EA4 – Éléments d'algorithmique II
Devoir maison n° 2 – groupe INFO 3
(Correction)

Exercice 1 : table des suffixes

1. Choisir un mot m de plus de 12 lettres.
   ▷ *Par exemple* `algorithmique` *(13 lettres).*

2. Proposer une méthode naïve permettant de tester si le mot `ment` est un facteur de m.
   ▷ *Pour chaque position i entre 0 et* `len`$(m)-5$*, tester si* `ment` *est un facteur à cette position en comparant les caractères les uns à la suite des autres.*

3. Combien de comparaisons de caractères votre méthode demande-t-elle ?
   ▷ *Dans le cas général, au plus $F \times (M - F + 1)$ si F est la taille du facteur à chercher et M celle du mot m. Dans notre exemple, 11, 1 pour chacune des 10 premières positions comparées à* `m` *plus 1 comparaison du* `e` *au* `i`*.*

4. Construire la table des suffixes du mot m.
   ▷ *La table des suffixes est donnée par* `T` *dans le tableau suivant :*

i	`T`$[i]$	*Suffixe correspondant*
0	*0*	`algorithmique`
1	*12*	`e`
2	*2*	`gorithmique`
3	*7*	`hmique`
4	*9*	`ique`
5	*5*	`ithmique`
6	*1*	`lgorithmique`
7	*8*	`mique`
8	*3*	`orithmique`
9	*10*	`que`
10	*4*	`rithmique`
11	*6*	`thmique`
12	*11*	`ue`

   Éventuellement on peut ajouter le mot vide ϵ comme tout premier suffixe.

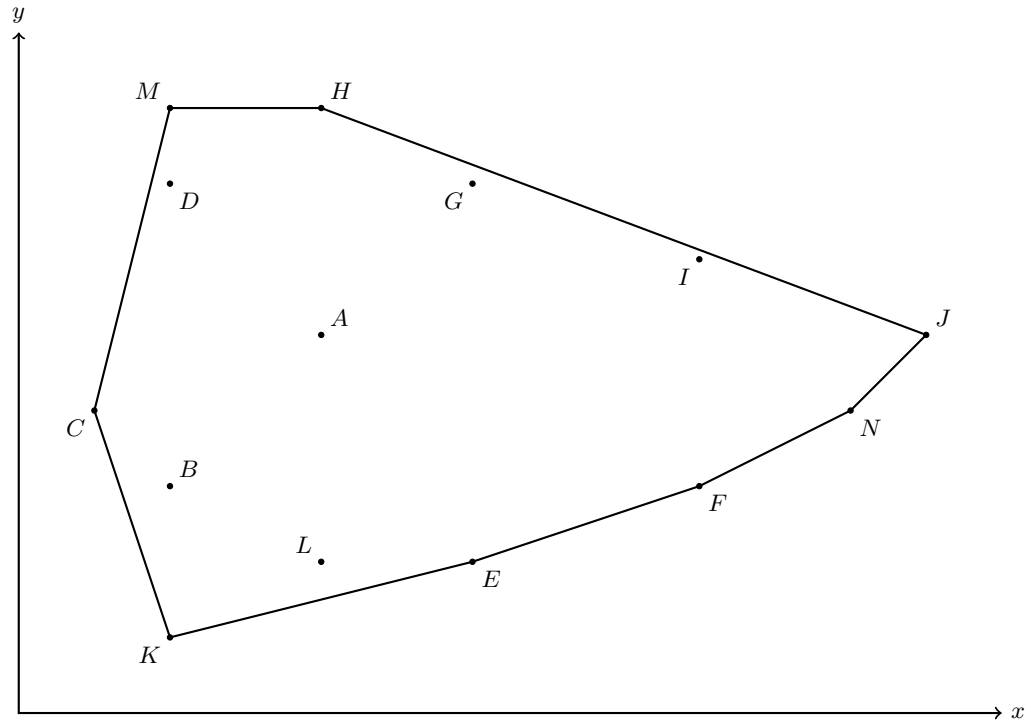
5. Expliquer comment, à partir de la table des suffixes, déterminer rapidement si le mot `ment` est un facteur de m.
   ▷ *Par recherche dichotomique dans la table des suffixes.*

6. Combien de comparaisons de caractères cela demande-t-il ? Détailler le calcul.
   ▷ *Dans le cas général au plus $F \times \lceil \log_2 N \rceil$.*
   Dans notre exemple, 4, 1 pour le premier caractère de `lgorithmique` *(indice 6 milieu de $[0, 12]$),* `que` *(indice 9 milieu de $[7, 12]$) et* `mique` *(indice 7 milieu de $[7, 8]$) comparés à* `m` *plus 1 pour le second caractère de* `mique` *comparé à* `e`*.*

Exercice 2 : enveloppe convexe

Soit $\mathcal{N}$ l'ensemble des points $A(4,5), B(2,3), C(1,4), D(2,7), E(6,2), F(9,3), G(6,7), H(4,8), I(9,6), J(12,5), K(2,1), L(4,2), M(2,8)$ et $N(11,4)$.

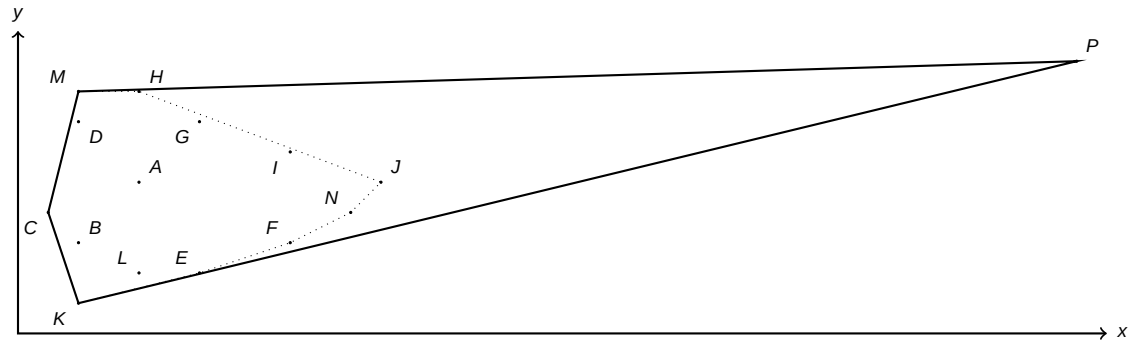
1. Calculer l'enveloppe convexe de $\mathcal{N}$ (dessin clair accepté).
   ▷ *Les sommets de l'enveloppe convexe sont $\{K, E, F, N, J, H, M, C\}$*



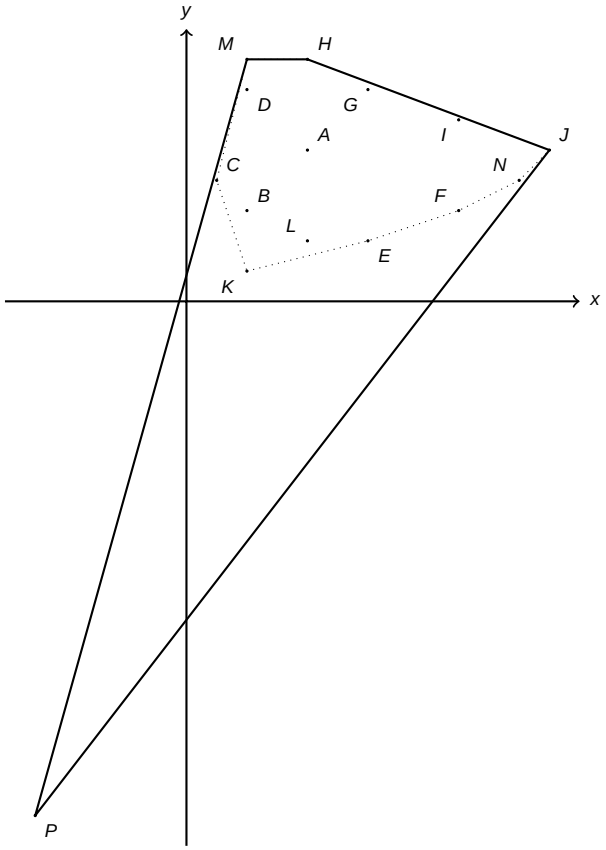
2. Ajouter un point $P(P_x, P_y)$ tel que l'enveloppe convexe de $\mathcal{N} \cup \{P\}$ soit composée d'un minimum de sommets.

   ▷ *L'idée est de prendre un point très éloigné et de le déplacer autour de notre nuage de points pour trouver les endroits qui minimisent le nombre de sommets de l'enveloppe convexe.*

   Par exemple le point $P(35, 9)$*. Dans ce cas l'enveloppe convexe se réduit aux sommets* $\{K, P, M, C\}$*.*



   Autre possibilité, le mettre en bas à gauche, par exemple en $(-5, -17)$*, pour obtenir* $\{P, J, H, M\}$*.*

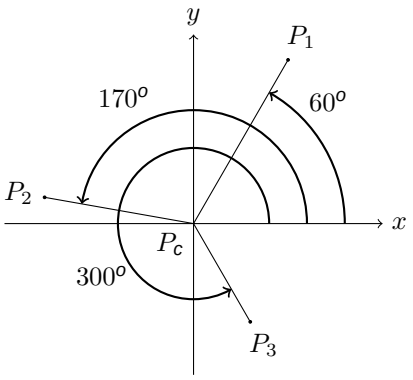


On propose l'algorithme suivant pour calculer les sommets $\mathcal{C}$ de l'enveloppe convexe d'un ensemble $\mathcal{N}$ de points :

Fonction EnveloppeConvexe($\mathcal{N}$)

```
1  si N est vide ou N ne contient qu'un élément alors
2  |  retourner N
3  C ← ∅
4  Soit P_c le point de N le plus bas
5  mode ← Droite
6  répéter
7  |  si mode = Droite alors
8  |  |  Soit P le point de N au-dessus de P_c d'angle polaire par rapport à P_c minimal
9  |  |  si il n'y en a pas alors
10 |  |  |  mode ← Gauche
11 |  si mode = Gauche alors
12 |  |  Soit P le point de N en-dessous de P_c d'angle polaire par rapport à P_c minimal
13 |  |  si il n'y en a pas alors
14 |  |  |  retourner C
15 |  C ← C ∪ {P}
16 |  P_c ← P
```

L'angle polaire d'un point P par rapport à un point P_c est l'angle polaire du vecteur $\overrightarrow{P_cP}$, c'est-à-dire celui de $(P - P_c)$ par rapport à l'origine. L'angle polaire d'un point par rapport à l'origine est la mesure, dans le sens trigonométrique (sens direct), de l'angle formé entre le point et la demi-droite des abscisses positives. Il est donc compris entre 0 et 2π radians (ou, de façon équivalente, entre 0^o et 360^o). Le calcul et la comparaison d'angles polaires s'effectue en temps constant ($\Theta(1)$).



3. (*) Montrer la terminaison et la correction de cet algorithme.
   ▷ *Les cas où $\mathcal{N}$ contient 0 ou 1 point sont triviaux. On suppose donc que $\mathcal{N}$ contient au moins 2 points.*
   Dans le mode $Droite$, à chaque tour de boucle, le point P_c est un point de $\mathcal{N}$ plus haut que le précédent, il y en a au plus $|\mathcal{N}|$. Arrivé au point le plus haut, le mode passe à $Gauche$ et le même raisonnement s'applique à la descente. Arrivé au point le plus bas, l'algorithme termine en au plus $2|\mathcal{N}|$ tours de boucle.
   Dans le mode $Droite$, l'algorithme construit le bord droit de l'enveloppe convexe, entre le point le plus bas et le plus haut. Montrons-le par récurrence en montrant que tous les points P_c sont des sommets du bord droit de l'enveloppe convexe et que tous les sommets du bord droit de l'enveloppe convexe sont bien parcourus.
   Au début P_c est le point le plus bas, c'est donc un sommet de l'enveloppe convexe, appartenant à la fois à son bord droit et à son bord gauche.
   Ensuite, étant donné un sommet P_c du bord droit de l'enveloppe convexe, le point P au-dessus de P_c d'angle polaire minimal par rapport à P_c est nécessairement un sommet du bord droit de l'enveloppe convexe. En effet, il n'y a aucun point à droite de la droite (PP_c), ni au-dessus de P_c, par hypothèse, ni en-dessous sinon P_c serait à gauche du segment formé par ce point et P ce qui contredirait le fait qu'il appartient au bord droit de l'enveloppe convexe. De plus, aucun autre point d'ordonnée comprise entre celle de P_c et celle de P n'appartient au bord droit de l'enveloppe convexe (puisqu'ils sont tous à gauche de (PP_c)).
   Un raisonnement similaire s'applique au mode $Gauche$ et au bord gauche de l'enveloppe convexe. Par conséquent l'algorithme construit bien exactement l'ensemble des sommets de l'enveloppe convexe.
4. Quelle est sa complexité dans le pire cas ? Quand cela arrive-t-il ?
   ▷ *Tout d'abord, les boucles des lignes 7 et 11 prennent un temps $O(|\mathcal{N}|)$ donc chaque itération de la boucle principale coûte également $O(|\mathcal{N}|)$.*
   Dans le pire cas, tous les points sont des sommets de l'enveloppe convexe, la variable P_c parcourt l'ensemble des points, chacun exactement une fois, la complexité est alors quadratique ($O(|\mathcal{N}|^2)$).
5. Quelle est sa complexité dans le meilleur cas ? Quand cela arrive-t-il ?
   ▷ *Dans le meilleur des cas, l'enveloppe convexe ne compte qu'un petit nombre $O(1)$ de sommets, la complexité est linéaire ($O(|\mathcal{N}|)$).*
6. Exprimer sa complexité dans le cas général.
   ▷ *Dans le cas général, la complexité est $O(|\mathcal{N}| \cdot |\mathcal{C}|)$ où $|\mathcal{C}|$ est le nombre de sommets de l'enveloppe convexe.*
7. Comparer à l'algorithme vu en cours. Dans quel cas vaut-il mieux utiliser l'un ou l'autre ?
   ▷ *La complexité de l'algorithme vu en cours (balayage de* Graham*) est $O(|\mathcal{N}| \log |\mathcal{N}|)$. La complexité de l'algorithme de ce DM (marche de* Jarvis*) est $O(|\mathcal{N}| \cdot |\mathcal{C}|)$.*
   L'algorithme de Jarvis *est donc préférable lorsque $|\mathcal{C}| = o(\log |\mathcal{N}|)$, c'est-à-dire lorsque très peu (« logarithmiquement peu ») de points sont des sommets de l'enveloppe convexe.*
8. En pratique, on ne calcule pas les angles polaires. Comment, de deux points A et B, savoir lequel a le plus petit angle polaire par rapport à un troisième point P ?
   ▷ *Pour des points A et B au-dessus de P, le point B a un plus petit angle polaire par rapport à P que A si et seulement si* `tourne_a_droite(B, P, A)`*. Il suffit pour cela de regarder le signe du produit en croix $\overrightarrow{BP} \times \overrightarrow{PA}$.*
9. (*) Proposer une implémentation en Python de `EnveloppeConvexe`.

Exercice 3 : équilibrage d'ABR

On suppose que les ABR sont représentés par une structure munie des fonctions suivantes : `créerArbreVide()`, `créerSommet(`*étiquette*`,` *filsGauche*`,` *filsDroit*`)`, `estVide(`*sommet*`)`, `étiquette(`*sommet*`)`, `filsGauche(`*sommet*`)` et `filsDroit(`*sommet*`)`.

1. Rappeler :
   a. l'ordre de grandeur de la hauteur moyenne d'un ABR,
      ▷ $\Theta(\log n)$ *où* n *est le nombre de sommets de l'ABR.*
   b. la fonction `nbSommets` qui compte les sommets d'un ABR,
      ▷
      ```
      def nbSommets(A) :
          if estVide(A) : return 0
          return 1 + nbSommets(filsGauche(A)) + nbSommets(filsDroit(A))
      ```
   c. l'algorithme d'ajout d'un sommet et sa complexité,
      ▷ *Aucune fonction ne permet de modifier l'arbre, il faut donc en créer un nouveau, comme cela a été fait en TP. On peut néanmois réutiliser des morceaux de l'ancien arbre. On dit que la structure de donnée est* persistante *ou* non mutable.

      ```
      # On suppose que l'étiquette à ajouter n'est pas déjà présente

        def ajouterSommet(A, x) :
          if estVide(A) :
            return créerFeuille(x)
          fg, fd = filsGauche(A), filsDroit(A)
          if x < étiquette(A) :
            fg = ajouterSommet(fg, x)
          else :
            fd = ajouterSommet(fd, x)
          return créerSommet(étiquette(A), fg, fd)

        def créerFeuille(x) :
          return créerSommet(x, créerArbreVide(), créerArbreVide())
      ```

      Complexité $O(h)$ *où* h *est la hauteur de l'ABR,*
   d. l'algorithme de parcours infixe et sa complexité,
      ▷
      ```
      def infixe(A) :
          if estVide(A) : return []
          return infixe(filsGauche(A)) + [étiquette(A)] + infixe(filsDroit(A))
      ```

      Complexité $\Theta(n)$.
   e. la fonction `minimum` qui retourne l'étiquette minimale d'un ABR non vide et sa complexité.
      ▷
      ```
      def minimum(A) : # on suppose A non vide
          if estVide(filsGauche(A)) : return étiquette(A)
          return minimum(filsGauche(A))
      ```

      Complexité $\Theta(h)$.
2. Écrire une fonction successeur($arbre$, $étiquette$) qui retourne l'étiquette du sommet successeur, dans l'arbre $arbre$, du sommet d'étiquette $étiquette$ ou `None` s'il n'en a pas. Quelle est sa complexité ?
   ▷ *Il s'agit ici de trouver le successeur d'un sommet de l'arbre, en partant de la racine. De plus, aucune fonction ne permet de remonter au père.*

   ```
   def successeur(A, x) :
     if estVide(A) :
       return None
     if x == étiquette(A) :
       if estVide(filsDroit(A)) :
         return None
       return minimum(filsDroit(A))
     if x > étiquette(A) :
       return successeur(filsDroit(A), x)
     s = successeur(filsGauche(A), x)
     if s != None :
       return s
     return étiquette(A)
   ```

Complexité $O(h)$.

3. Écrire un algorithme permettant de trier une liste en utilisant un ABR.

```
▷ def triABR(L) :
    A = créerArbreVide()
    for x in L :
      A = ajouterSommet(A, x)
    return infixe(A)
```

4. Quelle est sa complexité dans le pire cas ? Dans le meilleur cas ? (justifier)

   ▷ *Dans le pire cas, l'ABR créé va être filiforme. La complexité sera $(\sum_{i=1}^{n} i) + n = \Theta(n^2)$.*
   Dans le meilleur des cas, l'ABR sera équilibré. La complexité sera $(\sum_{i=1}^{n} \log i) + n = \Theta(n \log n)$.

5. Quelle est sa complexité en moyenne ? (justifier)

   ▷ *En moyenne, la hauteur de l'ABR sera $\Theta(\log n)$ donc la complexité sera $\Theta(n \log n)$.*

6. Comparer cet algorithme à QuickSort (fonctionnement et complexités).

   ▷ *Cet algorithme fonctionne exactement comme* QuickSort. *Dans* QuickSort, *chaque valeur est comparée une fois à chacun des pivots choisis précédemment. Dans* triABR, *chaque valeur est comparée une fois à chacun de ses ancêtres dans l'arbre. Il est donc tout à fait normal que les complexités soient identiques.*

Un arbre binaire est fortement équilibré s'il est vide ou si, pour chacun de ses sommets s, les fils gauche et droit de s ont même nombre de sommets, à un près :
$|\texttt{nbSommets(filsGauche(s))} - \texttt{nbSommets(filsDroit(s))}| \leq 1$.

7. Quelle est la hauteur d'un ABR fortement équilibré ? (justifier ; un ordre de grandeur pourra suffire)

   ▷ *En TD, nous avons vu que la hauteur d'un arbre binaire parfait à $n = 2^k - 1$ sommets était k. C'est aussi le cas pour un ABR fortement équilibré car sa hauteur est solution de l'équation $h(2^k - 1) = 1 + 2h(2^{k-1} - 1)$ correspondant au seul placement possible des sommets.*
   La hauteur est fonction croissante du nombre de sommets, donc la hauteur d'un ABR fortement équilibré à n sommets est égale à $\lceil \log_2(n+1) \rceil$. L'ordre de grandeur $\Theta(\log n)$ est suffisant.

8. Écrire une fonction de complexité optimale qui teste si un ABR est fortement équilibré. Quelle est sa complexité ?

```
▷ # renvoie le nombre de sommets si fortement équilibré, False sinon
  def estFortementÉquilibré(A) :
    if estVide(A) : return 0
    g = estFortementÉquilibré(filsGauche(A))
    d = estFortementÉquilibré(filsDroit(A))
    if (g is False) or (d is False) : return False
    if abs(g - d) > 1 : return False
    return g + d + 1
```

   Complexité $\Theta(n)$ car chaque sommet est visité exactement une fois.
   Attention, en Python il faut utiliser `is False` et non pas `== False` car `0 == False` est vrai.

9. Écrire un algorithme de complexité optimale permettant de construire un ABR fortement équilibré à partir d'un tableau trié. Quelle est sa complexité ?

   ▷ *Il suffit de mettre l'élément du milieu à la racine et de répéter l'opération récursivement à gauche et droite.*

```
  def abrFortementÉquilibré(T) :
    if T == [] :
      return créerArbreVide()
    m = len(T) // 2
    fg = abrFortementÉquilibré(T[:m])
    fd = abrFortementÉquilibré(T[m+1:])
    return créerSommet(T[m], fg, fd)
```

   Complexité $\Theta(n)$ car il n'y a qu'un nombre borné d'opérations pour chaque élément.

10. (*) Écrire une fonction qui supprime le sommet d'étiquette minimale d'un ABR fortement équilibré non vide tout en préservant cette propriété. Exprimer sa complexité $E(n)$ en fonction des complexités $S(n)$ de nbSommets et $A(n)$ de la fonction de la question suivante que l'on supposera déjà écrite.

▷ *Le minimum est à la racine si et seulement si le sous-arbre gauche est vide, dans ce cas il reste le sous-arbre droit.*

Sinon, le minimum est dans le sous-arbre gauche. Si celui-ci a un nombre de sommets égal ou supérieur (nécessairement de 1) au sous-arbre droit alors en extraire le minimum ne déséquilibrera pas l'arbre. Si par contre le sous-arbre gauche a un nombre de sommets strictement inférieur (toujours de 1) au sous-arbre droit alors il faut rééquilibrer l'arbre. La racine passe à gauche (il faut pour cela utiliser la fonction `ajouterSommets2` demandée à la question suivante) et le minimum du sous-arbre droit devient la racine.

```
# retourne le minimum et l'arbre sans cet élément
def extraireMinimum(A) : # A supposé non vide
  étiq, fg, fd = étiquette(A), filsGauche(fg), filsDroit(fd)
  if estVide(fg) :
    return étiq, fd
  if nbSommets(fg) >= nbSommets(fd) :
    min, fg = extraireMinimum(fg)
    return min, créerSommet(étiq, fg, fd)
  minDroit, fd = extraireMinimum(fd)
  min, fgSansMin = extraireMinimum(fg)
  fg = ajouterSommet2(fgSansMin, étiq)
  return min, créerSommet(minDroit, fg, fd)
```

Dans le pire cas, il y a deux appels à `nbSommets` (coût $S(n)$), deux appels à `extraireMinimum` et un appel à `ajouterSommet2` (coût $A(n)$).

La complexité s'exprime par $E(n) = 2S(n/2) + 2E(n/2) + A(n/2)$ avec $E(1) = 1$.

11. (**) Écrire une fonction qui ajoute un sommet à un ABR fortement équilibré tout en préservant cette propriété. Exprimer sa complexité $A(n)$ en fonction de $S(n)$ et $E(n)$. Résoudre la récurrence avec $S(n) = \Theta(n)$ puis avec $S(n) = \Theta(1)$ (comment obtenir cette complexité pour `nbSommets` ?). On pourra commencer par calculer $T(n) = E(n) + A(n)$ avant d'en déduire $A(n)$ et $E(n)$. Que peut-on en conclure sur ces deux fonctions ainsi que sur `nbSommets` ?

▷ *Les ennuies arrivent lorsque l'on veut ajouter un sommet à un fils qui a déjà plus de sommets que son frère. Dans ce cas, on peut procéder comme pour l'extraction du minimum, en faisant basculer la racine du côté le plus petit.*

```
def ajouterSommet2(A, x) :
  if estVide(A) :
    return créerFeuille(x)
  étiq, fg, fd = étiquette(A), filsGauche(A), filsDroit(A)
  if x < étiq :
    if nbSommets(fg) <= nbSommets(fd) :
      fg = ajouterSommet2(fg, x)
    else :
      max, fgSansMax = extraireMaximum(fg)
      fd = ajouterSommet2(fd, étiq)
      if x > max :
        étiq = x
      else :
        étiq = max
        fg = ajouterSommet2(fgSansMax, x)
  else :
    if nbSommets(fd) <= nbSommets(fg) :
      fd = ajouterSommet2(fd, x)
    else :
      min, fdSansMin = extraireMinimum(fd)
      fg = ajouterSommet2(fg, étiq)
      if x < min :
        étiq = x
      else :
        étiq = min
        fd = ajouterSommet2(fdSansMax, x)
  return créerSommet(étiq, fg, fd)
```

`nbSommets` coûte normalement $S(n) = \Theta(n)$. Si l'on suppose que la structure d'ABR est étendue d'un champ permettant de calculer `nbSommets` en temps constant (comme vu en TD le jeudi 16 avril) alors on pourra supposer que $S(n) = \Theta(1)$.

Dans le pire cas, il y a deux appels à `nbSommets`, un appel à `extraireMinimum` (ou `extraireMaximum`) et deux appels à `ajouterSommet2`.

La complexité s'exprime par $A(n) = 2S(n/2) + E(n/2) + 2A(n/2)$ avec $A(0) = A(1) = 1$.

$A(n)$ et $E(n)$ s'additionnent à merveille : on pose $T(n) = A(n) + E(n) = 4S(n/2) + 3T(n/2)$. On pourra retrouver $A(n)$ en résolvant $A(n) = 2S(n/2) + T(n/2) + A(n/2)$ ($E(n)$ s'obtient de la même façon donc $E(n) = A(n)$).

Avec $S(n) = \Theta(n)$, on retrouve la complexité de l'algorithme de Karatsuba, c'est-à-dire $T(n) = \Theta(n^{\log\ 3})$. Ensuite $A(n) = n + (n/2)^{\log\ 3} + A(n/2)$. Le terme $(n/2)^{\log\ 3} = n^{\log\ 3}/3$ domine n donc résolvons $A(n) = n^{\log\ 3}/3 + A(n/2)$. On pose $a(k) = A(2^k)$. $a(k) = (2^k)^{\log\ 3}/3 + a(k-1) = 3^k/3 + a(k-1)$. En développant : $a(k) = \frac{1}{3}\sum_{i=0}^{k} 3^i = \frac{1}{3}\frac{3^{k+i}-3}{3-1} \sim 3^k/2$. On retrouve $A(n) = \Theta(n^{\log\ 3})$.

Maintenant si $S(n) = \Theta(1)$, on trouve aussi $T(n) = \Theta(n^{\log\ 3})$ donc la suite est identique. Conclusion : optimiser `nbSommets` n'a ici aucun impact sur la complexité.

Les complexités $A(n)$ et $E(n)$ étant plus que linéaires, il est plus e cace – et plus facile ! – de repasser par une liste triée :

```
def extraireMinimum(A) : # complexité Θ(n)
  L = infixe(A)
  return L[0], abrFortementÉquilibré(L[1:])

def ajouterSommet2(A, x) : # complexité Θ(n)
  L = infixe(ajouterSommet(A, x))
  return abrFortementÉquilibré(L)
```

La propriété d'être fortement équilibré est inutilement forte. Cela pénalise les algorithmes d'insertion et de suppression. On fixe une constance $c > 1$. On dira qu'un arbre est *plutôt équilibré* (PE) si sa hauteur ne dépasse pas $c(\log_2(n+1)+1)$.

12. Pourquoi la propriété (PE) est-elle raisonnable ?

    ▷ *Les opérations d'ajout, de suppression et de recherche d'un élément sont en temps proportionnel à la hauteur de l'ABR. La propriété (PE) permet donc de borner la complexité de ces opérations par $O(\log n)$.*

    Le cas $c = 1$ correspondrait au cas d'un arbre presque parfait, il est donc normal de fixer $c > 1$.

13. Montrer que si les deux fils d'un sommet sont plutôt équilibrés alors cet arbre est plutôt équilibré.

On étend la structure d'ABR avec deux champs par sommet `nbSommets` et `hauteur` représentant respectivement le nombre de sommets du sous-arbre correspondant (y compris lui-même) et sa hauteur.

14. On a déjà vu en TD comment maintenir les champs `nbSommets` pendant l'ajout de sommets sans surcoût de complexité. Qu'en est-il de `hauteur` ? (comment le maintenir ? à quel coût ?)

    ▷ *Pareil. Cf. TP noté du mercredi 6 mai.*

15. (*) Modifier la fonction d'ajout de sommet à un ABR pour que tout sous-arbre créé reste *plutôt équilibré*. On pourra pour cela e□ectuer des *rééquilibrages forts* en utilisant les fonctions des questions 1.d et 9. Quelle est sa complexité ? (meilleur et pire cas). Peut-on borner le nombre de rééquilibrages forts ?

16. (**) Étudier la complexité de création d'un ABR qui n'utilise que la fonction de la question précédente (meilleur cas, pire cas, moyenne).