MP2I – DS2 2023-2024 – 4h

Pour chaque question, le langage imposé par l'énoncé doit être respecté.

Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.

Sauf précision contraire, les réponses doivent être justifiées.

En C, on suppose que les fichiers d'en-tête `stdbool.h`, `stdint.h`, `stdlib.h` *et* `stdio.h` *sont inclus.*

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac. Pas de montre connectée.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Tout le sujet est à traiter en C. Les trois parties sont complètement indépendantes.

PREMIÈRE PARTIE : DRAPEAU HOLLANDAIS

On considère l'algorithme suivant (proposé par Edsger Dijkstra en 1976) dit du drapeau hollandais [1] :

Algorithme 1 Algorithme du drapeau hollandais

entrée : un tableau T dont chaque case contient une couleur (bleu, blanc ou rouge), et sa longueur n
sortie : T forme le drapeau hollandais de bas en haut

```
1 : i1 ← 0 ; i2 ← n−1 ; i3 ← n−1
2 : tant que i1 ≤ i2 faire
3 :     si T[i1] est bleu alors
4 :         incrémenter i1
5 :     sinon
6 :         si T[i1] est blanc alors
7 :             échanger les couleurs de T[i1] et T[i2]
8 :             décrémenter i2
9 :         sinon
10 :            échanger les couleurs de T[i1] et T[i2]
11 :            échanger les couleurs de T[i2] et T[i3]
12 :            décrémenter i2
13 :            décrémenter i3
14 :        fin si
15 :    fin si
16 : fin tant que
```

Question 1 : Appliquer l'algorithme du drapeau hollandais au tableau

bleu	rouge	blanc	blanc	rouge	bleu

1. Cet énoncé étant en noir et blanc et au cas où je décris ici le drapeau hollandais, de bas en haut : bleu, blanc, rouge.

Correction :
Ici clairement on attend de voir l'évolution des couleurs dans les cases de T. Mettre la valeur initiale et la valeur finale sans les étapes intermédiaires ne sert à rien (la valeur finale est décrite dans l'énoncé). De même, une paraphrase de l'algorithme dans laquelle on ne voit pas l'évolution des données ne sert à rien.

0	1	2	3	4	5
i_1 ↓ bleu	rouge	blanc	blanc	rouge	i_2 i_3 ↓↓ bleu
	i_1 ↓ bleu			i_2 i_3 ↓↓	rouge
		i_1 ↓ rouge	i_2 ↓	blanc	
		i_2 ↓ blanc	i_3 ↓ blanc	rouge	
	i_2 ↓				

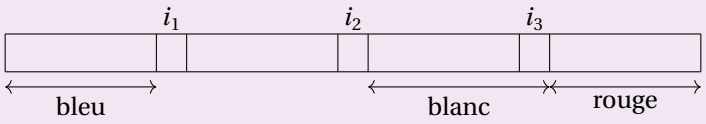
Question 2 : Prouver la terminaison de cet algorithme.

Correction :
Rappel 1 : l'existence d'un variant de boucle assure un nombre fini d'itérations, mais pas la terminaison de l'algorithme. Il faut s'assurer que chaque instruction se termine. Rappel 2 : le fait qu'un expression finisse par prendre la valeur nulle ne fait pas partie des propriétés d'un variant.
Les tests et instructions de l'algorithme s'exécutent tous en temps fini *(ici c'est évident, il n'y a pas besoin de rentrer dans les détails, mais il est indispensable de la dire)*.
La boucle tant que de la ligne 2 possède un nombre fini d'itérations car $i_2 - i_1$ est un variant pour cette boucle : c'est une expression entière, qui diminue strictement à chaque itération (car on passe soit par la ligne 4 qui incrémente i_1, soit par la ligne 8 ou la ligne 12 qui décrémentent i_2) et est positive car c'est la condition de sortie de boucle.
On peut conclure que l'algorithme du drapeau hollandais se termine.

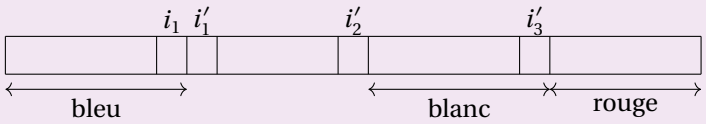
Question 3 : Prouver la correction de cet algorithme. On pourra montrer que le prédicat suivant est un invariant : *"Toute case d'indice strictement inférieur à i_1 est bleue, toute case d'indice strictement supérieur à i_2 et inférieur à i_3 est blanche et toute case d'indice strictement supérieur à i_3 est rouge"*.

Correction :
Bien sûr, l'énoncé suggère un invariant sans l'imposer. Faites preuve de modestie, et utilisez-le.
Notons (P) le prédicat de l'énoncé et prouvons que c'est un invariant pour la boucle de la ligne 2.
Remarquons tout d'abord que dans toute itération on a $i_1 \leq i_2 \leq i_3$, car :
- c'est vrai avant la boucle,
- si on n'a plus $i_1 \leq i_2$, on sort de la boucle (donc on ne rentre pas dans une itération),
- i_3 n'est décrémenté que si i_2 l'est (lignes 12 et 13).

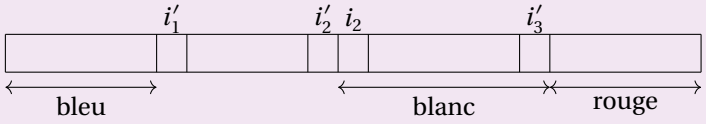
Avant la boucle, il n'y a pas de case d'indice i tel que : $i < i_1$ ou $i_2 < i \leq i_3$ ou $i > i_3$, le prédicat (P) est donc vérifié.
Supposons (P) vérifié au début une certaine itération et montrons qu'il l'ait encore à la fin de cette itération.
Notons i_1', i_2' et i_3' les valeurs respectives de i_1, i_2 et i_3 à la fin de l'itération (on garde la notation i_1, i_2 et i_3 pour le début de l'itération).
(P) étant vérifié en début d'itération, on est dans la situation suivante :



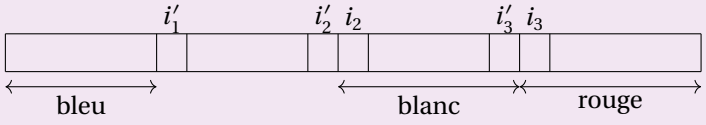
- si $T[i_1]$ est bleu, alors $i'_1 = i_1 + 1$, $i'_2 = i_2$ et $i'_3 = i_3$ et on se retrouve dans la situation :



  le prédicat (P) est donc vérifié ;
- si $T[i_1]$ est blanc, alors $i'_1 = i_1$, $i'_2 = i_2 - 1$, $i'_3 = i_3$ et $T[i'_2 + 1]$ est blanc, et on se retrouve dans la situation :



  le prédicat (P) est donc vérifié ;
- si $T[i_1]$ est rouge, alors $i'_1 = i_1$, $i'_2 = i_2 - 1$, $i'_3 = i_3 - 1$, $T[i'_2 + 1]$ est blanc et $T[i'_3 + 1]$ est rouge, et on se retrouve dans la situation :



  le prédicat (P) est donc vérifié.

Le prédicat (P) est donc un invariant pour la boucle de la ligne 2.

Une fois qu'on a prouvé l'existence d'un invariant, il faut conclure en faisant le lien avec la question initiale : il faut montrer pourquoi cet invariant assure la correction de l'algorithme.

On a prouvé à la question précédente que l'algorithme se termine. En sortie de boucle, on a $i_1 = i_2 + 1$ (car $i_2 - i_1$ est décrémenté de 1 à chaque itération comme vu à la question précédente) et $i_3 \geq i_2$. Le prédicat (P) est vérifié, on sait donc que toute case d'indice inférieur ou égal à i_2 est bleue, toute case d'indice entre $i_2 + 1$ et i_3 compris est blanche et toute case d'indice strictement supérieur à i_3 est rouge : le tableau T forme le drapeau hollandais. L'algorithme du drapeau hollandais possède donc une correction totale.

Question 4 : Prouver que la complexité dans le pire des cas de cet algorithme est en $\Theta(n)$.

Correction :

Ici il faut montrer que la complexité dans le pire des cas est en $O(n)$ et en $\Omega(n)$. On peut tout faire d'un coup car on est capable d'encadrer le nombre d'opérations élémentaires de chaque itération et on connaît le nombre d'itérations.

On peut considérer les tests de couleurs, les échanges de couleurs et les décrémentations d'indice comme des opérations élémentaires. *Attention à bien inclure une opération élémentaire qui permette de compter les passages par la ligne 4.*

À chaque itération, on a donc entre 2 opérations élémentaires (si on passe par la ligne 4) et 6 opérations élémentaires (si on passe par les lignes 10 à 13). Comme on l'a vu en prouvant la terminaison de l'algorithme, l'expression $i_2 - i_1$ est décrémentée de 1 à chaque itération ; sa valeur initiale est $n-1$ et sa valeur finale -1, il y a donc n itérations. Ce qui permet de conclure que la complexité de l'algorithme est en $\Theta(n)$. *Cet algorithme a la même complexité dans le pire des cas et dans le meilleur des cas.*

Question 5 : On a dit en cours que les algorithmes de tri qui n'utilisent que des comparaisons ont une complexité dans le pire des cas en $\Omega(n \log n)$ où n est la taille du tableau à trier. Expliquer pourquoi ce n'est pas contradictoire avec le résultat de la question précédente.

Correction :
Une partie non négligeable d'entre vous doit relire attentivement son cours sur la complexité – voir les commentaires sur les copies. Ici il n'y a pas de contradiction, car on ne se contente pas de comparer des valeurs, on se sert d'une information supplémentaire : il n'y a que trois valeurs possibles dans les cases.

DEUXIÈME PARTIE : NUAGE DE POINTS

On s'intéresse à un nuage de points (c'est-à-dire un ensemble de points) dans le plan, et on y cherche la paire de points les plus proches [2].

Tout au long du problème, on fait les trois hypothèses suivantes, que vous n'avez pas à vérifier, sur les nuages de points manipulés :

- les abscisses des points d'un nuage sont deux à deux distinctes,
- les ordonnées des points d'un nuage sont deux à deux distinctes,
- si le nuage de points comporte au moins deux points, alors une unique paire de points répond à la question.

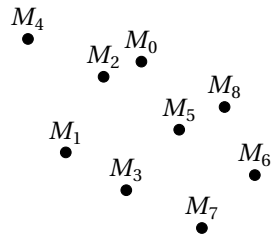
On représente un point par le type structuré suivant :

```
struct point {
  double x, y;
};
```

et un nuage de points par le type structuré :

```
struct nuage {
  int taille; // nombre de points
  struct point *P;
};
```

Exemple de nuage de points :



Cluster complet associé :

`nuage` : `N`

`taille` : 9

`abs` :	4	1	2	3	0	5	7	8	6
`ord` :	7	3	6	1	5	8	2	0	4

Sous-cluster $\{1,2,3\}$:

`nuage` : `N`

`taille` : 3

`abs` :	1	2	3
`ord` :	3	1	2

FIGURE 1 – Exemple de nuage de points et de deux clusters (cf. section 2) sur ce nuage.

1 Approche exhaustive naïve

Question 6 : Écrire et documenter une fonction `distance` qui prend en argument deux points et renvoie leur distance euclidienne (vous pouvez utiliser la fonction `sqrt` déclarée dans `math.h`).

Correction :
Pas de difficulté ici, il faut juste savoir calculer la distance entre deux points, ce que nous avons déjà fait plusieurs fois.

```
1 /** P1, P2 : deux points
2  * retour : leur distance euclidienne
3  */
4 double distance(struct point P1, struct point P2){
5   double dx = P2.x - P1.x, dy = P2.y - P1.y;
6   return sqrt(dx*dx + dy*dy);
7 }
```

2. Apparemment il s'agit d'un problème courant en navigation maritime.

Question 7 : Écrire et documenter une fonction

```
void plus_proche(struct nuage *N, int *p, int *q);
```

qui met à jour les entiers référencés par ses deux derniers paramètres, de sorte que ce soit les indices des deux points les plus proches. La recherche de ces points doit être faite de façon exhaustive en explorant toutes les paires de points.

Correction :
Je mets une solution qui utilise `assert`. Comme on n'a pas vraiment vu encore, vous pouviez juste sortir de la fonction si le nuage n'était pas assez grand. Mais il faut absolument être sûr qu'on a accès à une case avant d'y accéder.

```
1  /** N : pointeur sur nuage de points
2   * p, q : adresse d'entiers
3   * sortie : *p et *q sont les indices des deux points les plus proches dans le
        nuage
4   */
5  void plus_proche(struct nuage *N, int *p, int *q){
6    int min, candidat;
7
8    assert(N != NULL && p != NULL && q != NULL);
9
10   min = distance(N->P[0], N->P[1]);
11   *p = 0;
12   *q = 1;
13
14   for(int i=0; i<N->taille; i=i+1){
15     for(int j=i+1; j<N->taille; j = j+1){
16       candidat = distance(N->P[i], N->P[j]);
17       if(candidat<min){
18         min = candidat;
19         *p = i;
20         *q = j;
21       }
22     }
23   }
24 }
```

Question 8 : Donner, en la justifiant, la complexité de la fonction précédente.

Correction :
Prenons les affectations et les comparaisons comme opérations élémentaires.
Si on suppose que le calcul de la distance se fait en temps constant *(À noter que c'est vrai en pratique, l'algorithme utilisé est l'algorithme de Newton-Raphson très optimisé.)*, les instructions des lignes 16 à 21 se font en temps constant *(Pour rappel cela signifie en $\Theta(1)$, soit un encadrement par deux constantes.)*. En notant n la taille du nuage `N`, la complexité de la fonction est donc proportionnelle à :

$$\underbrace{\sum_{i=0}^{n-1}}_{\text{boucle ligne 14}}\ \underbrace{\sum_{j=i+1}^{n-1}}_{\text{boucle ligne 15}} 1 = \sum_{i=0}^{n-1}(n-1-i) = (n-1)^2 - \frac{(n-1)n}{2} = \Theta(n^2).$$

La complexité de la fonction `plus_proche` est donc quadratique en la taille du muage.

2 Méthode sophistiquée

On cherche maintenant à implémenter un algorithme qui suit la stratégie *"diviser pour régner"* pour résoudre ce problème. On est donc amené à considérer des sous-ensembles de nuages de points. Un tel sous-ensemble est appelé un *cluster*.
L'algorithme utilisé pour trouver la paire de points les plus proches dans un cluster est alors le suivant :

1. si le cluster est petit (deux ou trois points), on cherche les deux points les plus proches de façon exhaustive ;
2. sinon, on sépare le cluster en deux sous-clusters $\mathscr{G}$ et $\mathscr{D}$ de même taille (éventuellement à une unité près), suivant une médiane des abscisses [3]. Notons x_0 cette médiane. (Les deux points les plus proches sont alors soit tous les deux dans le même sous-cluster, soit chacun dans un sous-cluster.)
   (a) On calcule récursivement la paire de points les plus proches dans $\mathscr{G}$ et la paire de points les plus proches dans $\mathscr{D}$. On note δ la plus petite des deux distances obtenues et (P_1, P_2) le couple de points associés (P_1 et P_2 appartiennent donc tous les deux au même sous-cluster $\mathscr{G}$ ou $\mathscr{D}$).
   (b) On cherche s'il existe une paire de points $(G, D) \in \mathscr{G} \times \mathscr{D}$ dont la distance est strictement inférieure à δ.
   (c) Si on trouve une ou plusieurs paires de tels points, on garde la paire correspondant à la plus petite distance. Sinon on garde la paire (P_1, P_2).

Le principe "diviser pour régner" de cet algorithme est illustré en figure 2.

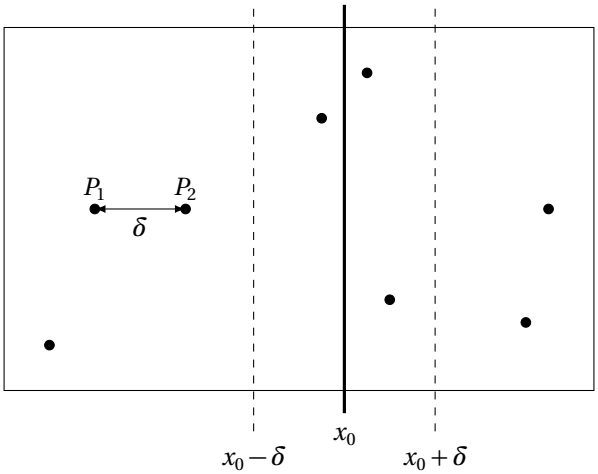


FIGURE 2 – Illustration du diviser pour régner.

Commençons par introduire quelques fonctions utilitaires.
Considérons la fonction :

```
1  /** tab : tableau à n cases
2   */
3  void tri(double *tab, int n){
4    int pos;
5
6    for(int i=0; i<n; i=i+1){
7      pos = i;
8      while(pos>0 && tab[pos]<tab[pos-1]){
9        int tmp = tab[pos];
10       tab[pos] = tab[pos-1];
11       tab[pos-1] = tmp;
12       pos = pos - 1;
13     }
14   }
15 }
```

Question 9 : Que fait cette fonction ? Le démontrer.

Correction :
Ce n'est pas parce que la question est formulée comme ça et pas "Montrer la correction totale de la fonction" que ça ne veut pas dire la même chose. Il faut utiliser les techniques vues en cours (qui ne s'appliquent pas toujours, mais très souvent sur les problèmes de concours : on ne réfléchit à autre chose que si ces techniques ne s'appliquent pas).
La fonction `tri` trie dans l'ordre décroissant les données du tableau `tab`.
Montrons sa terminaison. Les instructions des lignes 9 à 12 s'exécutent en temps constant. La boucle `while` de

3. On rappelle la définition d'une *médiane* d'un ensemble : élément tel qu'il existe autant d'éléments de l'ensemble qui lui sont supérieurs que d'éléments qui lui sont inférieurs (à une unité près).

la ligne 8 possède un nombre fini d'itérations car elle possède un variant (`pos` : expression entière positive qui diminue strictement à chaque itération), son exécution se termine donc. La boucle `for` de la ligne 6 possède un nombre fini d'opérations (variant : `n-i`) *(ici je ne rappelle pas que je connais la définition d'un variant : je viens de la donner)* et toutes ses instructions se terminent *(c'est immédiat, mais il faut le dire)*, donc elle se termine. *Pour prouver la terminaison de la boucle `for`, il faut avoir prouvé la terminaison de la boucle `while` : il faut donc commencer par cette dernière, même si elle arrive après dans le code, ça facilite la rédaction.*
Prouvons la correction de la fonction `tri`. *C'est une fonction itérative, on va donc raisonner en exhibant un invariant de boucle. Comme il y a deux boucles, il faut être très précis sur ce qu'on fait.*
Montrons tout d'abord que si les éléments des cases 0 à `pos-1` de `tab` sont dans l'ordre croissant à l'entrée de la boucle `while`, alors les éléments des cases 0 à `pos` de `tab` sont dans l'ordre croissant à la sortie de cette boucle.
On a l'invariant (I) : "Les cases 0 à `pos-1` de `tab` sont dans l'ordre croissant, ainsi que les cases `pos` à `i`, et la case `pos-1` de `tab` est inférieure à sa case `pos+1`." En effet :
avant la boucle : `pos` vaut `i`, donc (I) est vérifié.
hérédité : Supposons que (I) soit vérifié au début d'une itération. Si on rentre dans l'itération, c'est que la condition du `while` est vérifiée, en particulier le contenu de la case `pos` est strictement inférieur à celui de la case `pos-1`. L'itération inverse les contenus des desux cases et on peut en déduire que (I) est vérifié en sortie d'itération.
(I) est donc un invariant pour la boucle `while`. En sortie de boucle :

- si `pos`=0, (I) donne que les cases 0 à `i` de `tab` sont dans l'ordre croissant,
- sinon (I) donne que
  - les cases 0 à `pos-1` de `tab` sont dans l'ordre croissant et
  - les cases `pos` à `i` de `tab` sont dans l'ordre croissant,

  et la condition de sortie de boucle donne que le contenu de la case `pos-1` est inférieur au contenu de la case `pos`, donc les cases 0 à `i` sont dans l'ordre croissant.

Montrons maintenant que la boucle `for` possède l'invariant (J) : "Les `i` premières cases de `tab` sont triées dans l'ordre croissant.".
avant la boucle : `i` vaut 0, donc (J) est vérifié.
hérédité : Supposons que (J) soit vérifié au début d'une itération. Notons i la valeur de `i` au début de cette itération et i' sa valeur à la fin. On a montré qu'en sortie de la boucle `while`, les $i+1$ premières caes de `tab` sont triées dans l'ordre croissant. Comme $i+1=i'$, (J) est vérifié à la sortie de l'itération,
En sortie de boucle, i vaut `n`, donc (J) nous permet de conclure que les `n` premières cases de `tab` sont triées dans l'ordre croissant, et que la fonction `tri` possède une correction totale.

Question 10 : Donner, en la démontrant, la complexité de la fonction `tri` en fonction du paramètre `n`.

Correction :
La fonction `tri` est itérative, on raisonne donc en partant de la boucle la plus interne et en allant vers la boucle la plus externe.
On prend cmome opérations élémentaires les affectations et les tests.
Les instructions lignes 9 à 12 se font en temps contant et la boucle `while` de la ligne 8 possède au plus `i` itérations, le nombre d'opérations élémentaires de cette boucle est donc majoré par αi, pour une certaine constante $\alpha > 0$.
La complexité de la fonction `tri` est donc majorée par

$$\sum_{i=0}^{n-1}(1+\alpha i) = n + \alpha\frac{(n-1)n}{2},$$

soit une complexité dans le pire des cas en $O(n^2)$.
Cette complexité est atteinte par exemple si les cases du tableau sont initialement dans l'ordre décroissant : alors la boucle `while` possède exactement i itérations.
La complexité de la fonction tri dans le pire des cas est donc en $\Theta(n^2)$.

Question 11 : On souhaite trier les points d'un nuage de points suivant l'ordre des abscisses croissantes. Que faut-il changer à la fonction `tri` pour qu'elle réalise cette opération ? (On ne demande pas du code, juste une description.)

Correction :
Il faut adapter les types de la fonction et utiliser une comparaison entre les abscisses des points à la ligne 8.

Question 12 : Donner le nom et la complexité d'un algorithme de tri plus efficace dans le pire des cas (sans le programmer, ni faire la preuve de sa complexité).

Correction :
Le tri fusion possède une complexité dans le pire des cas en $\Theta(n \log n)$.

Le type structuré associé à un cluster est

```
struct cluster{
  struct nuage *N; // le nuage de points dont est extrait le cluster
  int taille;      // le nombre de points du cluster
  int *abs;        // les indices des points du cluster dans N,
                   // par abscisses croissantes
  int *ord;        // les indices des points du cluster dans N,
                   // par ordonnées croissantes
};
```

La figure 1 donne deux exemples de cluster.

On suppose qu'on dispose d'une fonction

```
struct cluster *cluster_complet(struct nuage *N);
```

qui renvoie le cluster contenant tous les points du nuage référencé par `N`.

Pour être efficace, on doit éviter de retrier des indices à chaque étape, mais plutôt extraire des informations du tri précédent.

Question 13 : Écrire et documenter une fonction

```
struct cluster *sous_cluster(struct cluster *c, double min, double max);
```

qui renvoie le sous-cluster de `c` composé des points de `c` dont l'abscisse est comprise entre `min` et `max` au sens large. Cette fonction doit avoir une complexité temporelle au plus linéaire en la taille du cluster `c`, complexité à justifier brièvement.

Correction :
Attention ici à ne pas confondre les indices et les abscisses : `min` et `max` sont des abscisses.

```
1  struct cluster *sous_cluster(struct cluster *c, double min, double max){
2    struct cluster*sc = (struct cluster *)malloc(sizeof(struct cluster));
3    sc->N = c->N;
4    sc->abs = NULL;
5    sc->ord = NULL;
6
7    //on cherche la taille du sous-cluster
8    sc->taille = 0;
9    bool dedans = false;
10   int g=c->taille; // indice du point le plus à gauche dans le sous-cluster
11   for(int i=0; i<c->taille; i=i+1){
12     if(c->N->P[c->abs[i]].x >= min && c->N->P[c->abs[i]].x <= max){
13       sc->taille = sc->taille + 1;
14       if (!dedans){ // c'est le point le plus à gauche du sous-cluster
15         dedans = true;
16         g = i;
17       }
18     }
19   }
20
21   if(sc->taille == 0){
22     return sc;
23   }
24
25   sc->abs = (int *)malloc(sc->taille * sizeof(int));
26   sc->ord = (int *)malloc(sc->taille * sizeof(int));
27   for(int i=0; i<sc->taille; i=i+1){
```

```
28      sc->abs[i] = c->abs[g+i];
29      sc->ord[i] = c->ord[g+i];
30    }
31    return sc;
32  }
```

Question 14 : Écrire et documenter une fonction

```
double mediane(struct cluster *c);
```

qui renvoie une abscisse médiane, c'est-à-dire l'abscisse x d'un des points du cluster référencé par `c`, telle que le nombre n_g de points du cluster d'abscisses strictement inférieures à x et le nombre n_d de points du cluster d'abscisses strictement supérieures vérifient $n_d \leq n_g \leq n_d + 1$. On supposera que le cluster contient au moins deux points. La complexité temporelle de cette fonction doit être constante (sans avoir à le justifier).

Correction :
Quand les structures deviennent compliquées, il faut absolument réfléchir en terme de type, ça évite beaucoup d'erreurs (surtout quand la partie algorithmique est immédiate). Et bien sûr il faut quand c'est possible utiliser ce qui a déjà été fait : ça simplifie souvent les réponses.

```
1  double mediane(struct cluster *c){
2    return c->N->P[c->abs[c->taille/2]].x;
3  }
```

Question 15 : Écrire et documenter une fonction

```
struct cluster *gauche(struct cluster *c);
```

qui renvoie le sous-cluster de `c` contenant la moitié des points de `c` les plus à gauche (moitié dont la taille sera éventuellement arrondie à l'entier supérieur si le cluster contient un nombre impair de points, et en supposant que le cluster référencé par l'argument contient au moins deux points, sans avoir à le vérifier).

Correction :

```
1  struct cluster *gauche(struct cluster *c){
2    return sous_cluster(c, c->abs[0], mediane(c));
3  }
```

On suppose qu'on dispose d'une fonction

```
struct cluster *droite(struct cluster *c);
```

qui renvoie le sous-cluster complémentaire (les points du cluster qui ne sont pas renvoyé par `gauche`).

Question 16 : Justifier que dans l'étape 2b de l'algorithme donné en début de section, on peut se contenter de chercher les points G et D de l'algorithme dans l'ensemble des points dont l'abscisse appartient à l'intervalle $[x_0 - \delta, x_0 + \delta]$.

Question 17 : Écrire et documenter une fonction

```
struct cluster *bande_centrale(struct cluster *c, double delta);
```

qui renvoie le sous-cluster de `c` correspondants aux points de `c` dont les abscisses se situent à distance au plus `delta` (au sens large) de la médiane des abscisses de `c`.

Correction :

```
1  struct cluster *bande_centrale(struct cluster *c, double delta){
2    double med = mediane(c);
3    return sous_cluster(c, med-delta, med+delta);
```

```
}
```

Question 18 : Par des considérations géométriques, on peut montrer qu'il n'y a pas plus de 6 points du cluster dans la bande d'abscisses $[x_0-\delta, x_0+\delta]$ dont l'ordonnée est strictement comprise entre les ordonnées de G et de D (x_0, δ, G et D sont définis dans l'étape 2 de l'algorithme en début de section) (résultat admis). Écrire et documenter une fonction

```
bool fusion(struct cluster *c, double delta, int *p, int *q);
```

qui prend en argument un pointeur sur un cluster et un nombre `delta` tels que tous les points du cluster ont une abscisse dans [`x0`—`delta`, `x0`+`delta`] (où `x0` est l'abscisse médiane du cluster `c`), sans avoir à le vérifier, et :

- renvoie `true` et met à jour les contenus de `p` et `q` de telle sorte qu'ils référencent les indices dans le nuage de points dont est extrait le cluster des deux points les plus proches du cluster, si leur distance est inférieure à `delta`,
- ou revoie `false`, sinon.

Cette fonction doit avoir une complexité linéaire en la taille du cluster (complexité à justifier).

Correction :

```
bool fusion(struct cluster *c, double delta, int *p, int *q){
  if(c->taille == 0 || c->taille == 1){
    return false;
  }
  *p = c->abs[0];
  *q = c->abs[1];
  double min_d = distance(c->N->P[c->abs[0]], c->N->P[c->abs[1]]);
  for(int i=1; i<c->taille-1; i=i+1){
    double d = distance(c->N->P[c->abs[i]], c->N->P[c->abs[i+1]]);
    if (d < min_d){
      min_d = d;
      *p = c->abs[i];
      *q = c->abs[i+1];
    }
  }
  return true;
}
```

Attention : la question demande de justifier la complexité. Surlignez en fluo la demande au moment de la lecture si vous êtes tête en l'air, pour ne pas oublier. Toutes les instructions du code se font en temps constant (en admettant à nouveau que le calcul de la distance se fait en temps constant) et le code contient une boucle dont le nombre d'itération est la taille du cluster moins 2, donc la complexité de la fonction est linéaire en cette taille.

Question 19 : Écrire et documenter une fonction

```
void distance_minimale(struct nuage *N, int *p, int *q);
```

qui met à jour les contenus de `p` et `q` en leur affectant les indices des deux points les plus proches dans le nuage référencé par `N`, en utilisant l'algorithme précédemment décrit.

Correction :

```
void distance_min(struct cluster *c, int *p, int *q){
  if(c->taille == 2){
    *p = c->abs[0];
    *q = c->abs[1];
    return;
  }
  //extraction des points les plus proches de la moitié gauche
  struct cluster *G = gauche(c);
  int pG, qG;
```

```
10    distance_min(G, &pG, &qG);
11    double deltaG = distance(G->N->P[G->abs[pG]], G->N->P[G->abs[qG]]);
12
13    //extraction des points les plus proches de la moitié droite
14    struct cluster *D = droite(c);
15    int pD, qD;
16    distance_min(D, &pD, &qD);
17    double deltaD = distance(D->N->P[D->abs[pD]], D->N->P[D->abs[qD]]);
18
19    //extraction des points les plus proches de la bande centrale
20    double delta;
21    if(deltaG < deltaD){
22      delta = deltaG;
23      *p = pG;
24      *q = qG;
25    } else {
26      delta = deltaD;
27      *p = pD;
28      *q = qD;
29    }
30    struct cluster *centre = bande_centrale(c, delta);
31    int pC, qC;
32    if(fusion(c, delta, &pC, &qC)){
33      *p = pC;
34      *q = qC;
35    }
36  }
37
38  void distance_minimale(struct nuage *N, int *p, int *q){
39    struct cluster *c = cluster_complet(N);
40    distance_min(c, p, q);
41  }
```

Question 20 : Si on note n la taille du nuage référencé par `N` dans la fonction précédente, et $C(n)$ le nombre d'opérations élémentaires réalisées par la fonction `distance_minimale`, justifier que l'on a :

$$C(n) = 2C(n/2) + \mathcal{O}(n).$$

Correction :
On va établir l'équation pour la fonction auxiliaire `distance_min`. On a deux appels récursifs (lignes 10 et 16) sur des clusters de taille la moitié du cluster fourni en argument et un appel à fusion (ligne 32) dont la complexité est linéaire d'après la question précédente. Les autres instructions se font en temps constant. D'où l'équation vérifiée par le nombre d'opérations élémentaires.
La fonction `distance_minimale` comporte une première étape de tri et on a vu qu'en utilisant un tri efficace cette première étape vérifie la même équation.

Question 21 : En déduire, en la démontrant, la complexité $C(n)$.

Correction :
On ne fait jamais de calculs avec des ordres de grandeur directement parce qu'on ne peut pas passer de n à $n+1$ ou à $2n$ dans un O par exemple. D'après la question précédente, il existe un $\alpha > 0$ tel que pour tout n, on a : $C(n) \leq 2C(n/2) + \alpha n$.
Faisons une somme téléscopique, en supposant que $n = 2^k$ pour un certain k :

$$\begin{aligned} C(2^k) &\leq 2C(2^{k-1}) + \alpha 2^k \\ 2C(2^{k-1}) &\leq 2^2 C(2^{k-2}) + \alpha 2^k \\ &\dots \\ \frac{2^{k-1}C(2) \leq 2^k C(1) + \alpha 2^k}{C(2^k) \leq 2^k C(1) + k\alpha 2^k} \end{aligned}$$

On a donc

$$C(2^k) = O(k2^k),$$

soit

$$C(n) = O(n \log n).$$

TROISIÈME PARTIE : INTERSECTION DE DEUX ENSEMBLES

Correction :
Cette partie a été tellement peu abordée, que je vous la laisse en DM.

Le but de cette partie est de calculer efficacement l'intersection de deux ensembles de points du plan discret. On commence par étudier l'intersection de deux ensembles de points de la droite discrète.

3 Ensembles d'entiers

Dans cette partie, on se propose de trouver une façon efficace de calculer l'intersection de deux ensembles d'entiers codés en `C` par le type :

```
struct ensemble {
  int cardinal;
  int *e;   // elements de l'ensemble
};
```

Question 22 : Écrire une fonction `C` qui prend en argument un tableau d'entiers, sa taille et une valeur et permet de tester si la valeur apparaît dans le tableau. Donner en la justifiant brièvement la complexité temporelle de votre fonction dans le pire des cas.

Question 23 : Écrire une fonction

```
int cardinal_intersection(struct ensemble *ens1, struct ensemble *ens2);
```

qui renvoie le cardinal de l'intersection des ensembles `ens1` et `ens2`, en testant pour chaque valeur de `ens1` si elle apparaît dans `ens2`. Donner en la justifiant la complexité temporelle de votre fonction dans le pire des cas.

Question 24 : Écrire une fonction

```
struct ensemble *intersection(struct ensemble *ens1, struct ensemble *ens2);
```

qui renvoie l'intersection des ensembles `ens1` et `ens2`, en suivant le même algorithme naïf que dans la question précédente. Donner en la justifiant brièvement la complexité temporelle de votre fonction dans le pire des cas.

Question 25 : On suppose maintenant que les champs `e` des deux instances de `struct ensemble` sont triés dans l'ordre croissant. Proposer un algorithme linéaire en la somme des longueurs des tableaux pour résoudre le même problème, en justifiant brièvement la complexité. (On ne demande pas de code.)

Question 26 : Quelle conclusion en tirer sur la façon la plus efficace de trouver l'intersection de deux ensembles d'entiers ?

4 Ensembles de points

On se propose maintenant de résoudre ce même problème d'intersection d'ensembles pour des ensembles de points du plan.

Soit n un entier naturel. On note D_n l'ensemble des entiers naturels compris entre 0 et $2^n - 1$ inclus :

$$D_n = [\![0, 2^n - 1]\!].$$

On appelle *point* de $D_n \times D_n$ tout couple d'entiers $(x, y) \in D_n \times D_n$.

On cherche à calculer efficacement l'intersection de deux ensembles de points de $D_n \times D_n$.[4]

Un point de coordonnées $(x, y) \in D_n \times D_n$ est représenté en C par le type `struct point` suivant[5] :

```
struct point {
  int x, y;
};
```

4.1 Une solution naïve en C

Dans cette section, on représente un ensemble de points par le type suivant :

```
struct ensemble {
  int cardinal;
  struct point *points; // points deux à deux distincts
};
```

Question 27 : Écrire une fonction

```
bool membre(struct point p, struct ensemble e);
```

qui teste si un point appartient à un ensemble de points. La complexité de cette fonction doit être linéaire en la cardinalité de l'ensemble (sans avoir à le justifier).

On pourrait suivre la même stratégie naïve que dans le cas de l'intersection d'ensembles d'entiers, qui aboutirait clairement à la même complexité que celle trouvée en question 24.
Il s'agit maintenant d'exploiter les idées mises en avant à la question 26 pour faire mieux.

4.2 Codage de Lebesgue

On souhaite implémenter en C une solution efficace au problème du calcul de l'intersection entre deux ensembles de points de $D_n \times D_n$ pour un certain n. En s'inspirant de la solution trouvée pour les ensembles d'entiers en section 4, on cherche à mettre un ordre total sur les points de $D_n \times D_n$.

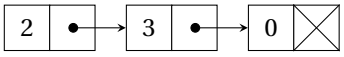
Notons $\overline{\cdots}^2$ la représentation binaire d'un entier. Ainsi, la représentation binaire de 6 est notée $\overline{110}^2$.
Le *codage de Lebesgue* d'un point de coordonnées $(x, y) \in D_n \times D_n$ s'obtient par entrelacement des bits des représentations binaires de x et y sur n bits, en commençant par un bit de x. On suppose que les bits de poids forts sont situés à gauche dans les représentations binaires des entiers naturels.
Par exemple, si $n = 3$, $x = 6 = \overline{\mathbf{110}}^2$ et $y = 2 = \overline{010}^2$, le codage de Lebesgue du point de coordonnées (x, y) est $\overline{\mathbf{101100}}^2$.
Le codage de Lebesgue d'un point peut être vu comme un nombre écrit dans la base formée des chiffres $\overline{\mathbf{00}}^2$, $\overline{\mathbf{01}}^2$, $\overline{\mathbf{10}}^2$ et $\overline{\mathbf{11}}^2$. Ainsi, si $n = 3$, le point $(6,2) = (\overline{\mathbf{110}}^2, \overline{010}^2)$ est codé par le nombre $\overline{\mathbf{10}}^2\overline{\mathbf{11}}^2\overline{\mathbf{00}}^2$.
On utilisera la notation décimale 0, 1, 2 et 3 pour représenter respectivement les chiffres $\overline{\mathbf{00}}^2$, $\overline{\mathbf{01}}^2$, $\overline{\mathbf{10}}^2$ et $\overline{\mathbf{11}}^2$, et on notera $\overline{c_{n-1} \dots c_0}^\ell$ la représentation en base 4 du codage de Lebesgue d'un point de $D_n \times D_n$ ainsi obtenue. Par exemple, pour $n = 3$, le codage du point $(6,2)$ s'écrit $\overline{230}^\ell$.

Question 28 : Si $n = 3$, donner la représentation en base 4 du codage de Lebesgue du point $(1,5)$.

En C, la séquence des chiffres d'un codage de Lebesgue c en base 4 d'un point de $D_n \times D_n$ est stocké dans une liste chaînée possédant n maillons et dont le maillon de tête contient le chiffre de poids fort, de type :

```
struct lebesgue {
  int chiffre;          // chiffre dans la représentation de Lebesgue
  struct lebesgue *s; // suite de la représentation
};
```

Ainsi, si $n = 3$, le codage de Lebesgue du point $(6,2)$ est représenté en C par la liste



On suppose que l'on dispose d'une fonction

4. Apparemment, la résolution de ce problème a des applications en simulation numérique, en robotique ou encore dans l'implémentation d'interfaces utilisateurs.
5. Attention, type différent du type de même nom défini en partie II.

```
int bits(int x, int k);
```

qui renvoie, en temps constant, la valeur du bit de coefficient 2^k dans le représentation binaire de x.

Question 29 : Écrire une fonction

```
struct lebesgue *code(int n, struct point p);
```

qui prend en argument un entier `n` strictement positif et un point `p` de $D_n \times D_n$ (sans avoir à le vérifier) et renvoie la représentation en base 4 du codage de Lebesgue de `p`. Cette fonction doit avoir une complexité linéaire en `n` (sans avoir à le justifier).

4.3 Représentation d'un ensemble de points

On utilise l'ordre lexicographique (autrement dit, l'ordre du dictionnaire) pour trier les codages : soit $c = \overline{c_{n-1}\dots c_0}^{\ell}$ et $d = \overline{d_{n-1}\dots d_0}^{\ell}$ des codages de Lebesgue de points de $D_n \times D_n$. On note $c < d$ pour « c est strictement plus petit que d » si

$$\exists i \in [\![0, n-1]\!], \quad (\forall j > i,\ c_j = d_j) \text{ et } c_i <_{\mathbb{N}} d_i$$

où $<_{\mathbb{N}}$ est l'ordre usuel sur les entiers naturels.

Question 30 : Trier les codages suivants par ordre croissant pour l'ordre lexicographique : $\{\overline{311}^{\ell}, \overline{000}^{\ell}, \overline{012}^{\ell}, \overline{101}^{\ell}, \overline{233}^{\ell}\}$.

Question 31 : Écrire une fonction

```
int compare_codages(int n, struct lebesgue *c1, struct lebesgue *c2);
```

qui prend en argument deux codages de Lebesgue de points de $D_n \times D_n$ (sans avoir à le vérifier) et renvoie 0 s'ils sont égaux, 1 si `c2` est plus grand que `c1` pour l'ordre lexicographique et -1 sinon. Cette fonction doit avoir une complexité linéaire en `n` (sans avoir à le justifier).

Nous allons maintenant représenter un ensemble S de points de $D_n \times D_n$ sous la forme d'une suite triée pour l'ordre lexicographique des codages de Lebesgue des points de S.

En guise d'exemple, nous allons coder l'ensemble de points $S_0 = \{(0,0),(1,0),(1,1),(2,2),(3,0),(0,1)\}$ de $D_2 \times D_2$. Ces points sont représentés en noir dans la figure 3 ci-dessous, dont l'origine est en bas à gauche, les abscisses croissent de gauche à droite et les ordonnées du bas vers le haut.

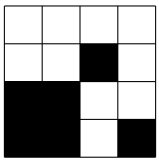


FIGURE 3 – L'ensemble de points S_0.

On considère d'abord les représentations binaires des coordonnées des points de S_0 :

$$\overline{S_0}^2 = \{(\overline{\mathbf{00}}^2, \overline{00}^2), (\overline{\mathbf{01}}^2, \overline{00}^2), (\overline{\mathbf{01}}^2, \overline{01}^2), (\overline{\mathbf{10}}^2, \overline{10}^2), (\overline{\mathbf{11}}^2, \overline{00}^2), (\overline{\mathbf{00}}^2, \overline{01}^2)\}$$

à partir desquelles on calcule le codage de Lebesgue de chaque point

$$\overline{S_0}^{\ell} = \{\overline{\mathbf{00}^2\mathbf{00}^2}, \overline{\mathbf{00}^2\mathbf{10}^2}, \overline{\mathbf{00}^2\mathbf{11}^2}, \overline{\mathbf{11}^2\mathbf{00}^2}, \overline{\mathbf{10}^2\mathbf{10}^2}, \overline{\mathbf{00}^2\mathbf{01}^2}\} = \{\overline{00}^{\ell}, \overline{02}^{\ell}, \overline{03}^{\ell}, \overline{30}^{\ell}, \overline{22}^{\ell}, \overline{01}^{\ell}\}.$$

Cet ensemble, une fois trié pour l'ordre lexicographique, s'écrit

$$\{\overline{00}^{\ell}, \overline{01}^{\ell}, \overline{02}^{\ell}, \overline{03}^{\ell}, \overline{22}^{\ell}, \overline{30}^{\ell}\}$$

Question 32 : On pose pour cette question $n = 2$. Donner la représentation sous forme de codage de Lebesgue trié par ordre lexicographique de l'ensemble de points

$$S_1 = \{(0,0),(3,3),(3,2),(1,1),(1,2),(2,2),(2,3)\}$$

On représente en `C` le Lebesgue trié par ordre lexicographique d'un ensemble de points par une liste chaînée de type :

```
struct suite {
  struct lebesgue *point;
  struct suite *s;
};
```

Question 33 : Donner une représentation graphique de S_0 en mémoire, sous la forme d'une instante de `struct suite`.

Question 34 : Montrer que l'ordre lexicographique sur les codages de Lebesgue de points de $D_n \times D_n$ est un ordre total. Conclure sur une méthode plus efficace que la méthode naïve pour calculer l'intersection de deux ensembles de points de $D_n \times D_n$.

4.4 Représentation plus compact d'un ensemble de points

Remarquons que nous venons d'effectuer un changement de système de coordonnées. Le codage de Lebesgue du point (x, y) représente le chemin à emprunter pour atteindre (x, y) dans l'espace récursivement divisé en quadrants.

1	3
0	2

FIGURE 4 – Numérotation des quadrants

En effet, en suivant la numérotation des quadrants donnée en figure 4, on s'aperçoit par exemple que le point de coordonnées $(1,1)$ dans le système de coordonnées usuel, dont le codage de Lebesgue est $\overline{03}^{\ell}$ est situé dans le quadrant 0 de $D_2 \times D_2$ et qu'à l'intérieur de ce quadrant subdivisé à son tour, le point $(1,1)$ est dans le quadrant 3, comme illustré en figure 5.

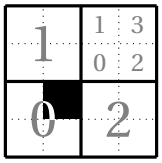


FIGURE 5 – Emplacement du point (1,1) de codage de Lebesgue $\overline{03}^{\ell}$: quadrant 0, sous-quadrant 3.

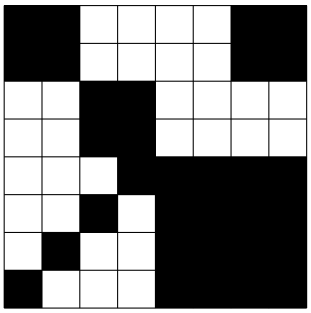
Formellement, pour $k < n$, le quadrant atteint dans $D_n \times D_n$ par le chemin $c = d_1 d_2 \cdots d_k$ (avec $d_i \in \{0,1,2,3\}$) est défini comme suit :

- Si k vaut 0 alors le chemin c est vide et le quadrant atteint dans $D_n \times D_n$ par c est l'ensemble des points de $D_n \times D_n$.
- Si $k > 0$ alors le chemin est de la forme $d_1 d_2 \cdots d_k$. Dans ce cas, le quadrant atteint par $d_1 d_2 \cdots d_k$ dans $D_n \times D_n$ est l'ensemble des points de $D_n \times D_n$ dont le codage de Lebesgue est de la forme $\overline{d_1 d_2 \ldots d_k c_{k+1} \ldots c_n}^{\ell}$ pour $c_{k+1}, \ldots, c_n \in \{0,1,2,3\}$.

Compactage par codage des quadrants.
Soit S un ensemble de points de $D_n \times D_n$ représenté par ordre lexicographique des codages de Lebesgue de ses points (comme en section 4.4). En se dotant d'un symbole supplémentaire, notons le 4, on compacte cette représentation en représentant chaque sous-séquence correspondant à un quadrant de chemin $d_1 \cdots d_k$ par l'unique mot $\overline{d_1 \cdots d_k \cdot 4 \cdots 4}^{\ell}$ (dans lequel on a rajouté $n-k$ fois le symbole 4) : si tous les points d'un sous-quadrant appartiennent à l'ensemble de points, plutôt que de les stocker un par un, on utilise cette représentation.
Notons qu'un codage de Lebesgue compacté représente *un ensemble de points* et non un unique point comme c'est la cas avec les codages de Lebesgue non compactés, par exemple, si $n = 2$, le codage $\overline{04}^{\ell}$ représente le quadrant 0 situé en bas à gauche de la figure 4 :

$$\overline{04}^{\ell} = \{\overline{00}^{\ell}, \overline{01}^{\ell}, \overline{02}^{\ell}, \overline{03}^{\ell}\}.$$

Ainsi, l'ensemble des points S_0 est compactable en $\{\overline{04}^{\ell}, \overline{22}^{\ell}, \overline{30}^{\ell}\}$.
On peut utiliser le type `struct lebesgue` pour coder en `C` un codage de Lebesgue compacté et par là même le type `struct suite` pour coder le codage de Lebesgue compacté par ordre lexicographique d'un ensemble de points.
Notons aussi que la liste des codages reste triée par ordre lexicographique.
Enfin, pour illustrer un cas où $n > 2$, la figure 6 décrit le codage compacté d'un ensemble de points de $D_3 \times D_3$.



est compacté en

$\{\overline{000}^{\ell}, \overline{003}^{\ell}, \overline{030}^{\ell}, \overline{033}^{\ell}, \overline{114}^{\ell}, \overline{124}^{\ell}, \overline{244}^{\ell}, \overline{334}^{\ell}\}$

FIGURE 6 – Un ensemble de points de $D_3 \times D_3$ et sa représentation compactée.

Structure de données d'AQL.
On appelle *AQL de l'ensemble de points S* la liste triée et compactée des codages de Lebesgue des points de l'ensemble S.

Question 35 : Donner l'AQL de l'ensemble S_1 de la question 32.

Question 36 : Écrire une fonction

```
struct lebesgue *ksuffixe(int n, int k, struct lebesgue *q);
```

qui prend en arguments un entier `n` strictement positif, un entier naturel `k` strictement inférieur à `n` une liste `q` représentant le codage de Lebesgue compacté d'un quadrant de $D_n \times D_n$ (sans avoir à vérifier ces conditions ; on admet en particulier qu'un 4 dans `q` ne peut être suivi que de 4, sans avoir à le vérifier) et :

- si les `k` derniers chiffres de la liste `q` ont pour valeur 4, cette fonction renvoie une nouvelle liste semblable à la liste `q` mais dont les `k+1` derniers chiffres valent 4,
- sinon, cette fonction renvoie `q`.

Ainsi, si `q1` représente $\overline{0144}^{\ell}$ et `q2` représente $\overline{0124}^{\ell}$, alors `ksuffixe(4, 2, q1)` renvoie une représentation de $\overline{0444}^{\ell}$ et `ksuffixe(4, 2, q2)` une représentation de $\overline{0124}^{\ell}$.

On suppose qu'on dispose d'une fonction

```
struct lebesgue *zip(int n, int k, struct lebesgue *q, struct lebesgue *p);
```

qui prend en argument un entier `n` strictement positif, un entier naturel `k` strictement inférieur à `n`, une liste `q` représentant le codage de Lebesgue compacté d'un quadrant de $D_n \times D_n$ et un maillon `p` de cette liste et qui :

- si le maillon `p` et les trois éventuels suivants appartiennent au même sous-quadrant de côté $2^k + 1$, remplace ces quatre maillons par un maillon représentant le sous-quadrant en question et renvoie le maillon de tête de la liste ainsi modifiée ;
- sinon renvoie le maillon de tête de la liste non modifiée.

Question 37 : L'algorithme de compactage d'une liste triée de codages de Lebesgue consiste à parcourir n fois la liste représentant l'ensemble de points. L'itération k vise à remplacer quatre codages successifs formant un quadrant complet de côté 2^{k+1} par la représentation compactée de ce quadrant.
Écrire une fonction

```
struct suite *compacte(int n, struct suite *S);
```

qui prend en arguments un entier strictement positif `n` et un ensemble de points S de $D_n \times D_n$ représenté par `s` et qui renvoie l'AQL de l'ensemble de points S.