

## TD 05 – Révisions (corrigé)

**(ElemMaj) Exercice 1.***Element majoritaire*

Soit  $E$  une liste de  $n$  éléments rangés dans un tableau numéroté de 1 à  $n$ . On suppose que la seule opération qu'on sait effectuer sur les éléments est de vérifier si deux éléments sont égaux ou non. On dit qu'un élément  $x \in E$  est *majoritaire* si l'ensemble  $E_x = \{y \in E \mid y = x\}$  a strictement plus de  $n/2$  éléments. Sauf avis contraire, on supposera que  $n$  est une puissance de 2. On s'intéressera à la complexité dans le pire des cas.

**1. Algorithme naïf**

Écrire un algorithme calculant le cardinal  $c_x$  de  $E_x$  pour un  $x$  donné. En déduire un algorithme pour vérifier si  $E$  possède un élément majoritaire. Quelle est la complexité de cet algorithme?

☞

**Algorithm 1: Naïf**

```

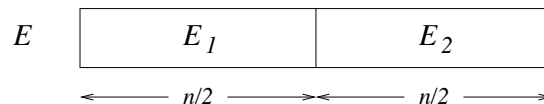
début
  pour i de 1 à n faire
    c ← 0;
    pour j de 1 à n faire
      si E[j] = E[i] alors
        c ← c + 1;
      si c > n/2 alors
        retourner "E[i] est majoritaire";
    retourner "Pas d'élément majoritaire".

```

Complexité : nombre total de comparaisons =  $n^2$ .

**2. Diviser pour régner**

- (a) Donner un autre algorithme récursif basé sur un découpage de  $E$  en deux listes de même taille. Quelle est sa complexité? ☞ 2.1 - Principe :



- Couper  $E$  en deux tableaux  $E_1$  et  $E_2$  de tailles  $n/2$  (on suppose  $n$  pair).
- S'il existe un élément majoritaire  $x$  dans  $E$ , alors  $x$  est majoritaire dans au moins une des deux listes  $E_1$  et  $E_2$  (en effet si  $x$  non majoritaire dans  $E_1$ , ni dans  $E_2$ , alors dans  $E$ ,  $c_x \leq n/4 + n/4 = n/2$ ).
- Algorithme récursif : calculer les éléments majoritaires de  $E_1$  et de  $E_2$  (s'ils existent) avec le nombre total d'occurrences de chacun et en déduire si l'un des deux est majoritaire dans  $E$ .

L'algorithme suivant *Majoritaire(i, j)* renvoie un couple qui vaut  $(x, c_x)$  si  $x$  est majoritaire dans le sous-tableau  $E[i..j]$  avec  $c_x$  occurrences et qui vaut  $(-, 0)$  s'il n'y a pas de majoritaire dans  $E[i..j]$ , l'appel initial étant *Majoritaire(1, n)*. La fonction *Occurrence(x, i, j)* calcule le nombre d'occurrences de  $x$  dans le sous-tableau  $E[i..j]$ .

**Algorithm 2: Majoritaire(i, j)**

```

début
  si i = j alors
    retourner (E[i], 1)
  sinon
    (x, c_x) = Majoritaire(i, ⌊(i+j)/2⌋);
    (y, c_y) = Majoritaire(⌊(i+j)/2⌋ + 1, j);
    si c_x ≠ 0 alors
      c_x ← c_x + Occurrence(x, ⌊(i+j)/2⌋ + 1, j);
    si c_y ≠ 0 alors
      c_y ← c_y + Occurrence(x, i, ⌊(i+j)/2⌋);
    si c_x > ⌊(j-i+1)/2⌋ alors
      retourner (x, c_x);
    sinon
      si c_y > ⌊(j-i+1)/2⌋ alors
        retourner (y, c_y);
      sinon
        retourner (-, 0);

```

**Complexité** : le nombre total de comparaisons dans le pire des cas  $C(n)$  vérifie la relation (on suppose que  $n$  est une puissance de 2) :

$$C(n) = 2(C(\frac{n}{2}) + \frac{n}{2}) \text{ avec } C(1) = 0$$

On en déduit  $C(n) = n \log_2(n)$  : avec le théorème général de résolution des récurrences du Cormen (Master Theorem dans le cours) :

Soient  $a \geq 1$  et  $b > 1$  deux constantes, soit  $f(n)$  une fonction et soit  $T(n)$  définie pour les entiers non négatifs par la récurrence  $T(n) = aT(n/b) + f(n)$ .  $T(n)$  peut alors être borné de la façon suivante.

1. Si  $f(n) = O(n^{\log_b a - \epsilon})$  pour une certaine constante  $\epsilon > 0$ , alors  $T(n) = \Theta(n^{\log_b a})$ .
2. Si  $f(n) = \Theta(n^{\log_b a})$ , alors  $T(n) = \Theta(n^{\log_b a} \log n)$ .
3. Si  $f(n) = \Omega(n^{\log_b a + \epsilon})$  pour une certaine constante  $\epsilon > 0$ , et si  $af(n/b) \leq cf(n)$  pour une certaine constante  $c < 1$  et pour tout  $n$  suffisamment grand, alors  $T(n) = \Theta(f(n))$ .

Ici  $f(n) = n = \Theta(n)$ , on est dans le deuxième cas, et  $C(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$ .

- (b) Même question quand  $n$  n'est pas une puissance de 2.  $\text{☞}$  Si  $n$  n'est pas une puissance de 2, les trois points du principe précédent restent vrais en coupant  $E$  en un tableau de taille  $\lfloor n/2 \rfloor$  et l'autre de taille  $\lceil n/2 \rceil$ , et l'algorithme décrit ci-dessus fonctionne sans aucune modification. La récurrence pour la complexité est alors  $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + n$  avec  $C(1) = 0$ , dont la résolution donne  $C(n) \sim n \log_2(n)$ .

### 3. Encore mieux

Pour améliorer l'algorithme précédent, on va se contenter dans un premier temps de mettre au point un algorithme possédant la propriété suivante :

- soit l'algorithme garantit que  $E$  ne possède pas d'élément majoritaire,
- soit l'algorithme fournit un entier  $p > n/2$  et un élément  $x$  tels que  $x$  apparaisse au plus  $p$  fois dans  $E$  et tout élément autre que  $x$  apparaît au plus  $n - p$  fois dans  $E$ .

- (a) Donner un algorithme récursif possédant cette propriété. Quelle est sa complexité?  $\text{☞}$  Algorithme récursif *Candidat – majoritaire*( $E$ ) renvoyant AUCUN s'il n'y a pas d'élément majoritaire dans  $E$  et sinon renvoyant  $(x, p)$  avec les bonnes propriétés.

---

#### Algorithm 3: *Candidat – majoritaire*( $E$ )

---

```

début
  si  $E$  n'a qu'un élément  $x$  alors
    retourner  $(x, 1)$ ;
  sinon
    couper  $E$  en deux tableaux  $E_1$  et  $E_2$  de taille  $n/2$ ;
    appeler Candidat – majoritaire( $E_1$ ) qui renvoie AUCUN ou  $(x, p)$ ;
    appeler Candidat – majoritaire( $E_2$ ) qui renvoie AUCUN ou  $(y, q)$ ;
    suivant la réponse pour  $E_1$  et  $E_2$ , faire;
    si AUCUN et AUCUN alors
      retourner AUCUN;
    si AUCUN et  $(y, q)$  alors
      retourner  $(y, q + \frac{n}{4})$ ;
    si  $(x, p)$  et AUCUN alors
      retourner  $(x, p + \frac{n}{4})$ ;
    si  $(x, p)$  et  $(y, q)$  alors
      si  $x \neq y$  alors
        si  $p > q$  alors
          retourner  $(x, p + \frac{n}{2} - q)$ ;
        si  $p < q$  alors
          retourner  $(y, q + \frac{n}{2} - p)$ ;
        si  $p = q$  alors
          retourner AUCUN;
          /* [
          h](sinon ce serait  $x$  ou  $y$  mais  $c_x \leq \frac{n}{2}$  et  $c_y \leq \frac{n}{2}$ ) */
      sinon
        /* [
        h] $x = y$  retourner  $(x, p + q)$ ;

```

**Complexité** : on a supposé  $n$  une puissance de 2, le nombre de comparaisons  $C(n)$  pour un tableau à  $n$  éléments vérifie :

$$C(n) = 2C(\frac{n}{2}) + 1 \text{ avec } C(1) = 0, \text{ ce qui donne } C(n) = n - 1 :$$

$$C_s = 2C_{s-1} + 1, \text{ donc } (E-2)(E-1)C_s = \bar{0}.$$

$$\text{D'où } C_s = k_1 2^s + k_2 = k_1 n + k_2, \text{ avec } C(1) = 0 \text{ et } C(2) = 1, \text{ donc } k_2 = -k_1 \text{ et } k_1 = 1.$$

- (b) Même question quand  $n$  n'est pas une puissance de 2.  $\text{☞}$  Si on ne suppose pas que  $n$  est une puissance de 2, couper  $E$  en deux tableaux de tailles  $\lfloor \frac{n}{2} \rfloor$  et  $\lceil \frac{n}{2} \rceil$ , et affiner l'analyse.

Si  $n$  est pair, faire comme à la question précédente en remplaçant juste  $\frac{n}{4}$  par  $\lceil \frac{n}{4} \rceil$ . La complexité vérifie alors  $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 1$ .

Si  $n$  est impair,  $n = 2m + 1$ , adapter la procédure précédente de la manière suivante :

---

```

début
  si  $m = 0$ ,  $E$  n'a qu'un élément  $x$  alors retourner  $(x, 1)$  ;
  sinon
    couper  $E$  en un tableau  $E_1$  de taille  $m$  et un tableau  $E_2$  de taille  $m + 1$  ;
    appeler  $Candidat - majoritaire(E_1)$  qui renvoie AUCUN ou  $(x, p)$  ;
    appeler  $Candidat - majoritaire(E_2)$  qui renvoie AUCUN ou  $(y, q)$  ;
    suivant la réponse pour  $E_1$  et  $E_2$ , faire
    si AUCUN et AUCUN alors retourner AUCUN ;
    si AUCUN et  $(y, q)$  alors retourner  $(y, q + \lceil \frac{m}{2} \rceil)$  ;
    si  $(x, p)$  et AUCUN alors retourner  $(x, p + \lceil \frac{m+1}{2} \rceil)$  ;
    si  $(x, p)$  et  $(y, q)$  alors
      si  $x \neq y$  et  $p \geq q$  alors retourner  $(x, p + m + 1 - q)$  ;
      si  $x \neq y$  et  $q \geq p + 1$  alors retourner  $(y, q + m - p)$  ;
      si  $x = y$  alors retourner  $(x, p + q)$ 

```

---

La complexité vérifie toujours  $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 1$  avec  $C(1) = 0$ . La résolution de cette formule de récurrence donne  $C(n) = n - 1$ .

- (c) En déduire un algorithme efficace vérifiant si  $E$  possède un élément majoritaire.  $\square$  Algorithme complet pour rechercher un majoritaire :
- Recherche d'un *candidat-majoritaire* avec l'algorithme précédent, soit  $n - 1$  comparaisons.
  - Si l'algorithme renvoie un candidat, vérification que le candidat est bien majoritaire en reparcourant le tableau, soit  $n - 1$  comparaisons.

Complexité au total =  $2n - 2$  comparaisons.

#### 4. Encore encore mieux

On change la manière de voir les choses. On a un ensemble de  $n$  balles et on cherche le cas échéant s'il y a une couleur majoritaire parmi les balles.

- (a) Supposons que les balles soient rangées en file sur une étagère, de manière à n'avoir jamais deux balles de la même couleur à côté. Que peut-on en déduire sur le nombre maximal de balles de la même couleur ?  $\square$  Soit  $k$  le nombre de balles rangées en file sur l'étagère telles que deux balles consécutives n'aient jamais la même couleur, alors le nombre de balles d'une même couleur est toujours  $\leq \lfloor \frac{k}{2} \rfloor$ .

Preuve précise : utiliser le *lemme des tiroirs*, c'est à dire "si  $M$  chaussettes sont rangées parmi  $m$  tiroirs avec  $M > m$ , alors il existe un tiroir avec au moins deux chaussettes". Ici découper l'étagère en  $\lfloor \frac{k}{2} \rfloor$  tiroirs disjoints qui sont des paires d'emplacements consécutifs sur l'étagère. S'il y a plus de  $\lfloor \frac{k}{2} \rfloor$  balles d'une même couleur, il en existe deux dans un même tiroir, c'est à dire ici adjacentes sur l'étagère.

On a un ensemble de  $n$  balles, une étagère vide où on peut les ranger en file et une corbeille vide. Considérons l'algorithme suivant :

- **Phase 1** - Prendre les balles une par une pour les ranger sur l'étagère ou dans la corbeille. Si la balle n'est pas de la même couleur que la dernière balle sur l'étagère, la ranger à côté, et si de plus la corbeille n'est pas vide, prendre une balle dans la corbeille et la ranger à côté sur l'étagère. Sinon, c'est à dire si la balle est de la même couleur que la dernière balle sur l'étagère, la mettre dans la corbeille.

- **Phase 2** - Soit  $C$  la couleur de la dernière balle sur l'étagère à la fin de la phase 1. On compare successivement la couleur de la dernière balle sur l'étagère avec  $C$ . Si la couleur est la même on jette les deux dernières balles sur l'étagère, sauf s'il n'en reste qu'une, auquel cas on la met dans la corbeille. Sinon on la jette et on jette une des balles de la corbeille, sauf si la corbeille est déjà vide auquel cas on s'arrête en décrétant qu'il n'y a pas de couleur majoritaire. Quand on a épuisé toutes les balles sur l'étagère, on regarde le contenu de la corbeille. Si elle est vide alors il n'y a pas de couleur majoritaire, et si elle contient au moins une balle alors  $C$  est la couleur majoritaire.

- (b) (*Correction de l'algorithme*) Montrer qu'à tout moment de la phase 1, toutes les balles éventuellement présentes dans la corbeille ont la couleur de la dernière balle de l'étagère. En déduire que s'il y a une couleur dominante alors c'est  $C$ .

Prouver la correction de l'algorithme.  $\square$  *Correction de l'algorithme* : on montre des invariants (propriétés restant vraies pendant une phase de l'algorithme).

- **Phase 1** -

**Invariant 1** : si la corbeille n'est pas vide, toutes les balles dans la corbeille ont la même couleur que la dernière balle sur l'étagère.

**Preuve** : montrer que si la prop. est vraie à un instant donné, elle reste vraie à l'étape suivante. Plusieurs cas peuvent se produire ici, vérifier la conservation de la propriété dans chaque cas (à faire) :

- ajout d'une balle différente de la dernière et corbeille pleine ...
- ajout d'une balle différente de la dernière et corbeille vide ...
- ajout d'une balle identique à la dernière ...


**Invariant 2** : sur l'étagère, il n'y a jamais deux balles de même couleur côte à côte.

**Preuve** : idem au cas par cas.

**Fin de Phase 1** : soit  $C$  la couleur de la dernière balle sur l'étagère. Les balles d'autres couleurs sont dans les  $k - 1$  premières balles de l'étagère, où  $k$  est le nombre de balles sur l'étagère et  $k \leq n$ . Comme il n'y a pas deux balles adjacentes de même couleur, d'après la question 1, le nombre de balles d'une couleur différente de  $C$  est  $\leq \lceil \frac{n-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ . Donc la seule couleur candidate pour être majoritaire est  $C$ .

- **Phase 2** - vérifie si  $C$  est bien majoritaire :

- Quand on retire une paire de balles, on en a toujours une de couleur  $C$  et l'autre différente, donc  $C$  est majoritaire si et seulement si une majorité de balles à la fin (étagère + corbeille) est de couleur  $C$ .
- Deux cas de fin se présentent :
  - La phase 2 s'arrête car on a besoin d'une balle de la corbeille mais la corbeille est vide. Dans ce cas, la dernière balle sur l'étagère n'est pas de couleur  $C$  et d'après la question 1, au plus la moitié des balles restant sur l'étagère sont de couleur  $C$ , il n'y a pas de majorité. Ok, c'est bien ce que répond l'algorithme.
  - L'algorithme va jusqu'au bout de l'étagère, les balles restantes (s'il en reste) sont dans la corbeille, elles sont toutes de couleur  $C$ . Donc  $C$  est majoritaire si et seulement si la corbeille est non vide, c'est bien le test effectué par l'algorithme.

(c) (**Complexité**) Donner la complexité dans le pire des cas en nombre de comparaisons de couleurs des balles.  **Complexité** :

Phase 1 :  $(n - 1)$  comparaisons ( $-1$  car la première balle est directement posée sur l'étagère).

Phase 2 : une comparaison par paire de balles éliminées, plus éventuellement une comparaison avec  $C$  pour la dernière balle de l'étagère s'il en reste une, soit  $\leq \lceil \frac{n}{2} \rceil - 1$  comparaisons ( $-1$  car au début de la phase 2, on sait que la dernière balle sur l'étagère est de couleur  $C$ ).

Complexité totale  $\leq \lceil \frac{3n}{2} \rceil - 2$  comparaisons.


## 5. Optimalité

On considère un algorithme de majorité pour les couleurs de  $n$  balles. Le but est de regarder faire l'algorithme, en choisissant au fur et à mesure les couleurs des balles pour que l'algorithme ait le maximum de travail (tout en restant cohérent dans le choix des couleurs). On obtiendra ainsi une borne inférieure de complexité pour un algorithme de majorité. C'est la technique de l'*adversaire*. À tout moment de l'algorithme, on aura une partition des balles en deux ensembles : l'*arène* et les *gradins*. L'*arène* contient un certain nombre de composantes connexes de deux sortes : les *binômes* et les *troupeaux*. Un binôme est un ensemble de deux balles pour lesquelles l'algorithme a déjà testé si elles étaient de la même couleur et a répondu non. Un troupeau est un ensemble non vide de balles de la même couleur, connectées par des tests de l'algorithme. Ainsi, un troupeau avec  $k$  éléments a subi au moins  $k - 1$  comparaisons de couleurs entre ses membres. Soient  $B$  le nombre de binômes et  $T$  le nombre de troupeaux. Soient  $g$  le nombre d'éléments dans les gradins et  $t$  le nombre total d'éléments dans tous les troupeaux. Enfin, soit  $m = \lfloor n/2 \rfloor + 1$  le "*seuil de majorité*". Au début de l'algorithme toutes les balles sont des troupeaux à un élément. La stratégie de l'*adversaire* est la suivante. L'algorithme effectue un test  $couleur(x) = couleur(y)$ ?

1. Si  $x$  ou  $y$  sont dans les gradins, la réponse est non.
2. Si  $x$  (resp.  $y$ ) est dans un binôme, la réponse est non et  $x$  (resp.  $y$ ) est envoyé dans les gradins alors que l'élément restant devient un troupeau singleton.
3. Si  $x$  et  $y$  sont dans le même troupeau la réponse est oui.
4. Si  $x$  et  $y$  sont dans des troupeaux différents alors cela dépend de  $d = B + t$ .
  - (a)  $d > m$  : cela signifie que les balles sont dans des troupeaux singletons. La réponse est non et les balles deviennent un nouveau binôme.
  - (b)  $d = m$  : la réponse est oui et les troupeaux de  $x$  et  $y$  fusionnent.

(a) Vérifier que les quatre cas précédents traitent tous les cas possibles.

Montrer qu'à tout moment  $d \geq m$  et que si  $d > m$  alors tous les troupeaux sont des singletons.

 Tous les cas possibles sont bien représentés. Au départ  $d = n$  et tous les troupeaux sont des singletons. Quels cas modifient  $d$ ?

Cas 1 : non.

Cas 2 : non (car  $-1$  binôme,  $+1$  dans les troupeaux).

Cas 3 : non.

Cas 4(a) : oui,  $d$  diminue de 1 (mais les troupeaux restent des singletons).

Cas 4(b) : non.

Conclusion :

- Invariant  $d \geq m$  (si  $d$  change, c'est que  $d > m$  avec le cas 4(a) et alors  $d - 1 \geq m$ ).
- $d > m$  implique que tous les troupeaux sont des singletons.

(b) Montrer qu'à tout moment les deux coloriage suivants sont cohérents avec les réponses de l'*adversaire* :

- (i) Toutes les balles sont de couleurs différentes sauf celles qui sont dans un même troupeau.  
(ii) Une même couleur est attribuée à toutes les balles de tous les troupeaux et à une balle de chaque binôme. Les balles restantes ont chacune une couleur distincte.

☞ Cohérence des réponses de l'adversaire à tout moment.

Coloration (i) : correspond bien aux réponses données, car respecte les binômes (*comparaisons inégales*) et les troupeaux (*comparaisons égales*). De plus les balles envoyées dans les gradins ont toujours engendré la réponse NON aux comparaisons, ce qui fonctionne si leurs couleurs sont toutes différentes et différentes des couleurs des balles dans l'arène.

Coloration (ii) : idem avec de plus le fait qu'une balle dans un binôme n'a jamais été comparée à quelqu'un d'autre que son binôme (donc pas de risque de contradiction avec les choix des couleurs ailleurs).

- (c) Montrer que si un algorithme correct s'arrête alors l'arène ne contient qu'une seule composante connexe qui est un troupeau de taille  $m$ . ☞ Un algorithme correct ne peut s'arrêter que si l'arène ne contient qu'une composante connexe qui sera un troupeau de taille  $m$ .

**Preuve par l'absurde** : supposons que l'arène contienne au moins deux composantes (ce qui implique  $n \geq 2$  et  $m \geq 2$ ). Par définition de  $d$ , chaque troupeau contient  $< d$  balles. Chaque troupeau contient donc  $< m$  balles, car soit  $d = m$ , soit  $d > m$  et d'après la question 1 chaque troupeau est un singleton. La coloration (i) n'a pas d'élément majoritaire, mais la coloration (ii) a un élément majoritaire (car  $d \geq m$ ). Donc aucun algorithme correct ne peut s'arrêter à cet instant.

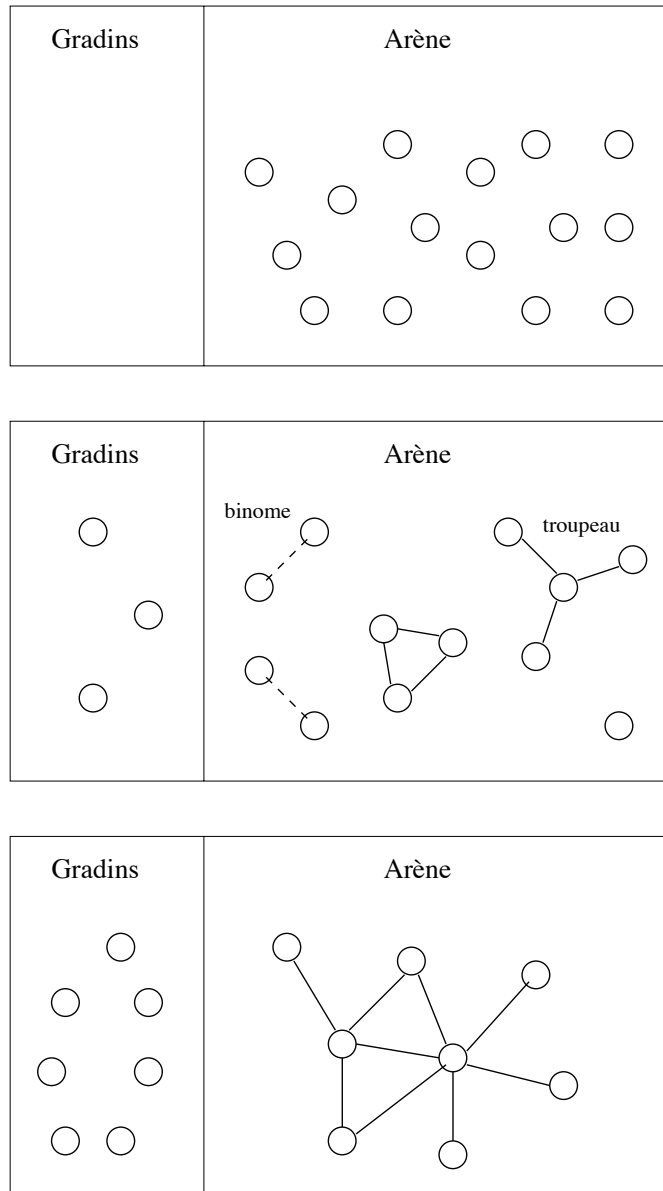
**Conclusion** : à la fin, il y a nécessairement une unique composante, nécessairement de taille  $d = m$  (par définition de  $d$  et d'après la question 1 qui implique que  $d \neq m$ ).

- (d) À tout moment le nombre de comparaisons inégales effectuées par l'algorithme est au moins  $2g + B$  et le nombre de comparaisons égales est au moins  $t - T$ . ☞ A tout moment, on a eu jusqu'à présent :

- Nombre de *comparaisons inégales*  $\geq 2g + B$ , car il faut 2 comparaisons inégales pour mettre une balle dans les gradins (entrée dans un binôme puis sortie du binôme) et il y a  $B$  comparaisons inégales qui ont construit les binômes en place.
- Nombre de *comparaisons égales*  $\geq t - T$ , car un troupeau  $i$  avec  $t_i$  balles est connexe donc il a  $\geq t_i - 1$  arêtes, soit ici  $t_i - 1$  comparaisons égales.

- (e) Considérons un algorithme qui résout la majorité. Montrer qu'il existe une donnée pour laquelle l'algorithme effectue au moins  $2(n - m) = 2\lceil n/2 \rceil - 1$  comparaisons d'inégalité et au moins  $\lfloor n/2 \rfloor$  comparaisons d'égalité, et donc au moins au total  $3\lceil n/2 \rceil - 2$  comparaisons.

☞ En fin d'algorithme, d'après la question 3,  $g = n - m$ ,  $B = 0$ ,  $t = m$  et  $T = 1$ . Donc, avec la question 4, il y a eu  $\geq 2(n - m) = 2\lceil \frac{n}{2} \rceil - 2$  comparaisons inégales et  $\geq m - 1 = \lfloor \frac{n}{2} \rfloor$  comparaisons égales, soit au total  $\geq \lceil \frac{3n}{2} \rceil$  comparaisons.



(plssc) **Exercice 2.**

*Plus longue sous suite croissante*

On considère une suite de  $n$  entiers :  $a_1, a_2, \dots, a_n$ . Une sous séquence est un sous ensemble d'entiers de la suite, pris dans le même ordre que la suite :  $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ , avec  $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq n$ . Une sous séquence croissante est une sous séquence où les entiers sont strictement croissants. Par exemple, la plus grande sous séquence croissante de la suite 5, 2, 8, 6, 3, 6, 9, 7 est 2, 3, 6, 9.

1. Donnez une plus longue sous séquence croissante de 11, 6, 2, 24, 25, 12, 21, 41, 34, 30.

☞

Par exemple 11, 24, 25, 41. Longueur max = 4 ;

2. Donnez un algorithme de programmation dynamique permettant de trouver la plus grande sous séquence croissante d'une suite de  $n$  entiers et donnez sa complexité.

☞

**Algorithme.** Pour que deux éléments  $i$  et  $j$  de la suite soient dans la solution, il faut que  $i < j$  (on respecte l'ordre de la suite) et  $a_i < a_j$  (on respecte le fait que la séquence soit strictement croissante).

L'idée est de construire un graphe contenant les transitions possibles, i.e., une arête ne peut exister entre deux nœuds  $i$  et  $j$  que si  $i < j$  et  $a_i < a_j$ . Notre but est alors de trouver le chemin le plus long dans ce graphe.

$Long[j]$  est la longueur du plus long chemin se terminant sur le nœud  $j$ .  $Pred[j]$  contient le nœud précédant  $j$  dans le plus long chemin, ce tableau permet de reconstruire la solution.

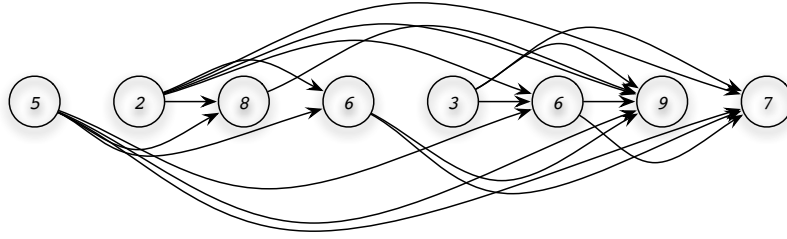


FIGURE 1 – DAG pour le problème de la plus longue sous séquence croissante.

---

**Algorithm 4:** Plus grande sous séquence croissante

---

```

début
  Long[1...n] ← [1...1];
  Pred[1...n] ← [1...n];
  pour i = 1 à n faire
    pour j = 1 à i - 1 faire
      si ai > aj alors
        Long[j] ← max{Long[j], 1 + Long[i]};
        si Long[j] = 1 + Long[i] alors
          Pred[j] = i;
  retourner maxj{Long[j]}

```

---

**Complexité.** On a une complexité en  $O(n^2)$ .

3. On peut cependant faire mieux que par programmation dynamique. Proposez un algorithme qui tourne en  $O(n \log n)$ .

☞

**Algorithme.** On utilise les variables suivantes :

- $Pred[i]$  est le prédécesseur de  $i$  dans la plus longue sous séquence terminant à  $i$ .
- $L$  est la longueur du plus long chemin.
- $M[l]$  est la position  $j$  du plus petit élément  $a_j$  tel que  $j \leq i$ , et il y a une séquence croissante de taille  $l$  terminant en  $a_j$ . La séquence  $a_{M[1]}, a_{M[2]}, \dots, a_{M[L]}$  est croissante.

L'idée est de traiter les éléments dans l'ordre en gardant à chaque étape la plus grande sous séquence trouvée, puis de rechercher par dichotomie la plus grande sous séquence telle qu'on puisse insérer l'élément en cours.

---

**Algorithm 5:** Plus grande sous séquence croissante en  $O(n \log n)$

---

```

début
  L ← 0;
  M[0] ← 0;
  pour i = 1 à n faire
    Faire une dichotomie pour trouver le plus grand  $l \leq L$ , tel que  $a_{M[l]} < a_i$  (ou alors  $l \leftarrow 0$  si ce  $l$  n'existe pas);
    Pred[i] ← M[l];
    si j == L ou  $a_i < a_{M[l+1]}$  alors
      M[l + 1] ← i;
      L ← max{L, l + 1};
  retourner L

```

---

**Complexité.** Il faut  $O(\log n)$  pour chaque dichotomie, et on boucle sur tous les éléments, soit une complexité en  $O(n \log n)$ .