Ce problème utilise les listes Python mais seules les opérations qui suivent sont autorisées. Si `L` est une liste :

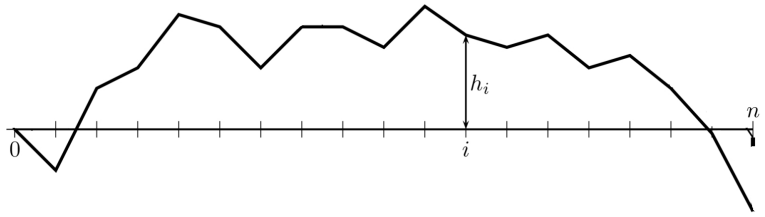
- ▷ `len(L)` renvoie la longueur de la liste L, c'est-à-dire son nombre d'éléments, complexité en O(1) ;
- ▷ `L[i]` désigne l'élément d'indice `i` de `L` (où `i` est compris entre `0` et `len(L)-1`), complexité en O(1) ;
- ▷ `L[-1]` désigne l'élément d'indice `len(L)-1` de `L`, complexité en O(1) ;
- ▷ `L.append(e)` modifie la liste `L` en lui ajoutant l'élément `e` en dernière position, complexité en O(1) ;
- ▷ `L.pop()` renvoie le dernier élément de la liste `L` (supposée non vide) et supprime l'occurrence de cet élément en dernière position de la liste, complexité en O(1) ;
- ▷ les itérations sur une liste peuvent être effectuées directement (une liste Python est un objet itérable) ou à l'aide de la fonction range ;
- ▷ les listes peuvent être créées avec des boucles, ou en compréhension (par exemple `[0 for i in range(5)]`) ou avec une syntaxe de la forme `[0]*5`.

D'autre part, les fonctions `max` et `min` ne peuvent être utilisées que pour considérer le maximum ou le minimum de deux éléments (et non pour une liste).

La complexité, ou le temps d'exécution, d'un programme P 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 $O(f(n))$, s'il existe $K > 0$ tel que la complexité de P soit au plus $K f(n)$, pour tout n. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Partie 1 - étude d'un profil de paysage

Nous considérons un paysage représenté par une suite de relevés de hauteurs c'est-à-dire décrit par une ligne brisée de $n+1$ points, dont le i-ème point P_i est de coordonnées (i, h_i).



Par exemple une liste `[5,0,3,4]` désigne un profil déterminé par les points d'abscisses 0, 1, 2 et 3 de hauteurs 5, 0, 3 et 4. La question de l'unité dans laquelle sont exprimées ces longueurs ne se posera pas dans le sujet.

► **Q1.** On dit qu'une liste `L` est croissante si ses éléments forment une suite croissante.
Par exemple, la liste `[2, 3, 5, 7]` est croissante mais pas la liste `[3, 4, 2, 6]`.
Écrire une fonction `croissante(l:list)->bool` prenant en argument une liste non vide `l` de nombres et renvoyant un booléen indiquant si la liste est croissante. Par exemple :

```
>>> croissante([2, 3, 4])
True
>>> croissante([2, 4, 3])
False
```

► **Q2.** Écrire une fonction `maxi(l:list)->list` prenant en argument une liste `l` de nombres et renvoyant la liste des indices où se trouve la valeur maximale de `l`. Par exemple :

```
>>> maxi([2, 5, 1, 5, 3])
[1, 3]
```

► **Q3.** On appelle *distance au sol* de i à j, la longueur de la ligne brisée allant du point P_i au point P_j (exprimée dans «l'unité» dans laquelle sont exprimées les hauteurs et les abscisses).
On suppose que l'on dispose d'une fonction `sqrt` donnant la racine carrée d'un nombre flottant.
Écrire une fonction `distance(l:list, i:int, j:int)->float` qui renvoie la distance au sol de P_i à P_j (où `l` est la liste des hauteurs des points).

► **Q4.** On dit qu'une liste `l` présente un *pic* en `i` lorsque :

- soit `i` = 0 et `l[0]>l[1]` ;
- soit 1 ⩽ `i` ⩽ `len(l)`−2 et `l[i]>l[i-1]` et `l[i]>l[i+1]` ;
- soit `i` = `len(l)`−1 et `l[len(l)-1]>l[len(l)-2]`.

Les pics de la deuxième catégorire seront qualifiés d'*intérieurs*.
Écrire une fonction `pic(l:list, i:int)->bool` d'arguments une liste `l` de nombres et un entier `i` compris entre 0 et `len(l)-1`, qui renvoie un booléen indiquant si `l` présente un pic en `i`. Par exemple :

```
>>> pic([3, 1, 2, 0], 0)
True
>>> pic([3, 1, 2, 0], 1)
False
>>> pic([3, 1, 2, 0], 2)
True
>>> pic([3, 1, 2, 0], 3)
False
```

► **Q5.** Une liste possède-t-elle au moins un pic intérieur ? Combien de pics possède-t-elle au maximum (en fonction de son nombre d'éléments n) ?

► **Q6.** Écrire une fonction `pic_intérieur(l:list)->bool` d'arguments une liste `l` de nombres et qui renvoie un booléen indiquant si `l` présente au moins un pic intérieur. Par exemple :

```
>>> pic_interieur([1, 2, 3, 0])
True
>>> pic_interieur([1, 2, 3])
False
>>> pic_interieur([5, 1, 2, 3])
False
```

► **Q7.** Écrire une fonction `pics(l:list)->list` d'arguments une liste `l` d'entiers tous distincts et qui renvoie la liste des indices des pics de `l`. Par exemple :

```
>>> pics([1, 2, 3, 0, 5, 4])
[2, 4]
>>> pics([11, 6, 3, 9])
[0, 3]
```

► **Q8.** On appelle *vallée* d'une liste `l` :

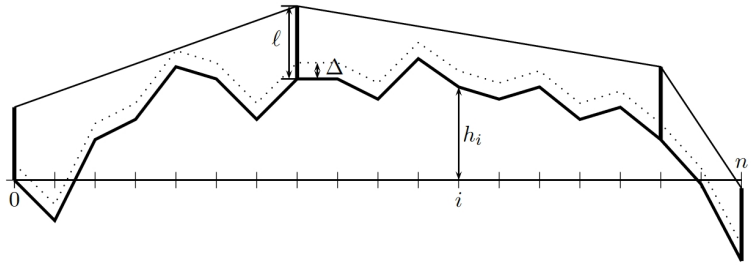
- une sous-liste de `l` allant d'un pic au suivant ;
- ou la sous-liste allant du début de `l` au premier pic si `l` n'a pas de pic en 0 ;
- ou la sous-liste allant du dernier pic à la fin de `l` si `l` n'a pas de pic au dernier indice.

Par exemple, les pics de `[0,2,4,6,8,5,7,3]` sont aux indices 4 et 6 et les vallées de cette liste sont `[0,2,4,6,8]`, `[8,5,7]` et `[7,3]`.
Écrire une fonction `plus_grande_vallee(l:list)->int` d'arguments une liste `l` de nombres et qui renvoie la longueur de la plus grande vallée de `l` (ou de la première d'entre elles s'il y en a plusieurs). Par exemple :

```
>>> plus_grande_vallee([0, 2, 4, 6, 8, 5, 7, 3])
5
>>> plus_grande_vallee([0, 10, 1, 2, 3, 4, 11, 5, 6])
6
>>> plus_grande_vallee([0, 2, 3])
3
>>> plus_grande_vallee([3, 2, 4])
3
```

Partie 2 - installation de poteaux

L'objectif désormais est 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 $\ell = 2.0$).

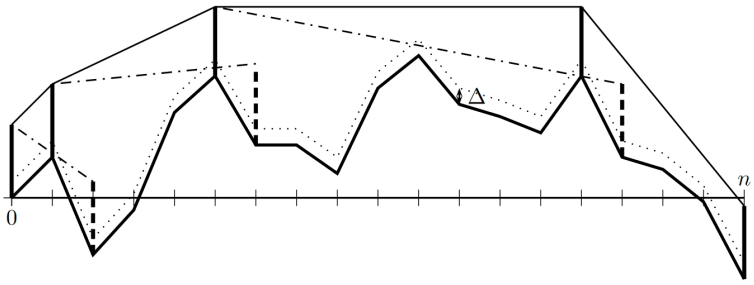


On souhaite relier le point le plus à gauche au point le plus à droite par un fil télégraphique. Pour cela, nous devons choisir à quels points, parmi $P_0, P_1, \ldots, P_n$, planter les poteaux télégraphiques intermédiaires.
La législation impose que le fil doive rester à une distance supérieure ou égale à Δ (mesurée verticalement) au-dessus du sol. On suppose que l'on dispose d'une fonction (`long` pour ℓ et `delta` pour Δ) :

```
estDeltaAuDessusDuSol(l:list, i:int, j:int, long:float, delta:float)->bool
```

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 (vers la 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 suivante illustre la solution produite par cet algorithme (où les poteaux et les fils en pointillés sont ceux ne respectant pas la législation).

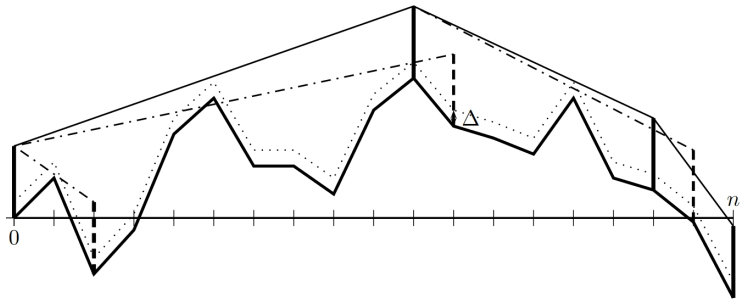


► **Q9.** On souhaite programmer une fonction `placementGloutonEnAvant(l, long, delta)` (où `long` correspond à ℓ et `delta` correspond à Δ) qui renvoie la liste des indices des poteaux à placer selon l'algorithme ci-dessus.
Compléter le code suivant (là où il y a `____`) :

```
def placementGloutonEnAvant(l, long, delta):
    n = len(l)
    poteaux = [0]       # premier poteau placé en x = 0
    i, j = ____
    while i != n - 1:
        while ____
            ____
        i = ____
        poteaux.append(i)
    return poteaux
```

► **Q10.** On admet que la complexité de `estDeltaAuDessusDuSol}(l, i, j, long, delta)` est en O(`j` − `i`). Déterminer une majoration de la complexité du temps de calcul de votre algorithme en fonction de n (longueur de la liste `l`).

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 suivante illustre la solution produite par cet algorithme sur le même paysage que celui de la figure précédente.



► **Q11.** Écrire une fonction `placementGloutonAuPlusLoin(l, long, delta)`qui renvoie la liste des indices des poteaux à placer selon l'algorithme ci-dessus.