DS INFO N°1 (le 24/01/2022)

PROBLÉME 1 (extrait de CAPES Informatique 2021

Pour ce sujet, vous pourrez utiliser les fonctions de manipulation de listes ou de matrices suivantes :

- Création d'une liste de taille n remplie avec la valeur x : `li = [x] * n`.
- Obtention de la taille d'une liste `li` : `len(li)`.
- Si li est une liste de n éléments, on peut accéder au k^e élément (pour $0 \leqslant k \leqslant$ `len(li)`) avec `li[k]`. On peut définir sa valeur avec `li[k]=x`.
- Un élément x peut être ajouté dans une liste `li` à l'aide de `li.append (x)`. On considèrera qu'il s'agit d'une opération élémentaire.
- Le minimum (resp. le maximum) des éléments d'une liste `li` contenant des valeurs numériques s'obtient par `min(li)` (resp `max(li)`). Cette opération est de complexité $O(n)$ où n est la longueur de la liste.
- Les matrices sont des listes de listes, chaque sous-liste étant considérée comme une ligne de la matrice. Si `mat` est une matrice, elle possède `len(mat)` lignes et `len(mat [0])` colonnes.
- Création d'une matrice de n lignes et p colonnes, dont toutes les cases contiennent x :

```
mat =[ [x for j in range(p)] for i in range(n)]
```

  .
- On accède à (resp. modifie) l'élément de mat dans la i^e ligne et j^e colonne avec `mat[i][j]` (resp. `mat[i][j]=x`).

À moins de les redéfinir explicitement, l'utilisation de toute autre fonction sur les listes (`sort`, `index`, etc.) est interdite. On rappelle enfin qu'une fonction qui s'arrête sans avoir rencontré l'instruction return renvoie `None`.

Ce problème, pouvant par exemple survenir dans le domaine de la navigation maritime, vise à déterminer, dans un nuage de points du plan, la paire de points les plus proches. Il est constitué de trois parties dépendantes.

Formellement, on suppose qu'on dispose de n points dans le plan $(M_0, M_1, \ldots, M_{n-1})$ dans un ordre quelconque pour le moment. Ils seront représentés en Python par deux listes de flottants de taille n : `coords_x` et `coords_y`, donnant respectivement les abscisses et les ordonnées des points. On dira ainsi que M_i est le point d'indice i, qu'il a pour abscisse $x_i =$ `coords_x[i]` et pour ordonnée $y_i =$ `coords_y[i]` . On supposera que `coords_x` et `coords_y` sont des variables globales, qu'on ne modifiera jamais au cours de l'exécution de l'algorithme.

I. Approche exhaustive

On utilise la distance euclidienne définie par $d\left(M_i, M_j\right) = \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2}$.

- **Question 1** Écrire une fonction distance (i, j) qui renvoie la distance entre les points M_i et M_j. On utilisera la fonction `sqrt` après l'avoir importée.

- **Question 2** Écrire une fonction `plus_proche()` qui renvoie, à l'aide d'une recherche exhaustive, le couple d'entiers des indices i et j des deux points les plus proches du nuage de points.

- **Question 3** Donner, en la justifiant sommairement, la complexité de la fonction précédente en fonction de n.

II. Quelques outils pour s'améliorer

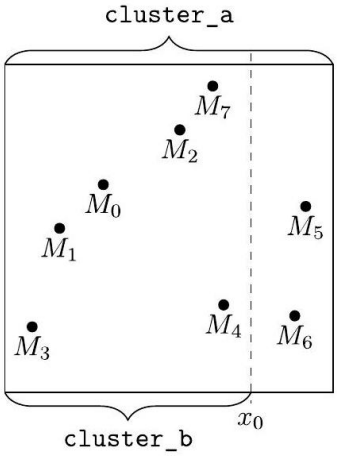
On souhaite maintenant obtenir la distance entre les deux points les plus proches avec une meilleure complexité. Pour cela nous allons décrire un algorithme utilisant une méthode de type *diviser pour régner*. Cette partie introduit des fonctions utiles pour la mise en œuvre de cet algorithme. On se donne la fonction suivante :

```
def tri(liste):
    n = len(liste)
    for i in range(n):
        pos = i
        while pos > 0 and liste[pos] < liste[pos-1]:
            liste[pos], liste[pos-1] = liste[pos-1], liste[pos]
            pos -= 1
```

- **Question 5** Que renvoie cette fonction ? Que fait-elle ? Le démontrer soigneusement en exhibant un invariant de boucle (*si vous ne trouvez pas l'invariant, essayez au moins d'expliquer sommairement son fonctionnement*).
- **Question 6** Donner, en la démontrant, la complexité dans le pire des cas de la fonction `tri` en fonction de la taille de la liste donnée en paramètre.
- **Question 7** On souhaite trier une liste contenant des indices de points suivant l'ordre des abscisses croissantes. Que faudrait-il changer à la fonction `tri` ci-dessus pour qu'elle réalise cette opération ?
- **Question 8** Indiquer le nom d'un autre algorithme de tri plus efficace dans le pire des cas, ainsi que sa complexité. Expliquer sommairement son principe. On ne demande pas de le programmer.

On admettra que l'on dispose de deux listes de n entiers `liste_x` (resp. `liste_y`) contenant les indices des points du nuage triés par abscisses croissantes (resp. par ordonnées croissantes). On supposera désormais que deux points quelconques ont des abscisses et des ordonnées distinctes.

Dans toute la suite, un sous-ensemble de points sera décrit par un cluster. Un cluster est une matrice de deux lignes contenant chacune les mêmes numéros correspondant aux numéros des points dans le sous-ensemble considéré. Dans la première ligne, les points sont triés par abscisses croissantes ; dans la seconde, ils sont triés par ordonnées croissantes. La figure 1 donne la représentation de deux clusters.




```
liste_x = [3, 1, 0, 2, 7, 4, 6, 5]
liste_y = [3, 6, 4, 1, 5, 0, 2, 7]

cluster_a = [liste_x, liste_y]

cluster_b = [ [3, 1, 0, 2, 7, 4],
              [3, 4, 1, 0, 2, 7] ]
```

FIGURE 1 - Représentation en Python de deux clusters.

Pour être efficace, notre algorithme ne doit pas re-trier les listes des indices de points à chaque étape. Nous allons donc définir une fonction qui permet d'extraire des indices d'un cluster et former ainsi un nouveau cluster plus petit.

- **Question 9** Écrire une fonction `sous_cluster(cl, x_min, x_max)` qui prend en arguments un cluster `cl` et deux flottants `x_min` et `x_max`, et renvoie le sous-cluster des points dont l'abscisse est comprise entre `x_min` et `x_max` (au sens large). Cette fonction doit avoir une complexité linéaire en la taille du cluster.
- **Question 10** Écrire une fonction `mediane(cl)` qui prend en entrée un cluster `cl` contenant au moins 2 points et renvoie une abscisse médiane, c'est-à-dire que la moitié (au moins) des points a une abscisse inférieure ou égale à cette valeur, et la moitié (au moins) des points a une abscisse supérieure ou égale à cette valeur. Cette fonction doit avoir une complexité en $O(1)$.

III. Méthode sophistiquée

Le fonctionnement de l'algorithme utilisant une méthode de type diviser pour régner est illustré par la figure 2 :

1. Si le cluster contient deux ou trois points, on calcule la distance minimale en calculant toutes les distances possibles.
2. Sinon, on sépare le cluster en deux parties G et D qu'on supposera de tailles égales (éventuellement à un point près) suivant la médiane des abscisses, qu'on notera x_0.
3. Les deux points les plus proches sont soit tous les deux dans G, soit tous les deux dans D, soit un dans G et un dans D.
4. On calcule récursivement le couple le plus proche dans G et le couple le plus proche dans D. On note d_0 la plus petite des deux distances obtenues.
5. On cherche s'il existe une paire de points (M_1, M_2) telle que M_1 est dans G, M_2 dans D, et $d(M_1, M_2) < d_0$.
6. Si on en trouve une (ou plusieurs), on renvoie la plus petite de ces distances. Sinon, on renvoie d_0.

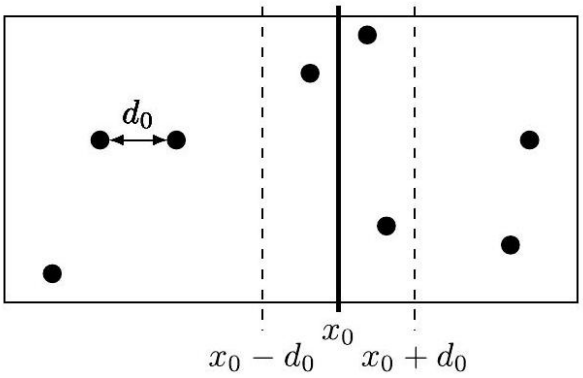


Figure 2 - Illustration du diviser pour régner

► **Question 11** Écrire une fonction `gauche(cl)` qui prend en argument un cluster `cl` contenant au moins deux points et renvoie le cluster constitué uniquement de la moitié (éventuellement arrondie à l'entier supérieur) des points les plus à gauche du cluster `cl`.

On supposera qu'on dispose aussi d'une fonction `droite(cl)` qui renvoie le cluster contenant tous les autres points du cluster `cl` n'appartenant pas au cluster renvoyé par la fonction `gauche(cl)` (on ne demande pas de l'écrire).

► **Question 12** Justifier que l'on peut se contenter de chercher les points M_1 et M_2 de l'étape 5 de l'algorithme dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 - d_0 ; x_0 + d_0]$.

► **Question 13** Écrire une fonction `bande_centrale(cl, d0)` qui prend en argument un cluster `cl` et un réel `d0`, et renvoie le cluster des points dont l'abscisse est dans I_0. Cette fonction doit avoir une complexité linéaire en la taille du cluster.

► **Question 14** Montrer que deux points M_1 et M_2 (de l'étape 5 de l'algorithme) situés à une distance inférieure à d_0 se trouvent, dans la deuxième ligne du cluster obtenu à la question précédente (c'est-à-dire la ligne triée par ordonnées croissantes), séparés d'au plus 6 éléments.

Le résultat de cette question 14 (un peu délicate) sera admis.

► **Question 15** En déduire une fonction `fusion(cl, d0)` qui prend en entrée un cluster de points dont toutes les abscisses sont dans un intervalle $[x_0 - d_0 ; x_0 + d_0]$, et renvoie la distance minimale entre deux points du cluster si elle est inférieure à d_0, ou d_0 sinon. Cette fonction doit avoir une complexité linéaire en la taille du cluster `cl`. Vous justifierez cette complexité.

► **Question 16** Écrire une fonction récursive `distance_minimale(cl)` qui prend en argument un cluster et utilise l'algorithme décrit plus haut pour renvoyer la distance minimale entre deux points du cluster.

Les questions 17 et 18 ne font pas partie du DS.

► **Question 17** Si on note n la taille du cluster `cl`, et $C(n)$ le nombre d'opérations élémentaires réalisées par la fonction `distance_minimale(cl)`, justifier que l'on a :

$$C(n) = 2C(n/2) + O(n)$$

► **Question 18** En déduire $C(n) = O\left(n \log_2(n)\right)$. On pourra commencer par traiter le cas où n est une puissance de 2.

PROBLÉME 2 (d'après X PSI 2013

Cette épreuve a pour objectif de choisir où placer des poteaux télégraphiques pour relier le point le plus à gauche d'un paysage unidimensionnel au point le plus à droite en fonction de critères de coût. Nous ferons les simplifications suivantes : les fils sont sans poids et tendus ; ils relient donc en ligne droite les sommets de deux poteaux consécutifs. Les normes de sécurité imposent que les fils soient en tout point à une distance d'au moins Δ (mesurée verticalement) au-dessus du sol. Les poteaux sont tous de longueur identique $\ell \geqslant \Delta$. Voici par exemple une proposition valide de placement de poteaux pour le paysage ci-dessous avec $\Delta = 0.5$ et $\ell = 2.0$.

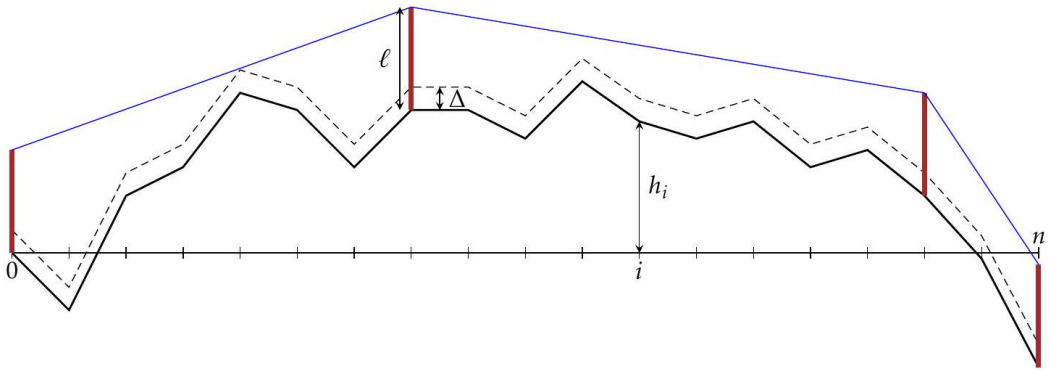


FIGURE 1 - Un exemple de poteaux où le fil est en tout point à au moins Δ au-dessus du sol.

Nous attacherons la plus grande importance à la lisibilité du code produit par les candidats ; aussi, nous encourageons les candidats à introduire des fonctions intermédiaires lorsque cela en simplifie l'écriture.

Complexité. La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P a une complexité en $O\big(f(n)\big)$ s'il existe $K > 0$ tel que la complexité de P est inférieure à $Kf(n)$ pour tout n.

De manière générale, on s'attachera à garantir une complexité aussi petite que possible dans les fonctions écrites. Le candidat y sera tout particulièrement sensible lorsque la complexité d'une fonction est demandée. Dans ce cas, il devra de plus justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Listes en Python. On rappelle qu'en Python, les listes sont des tableaux dynamiques à une dimension. Sur les listes, on dispose des opérations suivantes, qui sont de complexité constante :

- `[ ]` crée une liste vide c'est-à-dire ne contenant aucun élément.
- si n est un entier, `[x]*n` crée une liste de longueur n où chaque élément est égal à la valeur de `x`.
- `len(liste)` renvoie la longueur de la liste `liste`.
- `liste`$[i]$ renvoie l'élément d'indice i de la liste `liste` s'il existe, ou produit une erreur sinon (noter que les éléments sont indicés à partir de 0).
- `liste.append(x)` ajoute la valeur de `x` à la fin de la liste `liste` qui s'allonge ainsi d'un élément.

Important : L'usage de toute autre fonction sur les listes telle que `max(liste)`, `min(liste)`, `liste.insert(i,x)`, `liste.remove(x)`, `liste.index(x)`, ou encore `liste.sort()` est rigoureusement interdit (ces fonctions devront être programmées explicitement si nécessaire).

Toutes les listes définies dans ce problème seront considérées comme des variables globales, accessibles et modifiables directement par toutes les procédures et fonctions.

Dans tout le sujet, la fonction `sqrt` qui calcule la racine carrée d'un nombre positif ou nul pourra être utilisée.

Partie I : Planter le paysage

Nous définissons le paysage à partir d'une suite de relevés de dénivelés stockés dans une liste `deniveles` de nombres flottants de taille $n+1$. La liste `deniveles` est supposée être une variable globale.

Le paysage est alors décrit par une ligne brisée de $n+1$ points, dont le i-ème point P_i est de coordonnées (i, h_i) où :

$$h_0 = 0.0 \quad \text{et} \quad h_i = h_{i-1} + \texttt{deniveles}[i] \quad \text{pour } i \in [\![1\,;n]\!].$$

- **Question 1** Écrire une fonction `caculHauteurs(n)` qui stocke les hauteurs des points dans une variable globale `hauteurs` représentant une liste de taille $n+1$.
- **Question 2** Écrire une fonction `calculFenetre(n)` qui calcule les hauteurs minimale et maximale d'un point du paysage et les stocke dans deux variables globales `hMin` et `hMax`. On stockera également dans deux variables globales `iMin` et `iMax` les indices des points les plus à gauche de hauteur minimale et maximale respectivement.

Pour tout $(i, j) \in [\![1\,;n]\!]^2$; on appelle distance au sol de i à j, la longueur de la ligne brisée allant du point P_i au point P_j.

- **Question 3** Écrire une fonction `distanceAuSol(i, j)` qui calcule et renvoie la distance au sol de P_i à P_j.

On appelle pic un point P_i d'indice $1 \leqslant i < n$ tel que $h_i > \max(h_{i-1}, h_{i+1})$. On appelle point remarquable, les pics ainsi que les points des bords gauche (point P_0) et droit (point P_n). On appelle bassin toute partie du paysage allant d'un point remarquable au suivant.

- **Question 4** Écrire une fonction `estRemarquable(i)` qui permet de déterminer si le point P_i est remarquable. Cette fonction renverra un booléen.
- **Question 5** Écrire une fonction `longueurDuPlusLongBassin(n)` qui calcule et renvoie la distance au sol maximale d'un bassin dans le paysage.

Partie II : Planter les poteaux

On souhaite relier le point le plus à gauche au point le plus à droite par un fil télégraphique. Pour cela, nous devons choisir à quels points parmi les $(P_i)_{0\leqslant i\leqslant n}$ planter les poteaux télégraphiques intermédiaires. Rappelons que le fil est supposé sans poids et tendu et qu'il relie donc les sommets de chacun des poteaux en ligne droite. Les poteaux sont plantés verticalement et ont tous une longueur identique $\ell \geqslant \Delta$.

ℓ et Δ seront respectivement représentées par les variables globales `l` et `delta`.

La législation impose que le fil doit rester à une distance supérieure ou égale à Δ (mesurée verticalement) au-dessus du sol. Pour tester si un fil tiré entre un poteau placé au point P_i et un poteau placé au point P_j (avec $i < j$) respecte la législation, notons :

$$\alpha_{i,k} = \frac{(h_k + \Delta) - (h_i + \ell)}{k - i} \text{ pour } i < k < j \quad \text{et} \quad \beta_{i,j} = \frac{(h_j + \ell) - (h_i + \ell)}{j - i}$$

les pentes des fils tirés depuis le poteau en P_i jusqu'à une hauteur Δ au-dessus du point P_k d'une part et jusqu'à une hauteur ℓ au-dessus du point P_j d'autre part (voir la Figure 2).

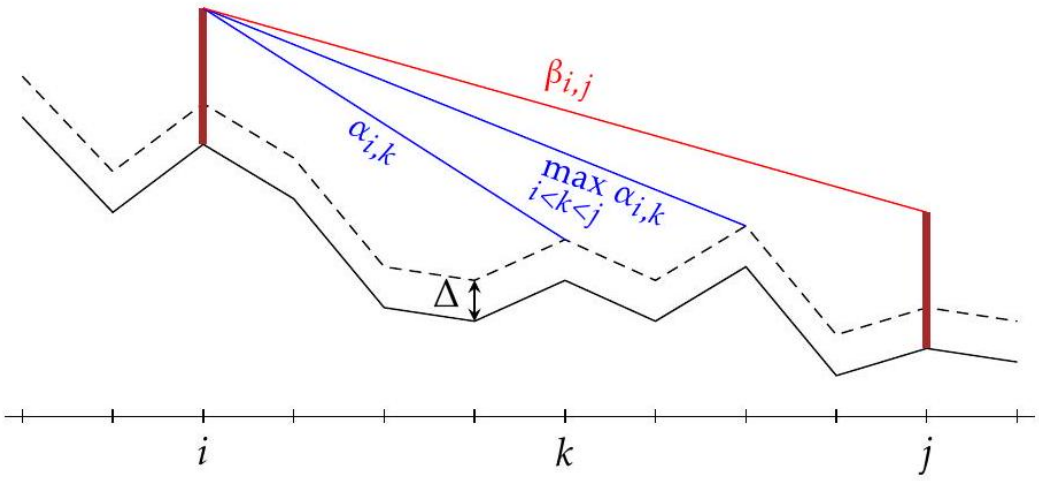


FIGURE 2 - Condition sur les pentes pour pouvoir tirer un fil.

On remarquera que le fil tiré d'un poteau en P_i à un poteau P_j (avec $j > i$) respecte la législation si et seulement si :

$$\beta_{i,j} \geqslant \max_{i<k<j} \alpha_{i,k}$$

► **Question 6** En utilisant cette méthode, écrire une fonction `estDeltaAuDessusDuSol(i, j, 1, delta)` qui renvoie `True` si un fil tiré entre les sommets d'un poteau placé au point P_i et d'un poteau placé au point P_j respecte la législation, et renvoie `False` dans le cas contraire.

Considérons une première stratégie, dite *algorithme glouton en avant*. Le premier poteau est planté en P_0. Pour calculer l'emplacement du prochain poteau, on part du dernier poteau planté et on avance (à droite) avec le fil tendu tant que la législation est respectée (et que P_n n'est pas atteint). Un nouveau poteau est alors planté, et on recommence jusqu'à ce que P_n soit atteint.

La figure 3 illustre la solution produite par cet algorithme.

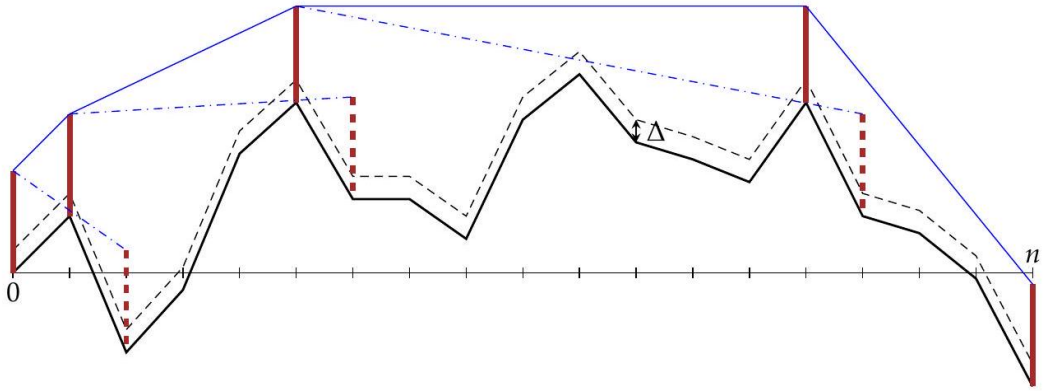


FIGURE 3 - Solution produite par l'algorithme glouton en avant. Les poteaux et les fils en pointillés sont ceux violant la législation.

La fonction `placementGloutonEnAvant` renvoie ici la liste `[1, 5, 14, 18]`.

► **Question 7** Écrire une fonction `placementGloutonEnAvant(n, 1, delta)` qui calcule la disposition des poteaux selon l'algorithme ci-dessus. La solution retournée sera donnée sous la forme d'une liste globale `poteaux` dans laquelle :

- la case `poteaux[0]` contient le nombre de poteaux utilisés ;
- la case `poteaux[t]`, pour $t \geqslant 1$, contient l'indice i indiquant que le t-ème poteau, s'il existe, est planté en P_i.

► **Question 8** Donner une majoration de la complexité du temps de calcul de votre algorithme en fonction de n. Justifier qu'on peut se contenter du calcul de $O(n)$ pentes pour implémenter l'algorithme glouton en avant, et modifier votre fonction (si nécessaire) pour obtenir une complexité en $O(n)$.

L'algorithme glouton en avant a tendance à placer beaucoup trop de poteaux, en particulier dans les vallées alors qu'il suffirait de relier les deux extrémités par un unique fil. Nous considérons donc une alternative, dite *glouton au plus loin*, qui consiste à planter le prochain poteau le plus à droite possible de la position courante. Le premier poteau est toujours planté en P_0.

La figure 4 illustre la solution produite par cet algorithme sur le même paysage que celui de la figure 3 .

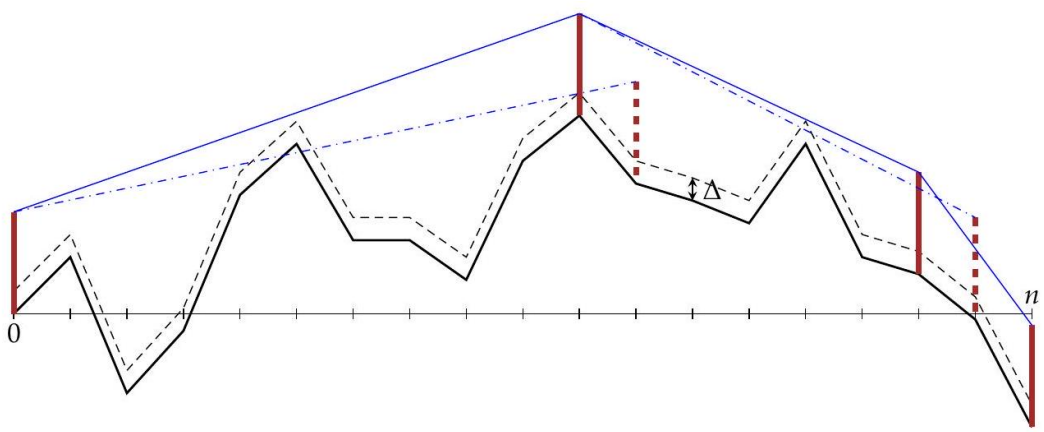


FIGURE 4 - Solution produite par l'algorithme glouton au plus loin. Les poteaux et fils en pointillés sont ceux violant la législation.

La fonction `placementGloutonAuPLusLoin` renvoie ici la liste `[10, 16, 18]`.

- **Question 9** Écrire une fonction `placementGloutonAuPlusLoin(n, l, delta)` qui calcule la disposition des poteaux selon l'algorithme ci-dessus. La solution sera donnée sous la forme d'une liste globale `poteaux` dans laquelle :
  - la case `poteaux[0]` contient le nombre de poteaux utilisés ;
  - la case `poteaux[t]`, pour $t \geqslant 1$, contient l'indice i indiquant que le t-ème poteau, s'il existe, est planté en P_i.

Partie III : Minimiser la longueur du fil

L'objectif est maintenant de calculer un placement optimal des poteaux en terme de longueur totale du fil (le nombre de poteaux pouvant être maintenant arbitraire), en utilisant les principes de la programmation dynamique.

Une liste `optL` peut être calculée de proche en proche, telle que `optL[i]` désigne la longueur minimale de fil à utiliser si l'on souhaitait relier le point P_0 au point P_i.

- **Question 10** Soit $L_{i,j}$ la longueur du fil nécessaire pour relier P_i et P_j (si cela est possible), c'est-à-dire la distance en ligne droite entre ces deux points.

  Montrer que le tableau `optL` vérifie :
  - `optL[0]=0` ;
  - et pour $1 \leqslant i \leqslant n$:

$$\texttt{optL[i]} = \min\left\{L_{i,j} + \texttt{optL[j]},\ \text{où } j \text{ est tel que : } 0 \leqslant j < i \text{ et il est possible de tirer un fil de } P_j \text{ à } P_i\right\}.$$

La case `optL[n]` contient alors la longueur minimale de fil pour relier le bord gauche au bord droit.

- **Question 11** Écrire une fonction `longueurDuSegment(i,j)` qui calcule $L_{i,j}$.

- **Question 13** En utilisant les principes de la programmation dynamique descendante, écrire une fonction récursive `longueurMinimale1(n, l, delta)` qui renvoie la longueur minimale de fil pour relier le bord gauche au bord droit, en respectant la législation.

- **Question 13** En utilisant les principes de la programmation dynamique ascendante, écrire une fonction non récursive `longueurMinimale2(n, l, delta)` qui renvoie la longueur minimale de fil pour relier le bord gauche au bord droit, en respectant la législation.

- **Question 14** Modifier votre fonction `longueurMinimale2(n, l, delta)` pour qu'elle retourne en plus une liste `precOptL` de taille $n+1$, où `precOptL[i]` contient l'indice du poteau précédant le poteau placé en P_i dans une solution de longueur minimale reliant P_0 à P_i (par convention, `precoptL[0]=-1`.)

- **Question 15** Écrire une fonction qui retourne un placement des poteaux minimisant la longueur du fil de P_0 à P_n à partir du tableau `precOptL`. La solution sera retournée sous la forme d'une liste `poteaux` définie de la même façon que dans les questions 7 et 9.

  Quelle est la complexité de votre procédure en fonction de n ?