) structure(s) de données serai(en)t utile(s).

IV - Second concours ENS

Si vous traitez cet exercice, ne traitez pas le précédent (CCINP)

Définitions et notations

Un *intervalle* désigne un intervalle fermé non-vide d'entiers de $\mathbb{N}$ c'est à dire de la forme $[\![a;b]\!]$ avec $a \in \mathbb{N}$, $b \in \mathbb{N}$ et $a \leq b$.

Le *cardinal* d'un ensemble fini est son nombre d'éléments. Pour un ensemble X et un entier naturel n, on utilise l'abus de notation qui consiste à écrire $x_1, x_2, ..., x_n \in X$ à la place de $(x_1, x_2, ..., x_n) \in X^n$.

Une *séquence* $S = (s_i)_{1 \leq i \leq k}$ avec $k \in \mathbb{N}$ est une suite finie d'éléments. L'entier k est appelé la *longueur* de la séquence S et par convention la séquence est vide lorsque $k = 0$. Dans la suite, les séquences S seront notées entre crochets : $[s_1, s_2, ..., s_k]$. De manière intuitive, une *sous-séquence* S' de S est une séquence formée d'éléments de S qui apparaissent dans S' avec le même ordre relatif que dans S. Formellement, une sous-séquence S' de longueur $k' \leq k$ d'une séquence $S = [s_1, s_2, ..., s_k]$ est une séquence de k' éléments telle qu'il existe une application strictement croissante σ de $[\![1, k']\!]$ dans $[\![1, k]\!]$ telle que $S' = [s_{\sigma(1)}, s_{\sigma(2)}, ..., s_{\sigma(k')}]$.

Une séquence est dite *sans répétitions* lorsque tous ses éléments sont deux à deux distincts. On dit qu'une séquence d'entiers sans répétitions S de longueur k *inverse deux de ses éléments* s_i et s_j, avec $i, j \in [\![1, k]\!]$, si et seulement si $i < j$ et $s_i > s_j$. Une *permutation* S de $[\![1, n]\!]$ est une séquence sans répétitions de longueur n dont les éléments sont les entiers de 1 à n.

Exercice 4 – *Tri par files parallèles (2nd concours ENS 2011)*

Cet exercice est à traiter en **pseudo-code ou en français**.

Le but de cet exercice est de trier une séquence d'entiers strictement positifs sans répétitions en faisant circuler les entiers qui la composent dans un réseau de files en parallèle. Nous considérons l'ensemble de toutes les séquences, *sans restriction sur leur longueur*. On se propose de répondre aux questions suivantes : quelles séquences peut-on trier avec k files en parallèle ? ou réciproquement, combien de files faut-il pour trier une séquence donnée ?

Dans toute la suite, même si on omet de le préciser, les séquences d'entiers considérées sont des *séquences d'entiers strictement positifs sans répétitions*.

Une *file* est un espace de stockage linéaire à deux extrémités dans lequel les éléments entrent d'un côté, appelé la *queue* de la file, et ressortent de l'autre côté, appelé la *tête* de la file, sans modification de leur ordre relatif. C'est-à-dire que l'ordre de sortie des éléments par la tête de file est le même que leur ordre d'entrée par la queue de file. Les notions de *croissance*, *décroissance* et *monotonie* d'une file se rapportent à la notion en question pour la séquence des entiers qu'elle contient pris depuis la tête de file vers la queue de file. Les seules opérations possibles sur une file sont : tester si la file est vide, lire la tête et la queue de la file, défiler la file (c'est-à-dire extraire son élément de tête), enfiler un élément dans la file (c'est-à-dire insérer l'élément en queue de file).

Un réseau de k files en parallèle est composé au total de $k + 2$ files :

- 1 *file donnée*
- k *files intermédiaires* en parallèle
- 1 *file résultat*

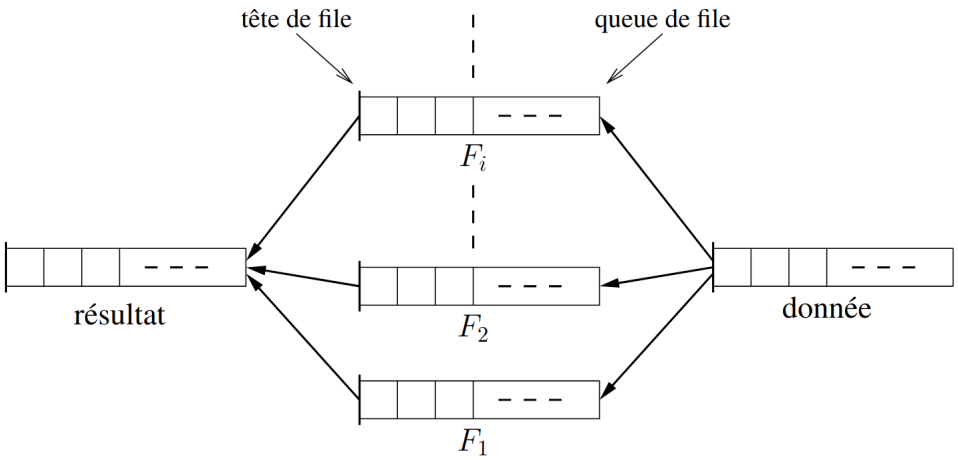

Figure IV.3 – Un réseau de files en parallèle. La tête des files est matérialisée par un trait plus long et plus épais.

Pour trier une séquence S par un réseau de k files en parallèle (voir figure ci-dessus) on fait circuler les entiers composant S dans le réseau en suivant les arcs joignant les files, exclusivement dans le sens de leur orientation, depuis la file donnée jusque dans la file résultat.
Initialement, la séquence $S = [s_1, s_2, ..., s_n]$ à trier, où $n \in N^*$ est la longueur de S, est placée dans la file donnée : s_1 en tête de file et s_n en queue de file. Ensuite, on effectue les déplacements des entiers dans le réseau : un déplacement après l'autre et un entier à la fois. On ne peut opérer que deux types très contraints de déplacement :

- les *déplacements d'entrée* consistent à retirer l'élément en tête de la file donnée pour le placer en queue d'une file intermédiaire quelconque F_i, avec $i \in [\![1, k]\!]$. Un tel déplacement est noté $In(i)$.
- les *déplacements de sortie* consistent à retirer l'élément en tête d'une file intermédiaire quelconque non vide F_i, avec $i \in [\![1, k]\!]$, pour le placer en queue de la file résultat. Un tel déplacement est noté $Out(i)$.

Le tri se termine lorsque tous les entiers se retrouvent dans la file résultat triés dans l'ordre croissant depuis la tête de file vers la queue de file. Rappelez vous que l'algorithme de tri prend en entrée une séquence quelconque sans restriction sur sa longueur. Aussi, dans tout le problème, on considère que les files intermédiaires ainsi que la file donnée et la file résultat sont de capacité infinie. On appelle scénario de tri une séquence de déplacements qui amène tous les éléments dans la file résultat rangés dans l'ordre croissant. Exemple : le scénario $[In(1), In(2), Out(2), Out(1), In(1), Out(1)]$ trie la séquence $[9, 3, 20]$ avec 2 files intermédiaires.
Le but de cet exercice est de prouver le théorème suivant et de donner un algorithme de tri.

Théorème *Pour trier une séquence S d'entiers strictement positifs sans répétitions dont la plus grande sous-séquence décroissante est de longueur k, k files en parallèle sont nécessaires et suffisantes. De plus, étant donné un réseau d'une infinité de files en parallèle, il existe un algorithme qui trie toute séquence d'entiers strictement positifs sans répétition S à l'aide de ce réseau en n'en utilisant que les k premières files intermédiaires.*

1. De combien de déplacements se compose un scénario de tri ? Justifiez.
2. Montrer que pour n'importe quel scénario de tri T, on peut considéer un scénario de tri T′ qui utilise exactements les mêmes déplacements mais qui soit tel que tous les déplacements d'entrée soient effectués avant tous les déplacements de sortie.

Un tel scénario T′ est dit normal. Dans toute la suite, on omettra de le préciser mais *on ne considère que des scénarios normaux.*

3. Montrez qu'à chaque étape d'un scénario de tri, chacune des files intermédiaires qui sont non-vides est triée dans l'ordre croissant.
4. Déduisez-en que, dans un scénario de tri, deux éléments qui sont inversés par S ne peuvent pas aller dans la même file intermédiaire. Et montrez que si S contient une sous-séquence décroissante de longueur k, avec $k \in \mathbb{N}^*$, il faut au moins k files en parallèle pour trier S.
5. Montrez que pour trier une séquence d'entiers S, il suffit de ranger tous les éléments de S dans les files intermédiaires de sorte que chacune de ces files soit triée dans l'ordre croissant.
   Donnez un algorithme (décrit en pseudo-code ou en langage courant) pour terminer le tri à partir de cette configuration. Veillez à n'utiliser que les opérations autorisées sur les files.
6. Quel est le nombre minimum de files nécessaire pour trier la séquence suivante ?

   $$S = [6, 3, 1, 7, 5, 9, 8, 2, 4]$$

   Dessiner l'état des files intermédiaires, dans un scénario de tri utilisant ce nombre minimum de files, juste après que tous les éléments soient sortis de la file donnée, et avant qu'aucun d'entre eux n'entre dans la file résultat.

Soit n un entier naturel, soit S une permutation de $[\![1, n]\!]$ et soit $a, b \in [\![1, n]\!]$. L'intervalle $[\![a, b]\!]$ est dit conservé par la permutation S si et seulement si $\{i \in [\![1, n]\!] \ t.q. \ a \leq s_i \leq b\}$ est un intervalle. L'intervalle $[\![1, n]\!]$ et les intervalles singletons $[\![a, a]\!]$, pour $a \in [\![1, n]\!]$, sont toujours conservés par toutes les permutations de $[\![1, n]\!]$, on les appelle les intervalles triviaux. Exemple : la permutation $S = [1, 3, 5, 4, 2]$ conserve l'intervalle $[\![3, 5]\!]$ car $\{i \in [\![1, 5]\!] \ t.q. \ 3 \leq s_i \leq 5\} = \{2, 3, 4\} = [\![2, 4]\!]$; par contre S ne conserve pas l'intervalle $[\![2, 4]\!]$ car $\{i \in [\![1, 5]\!] \ t.q. \ 2 \leq s_i \leq 4\} = \{2, 4, 5\}$ n'est pas un intervalle.

7. Donnez une permutation S de $[\![1; 9]\!]$ qu'il soit possible de trier avec 2 files en parallèle et telle que S ne conserve aucun intervalle non trivial.

On laisse de côté le cas particulier des permutations envisagé dans la question précédente pour revenir au cas général des séquences d'entiers strictement positifs sans répétitions, jusqu'à la fin de cette partie.

8. Dans cette question, on considère le cas où on dispose d'une infinité de files en parallèle $(F_i)_{i\in\mathbb{N}^*}$. Pour $i \in \mathbb{N}^*$, on note $Q(Fi)$ l'élément en queue de la file F_i lorsque celle-ci est non vide, et $Q(Fi) = 0$ sinon. Donnez un algorithme qui effectue tous les déplacements d'entrée et qui garantit les invariants suivants. A chaque étape de l'algorithme :

- pour tout $i \in N^*$, si F_i est non-vide alors F_i est triée en ordre croissant ; et
- la suite $(Q(F_i))_{i\in\mathbb{N}^*}$ est décroissante.

Veillez à n'utiliser que les opérations autorisées sur les files.

9. On note $S = [s_1, ..., s_n]$, avec $n \in \mathbb{N}^*$, la séquence fournie en entrée de l'algorithme. Montrez qu'un algorithme $\mathscr{A}$ vérifiant l'invariant de la question 8 vérifie également la propriété suivante : après le p-ième déplacement d'entrée, avec $1 \leq p \leq n$, pour tout $i \in \mathbb{N}^*$ tel que F_i est non vide, il existe une sous-séquence décroissante de la séquence $S_p = [s1, ..., sp]$ qui est de longueur i et dont le dernier élément est $Q(F_i)$.
Déduisez en que $\mathscr{A}$ utilise au plus k files pour trier S, où k est le maximum des longueurs d'une sous-séquence décroissante de S.

À partir d'ici, le sujet diffère légèrement de l'original.

10. Quelle est la complexité dans le pire des cas de l'algorithme que vous avez proposé à la question 8 en fonction de la longueur n de la séquence donnée et de la longueur k de sa plus longue sous-séquence décroissante ?
Expliquer quelles structures de données seraient nécessaires pour l'implémenter en C. Il n'est pas nécessaire d'expliquer comment implémenter à leur tour ces structures. Pensez à décrire l'implémentation de la partie de l'algorithme effectuant les déplacements de sortie.

Dans la question suivante on s'intéresse uniquement à améliorer la complexité de la partie de l'algorithme effectuant les déplacements d'entrée.

11. Comment implémenter la partie de votre algorithme traitant les déplacements d'entrée pour obtenir une complexité de $O(n \log k)$ dans le pire des cas, pour l'ensemble des déplacements d'entrée ? On considèrera que la longueur n de la séquence se trouvant initialement dans la file donnée vous est fournie au début de l'algorithme (en pratique, il suffit de parcourir la file donnée pour la déterminer, ce que vous n'avez pas à faire).

Dans les deux questions suivantes, on s'intéresse à la partie de l'algorithme traitant les déplacements de sortie.

12. (Impossible au Semestre1) ~~Dans cette question, et dans cette question seulement, vous avez droit à toute structure de donnée que vous jugez utile. Comment implémenter la partie de votre algorithme traitant les déplacements de sortie pour obtenir une complexité de $O(n \log k)$ dans le pire des cas ?~~

La question suivante vise à améliorer la complexité de la partie de l'algorithme traitant les déplacements de sortie dans le cas particulier où la séquence S fournie en entrée est une permutation.

13. Lorsque la séquence donnée S est une permutation, que peut-on faire lors des mouvements d'entrée pour pouvoir implémenter la partie effectuant les mouvements de sortie en temps $O(n)$ dans le pire des cas, pour l'ensemble des mouvements de sortie ? Comme précédemment, on considère que la longueur n de la permutation se trouvant initialement dans la file donnée vous est fournie au début de l'algorithme.

V - Humour

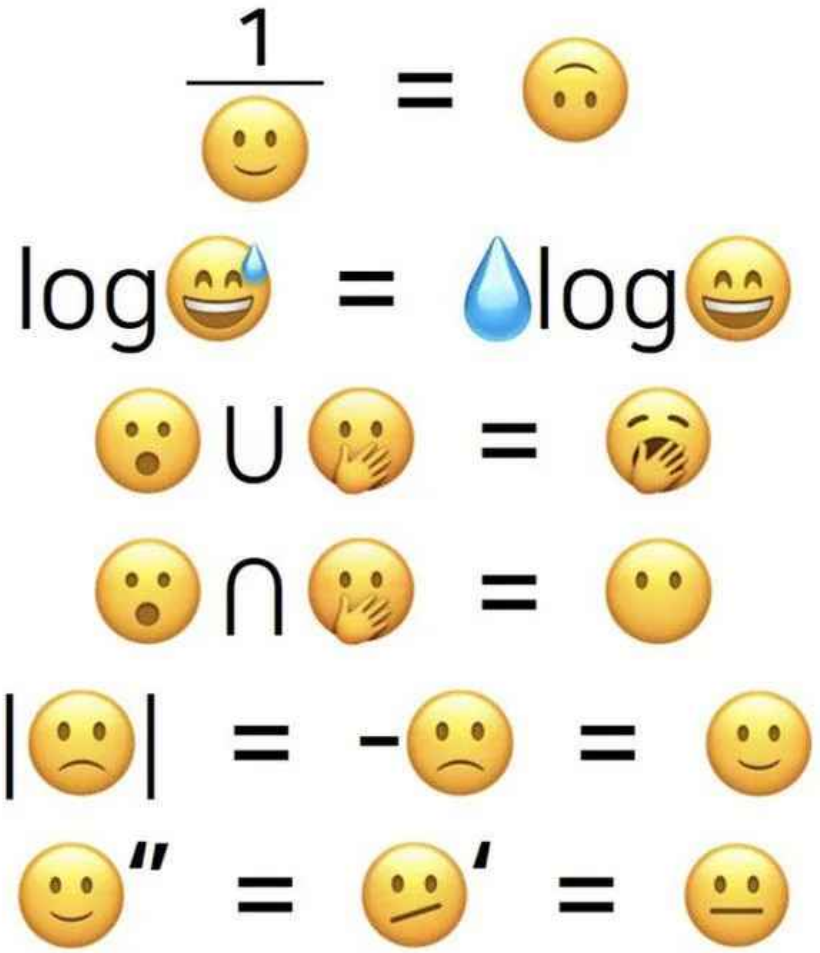

Figure IV.4 – Quand les calculs sur les caractères vont trop loin (et les pixels pas assez loin)