Devoir d'informatique

Quelques remarques avant de commencer

Le sujet est constitué de trois exercices indépendants. Les fonctions demandées doivent être écrites dans le langage C. Ne vous précipitez pas, essayez de faire *bien* avant de faire *beaucoup*, et n'hésitez pas à privilégier les parties qui vous parlent davantage.

On prendra bien soin à veiller à la lisibilité du code proposé, en choisissant judicieusement les noms de variables utilisés, et en assortissant les fonctions de commentaires, d'invariants ou d'explications brèves mais pertinentes permettant de comprendre les choix effectués. Ces commentaires n'ont pas à être insérés dans le code, il est généralement préférable de décrire la fonction avant ou après le code pour des raisons de lisibilité.

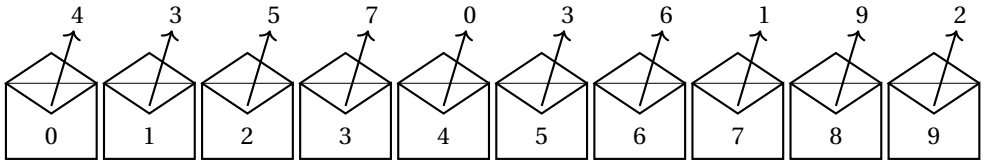
Si une question impose une complexité, votre proposition doit respecter cette complexité, ne perdez pas de temps à proposer une solution moins efficace, cela ne vous rapportera pas de points.

Vous pouvez introduire toutes les fonctions auxiliaires dont vous avez besoin. En langage C, les fonctions `abs` et `sqrt` sont considérée fournies. Vous pouvez également utiliser dans une question toutes les fonctions décrites dans les questions précédentes du même problème, et ce *même si vous n'avez pas réussi à en proposer une implémentation.*

1 Enveloppes

On suppose disposer de n enveloppes, numérotées de 0 à $n-1$. Chacune de ces enveloppes contient un entier compris entre 0 et $n-1$ (inclus).

Par exemple, pour $n = 10$, les dix enveloppes peuvent contenir les valeurs suivantes :



On suppose défini une constante entière `N` contenant le nombre d'enveloppes, et une fonction « `int content(int k)` » renvoyant le contenu de l'enveloppe dont le numéro est fourni en paramètre (par exemple, « `content(3)` », pour l'exemple proposé, donnerait 7). Le comportement de `content` est indéfini pour des arguments qui ne sont pas dans $[\![0\,..\,N-1]\!]$, tout appel à `content` avec un argument n'étant pas dans cet intervalle est donc à proscrire.

`N` et `content` sont supposés accessibles et utilisables depuis n'importe quelle fonction.

1. Proposer une fonction `int nb_inf(void)` ne prenant aucun argument et renvoyant le nombre d'enveloppes dont le nombre placé à l'intérieur de l'enveloppe est strictement inférieur au numéro écrit sur cette même enveloppe (il y en a quatre sur l'exemple proposé, les enveloppes 4, 5, 7 et 9).

2. Proposer une fonction `int max_diff(void)` ne prenant aucun argument et renvoyant la plus grande différence, en valeur absolue, entre le numéro de l'enveloppe et le nombre qu'elle contient (7 sur l'exemple fourni).

3. Soit v un entier. On considère l'ensemble $\mathcal{E}$ des enveloppes dont le nombre placé à l'intérieur est supérieur ou égal à v. Proposer une fonction `int max_over(int v)` prenant en argument l'entier `v` et renvoyant le plus grand numéro figurant *sur* les enveloppes de $\mathcal{E}$ si l'ensemble $\mathcal{E}$ est non-vide, et -1 si cet ensemble $\mathcal{E}$ est vide.

4. Proposer une fonction `int nb_pairs(void)` renvoyant le nombre de paires d'enveloppes telles qu'à l'intérieur de *chacune* des deux enveloppes se trouve le numéro inscrit à l'extérieur de l'autre (sur notre exemple, les enveloppes 0 et 4 forment ainsi une paire, la seule présente dans cet exemple).

On s'intéresse à présent aux règles suivantes, qui permettent de construire des séquences parmi les enveloppes : on part d'une enveloppe portant le numéro a_0, on regarde le numéro a_1 qu'elle contient et on prend l'enveloppe portant ce numéro a_1 ; on regarde alors le numéro a_2 qu'elle contient, et ainsi de suite.

Dans l'exemple proposé, en partant de l'enveloppe 5, on construit ainsi la séquence $5 \mapsto 3 \mapsto 7 \mapsto 1 \mapsto \ldots$.

5. Proposer une fonction `int max_incr(void)` renvoyant la longueur de la plus longue séquence a_i strictement croissante que l'on puisse construire de la sorte (3 pour notre exemple, $a_0 = 1, a_1 = 3, a_2 = 7$). On ne demande pas ici d'obtenir une complexité optimale, faute d'outils adéquats.

6. Quelle est la complexité dans le pire des cas de votre fonction ? On décrira le contenu des enveloppes qui conduirait à cette complexité.

Un *cycle de longueur* $p \geqslant 2$ est une séquence $a_0 \mapsto a_1 \mapsto a_2 \mapsto \ldots \mapsto a_{p-1} \mapsto a_0$ où tous les a_i sont distincts.

Toujours dans notre exemple, $1 \mapsto 3 \mapsto 7 \mapsto 1$ est un cycle de longueur 3.

7. Proposer une fonction `int len_cycle(int v)` prenant en argument un entier $v \in [\![0\,..\,n-1]\!]$ et renvoyant (on se contentera d'une implémentation simple et d'une complexité en $O(n)$) :

- la longueur du cycle dont fait partie l'enveloppe portant le numéro v, si cette enveloppe fait effectivement partie d'un cycle ;
- la valeur 0 dans le cas contraire.

On suppose que le contenu de chacune des enveloppes est un entier distinct de $[\![0\,..\,n-1]\!]$. Il existe alors un entier r tel que, *quel que soit le a_0 que l'on choisisse*, la séquence $a_0 \mapsto a_1 \mapsto a_2 \mapsto \ldots \mapsto a_r$ (définie comme au-dessus) vérifie $a_r = a_0$.

8. Proposer une fonction `int order(void)` ne prenant aucun argument et retournant cet entier r. On ne demande pas de complexité particulière (faute des outils adéquats), $O(n^2 \log r)$ est parfaitement acceptable ici. Il pourra être utile, pour cette question, de définir une fonction auxiliaire pour rendre les choses plus lisibles.

2 Correction algorithmique du gnome sort

On propose un algorithme dont on espère qu'il permet de trier un tableau `arr` contenant `n` éléments dans l'ordre croissant :

```c
int i = 0;
while (i < n) {
    if (i==0 || arr[i] >= arr[i-1]) {
        i = i+1;
    } else {
        int tmp = arr[i];
        arr[i] = arr[i-1];
        arr[i-1] = tmp;
        i = i-1;
    }
}
```

1. Combien d'itérations de la boucle `while` sont effectuées si le tableau est déjà trié par ordre croissant ?

2. Proposer un invariant de boucle permettant de justifier que l'algorithme est partiellement correct (en d'autres termes, si la boucle `while` se termine, alors le tableau `arr` a été trié par ordre croissant par l'algorithme). Indice : c'est le même invariant de boucle qu'un des tris étudiés en cours. On justifiera (en quelques lignes) que cet invariant de boucle est correct.

On définit le *nombre d'inversions* dans un tableau de n entiers comme le nombre de couples $(i, j) \in [\![0\,..\,n-1]\!]^2$ vérifiant `i<j` et `arr[i]>arr[j]`.

3. Donner un encadrement des valeurs possibles pour le nombre d'inversions dans un tableau de taille n contenant des valeurs quelconques.

4. Justifier que durant l'exécution de l'algorithme, le nombre d'inversions dans le tableau `arr` ne peut que décroître.

5. En déduire (soigneusement) la terminaison de l'algorithme.

6. Déterminer la complexité temporelle (dans le pire des cas) de l'algorithme proposé.

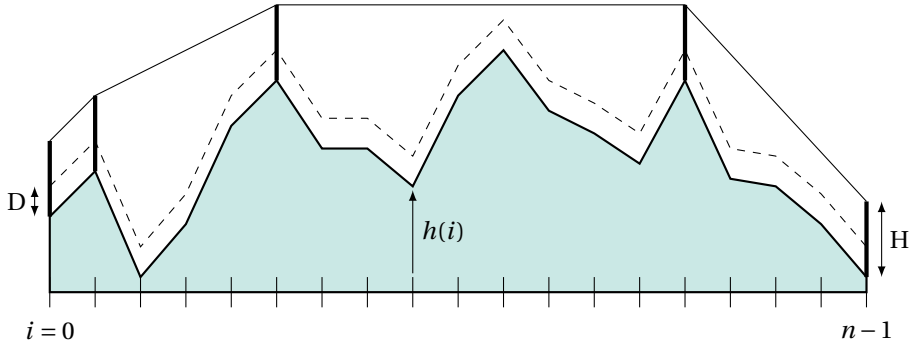
3 Pose de poteaux

3.1 Description du problème

On souhaite, dans cet exercice, poser des poteaux électriques sur un terrain accidenté. Le terrain est modélisé par une ligne brisée de points $(i, h(i))$ où $i \in [\![0\,..\,n-1]\!]$. Le but étant de relier le point d'abscisse 0 au point d'abscisse $n-1$.

Les poteaux, posés verticalement, ont chacun une hauteur H, et on supposera que la ligne électrique posée entre les poteaux est parfaitement tendue[1]. La réglementation impose que le fil soit toujours à une distance, mesurée verticalement, supérieure ou égale à D du sol (on suppose évidemment H ⩾ D).

Il y a toujours un poteau aux points $i = 0$ et $i = n-1$, et il peut y avoir un nombre quelconque de poteaux entre les deux. Un exemple d'implantation est illustré ci-dessous :



Dans la suite, des constantes `H` et `D` sont considérées définies sous la forme de valeurs flottantes (de type `double`) globales, donc disponibles dans toutes les fonctions. Il est également fourni une fonction `double h(int i)` prenant en argument une abscisse entière `i` et retournant l'altitude du terrain $h(i)$ pour cette abscisse, qui peut, elle aussi, être appelée depuis n'importe quelle fonction. **Ce sont les seules informations supposées disponibles à l'intérieur d'une fonction, exception faite des arguments de la fonction.**

3.2 Premiers tests et fonctions

1. Proposer une fonction `double hauteur(int i, int j, int k)` prenant en argument trois entiers `i`, `j` et `k` vérifiant $i \leqslant k \leqslant j$. `i` et `j` représentent l'abscisse de deux poteaux *c*onsécutifs. La fonction devra retourner la hauteur du fil par rapport au sol au point d'abscisse `k`.

2. Justifier brièvement que pour vérifier la validité d'une installation, il suffit de vérifier que la condition sur la hauteur du fil par rapport au sol est vérifiée pour toutes les abscisses entières.

1. En réalité, la forme dessinée par le fil entre deux supports est celle d'un cosinus hyperbolique.

3. Proposer une fonction `bool direct_possible(int i, int j)` prenant deux abscisses i et j vérifiant $i < j$, et retournant un booléen indiquant s'il est légal de tirer un cable entre deux poteaux placés aux abscisses `i` et `j`.

On suppose que l'on dispose d'un tableau `idx` de p entiers contenant la position de chacun des poteaux que l'on souhaite poser. Conformément aux hypothèses, on suppose `idx[0] = 0`, `idx[p-1] = n-1` et pour tout $i < j$, `idx[i] < idx[j]`. Pour l'implantation prise en exemple, le tableau serait donc :

idx	0	1	5	14	18

4. Proposer une fonction `bool allowed(int idx[], int p)` retournant un booléen indiquant si l'implantation fournie en argument est valide. On admettra que, dans une telle fonction il est possible d'utiliser l'écriture `idx[k]` pour obtenir la valeur dans la case `k` du tableau fourni comme premier argument de la fonction.

5. Proposer une fonction `double length(int idx[], int p)` retournant un flottant indiquant la longueur de cable nécessaire pour l'implémentation proposée en argument.

Dans la suite, on suppose qu'il n'est pas possible de tirer un cable directement entre les poteaux placés aux abscisses 0 et $n-1$. On souhaite, tout d'abord, savoir s'il est possible d'obtenir une implémentation légale en ajoutant un *unique* poteau supplémentaire à une abscisse `k` vérifiant $0 < k < n-1$.

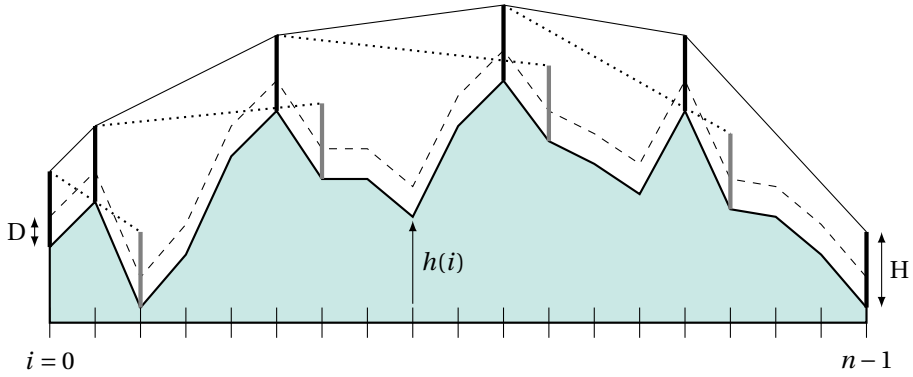
6. Proposer une fonction `int possible_3(int n)` prenant `n` pour seul argument et retournant un entier k représentant une abscisse `k` pour un poteau intermédiaire s'il est possible d'effectuer une installation avec trois poteaux, et `-1` sinon. Si plusieurs solutions sont possibles pour `k`, on retournera une quelconque de ces solutions, au choix.

7. Quelle est la complexité de la fonction précédente ?

3.3 Recherche de solution

On souhaite à présent trouver une implémentation légale utilisant un nombre quelconque de poteaux, mais sans aller jusqu'à en poser pour chaque abscisse entière. On propose l'algorithme suivant, appelé *glouton avant* : les poteaux sont posés par abscisses croissantes, le premier poteau étant placé en 0. Pour calculer l'emplacement du prochain poteau, on part du dernier poteau planté et on avance (vers les abscisses croissantes) avec le fil tendu tant que la législation est respectée (et que l'abscisse $n-1$ n'est pas atteinte). Un nouveau poteau est alors planté, et on recommence jusqu'à atteindre l'abscisse $n-1$.

La figure ci-après illustre la solution produite par cet algorithme. Les poteaux gris et les morceaux de ligne électrique en pointillés indiquent les implantations étudiées par l'algorithme qui ne sont pas conformes à la réglementation.



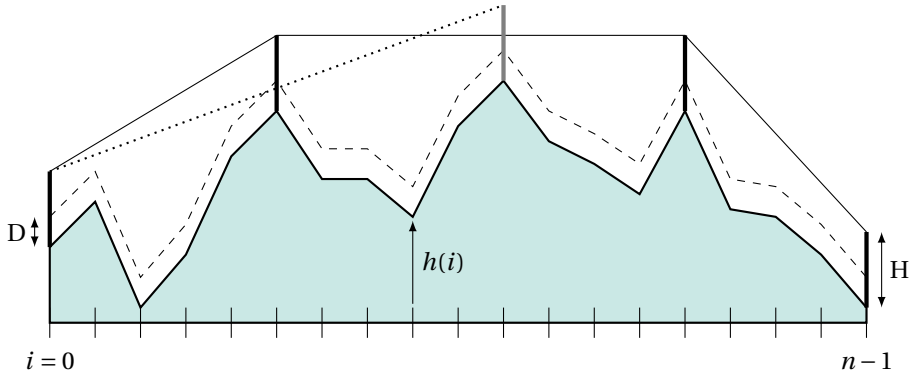
8. Proposer une fonction `int nb_posts_forward(int n)` prenant `n` pour seul argument et retournant le nombre de poteaux *intermédiaires* (on ne compte pas les poteaux en $i = 0$ et $i = n-1$) nécessaires pour poser le cable en utilisant l'algorithme du « glouton avant ». On ne demande pas de complexité particulière, ni de fournir la position des poteaux, la fonction retournera par exemple simplement « 4 » dans l'exemple précédent.

9. Quelle est la complexité, en fonction de n de la fonction proposée ?

10. Décrire une approche permettant d'obtenir une complexité linéaire ($O(n)$) pour la fonction précédente. **On ne demande pas ici d'implémentation, juste une explication détaillée**.

La méthode gloutonne avant a pour défaut de placer plus de poteaux que nécessaire, comme on pourra le constater sur l'exemple proposé. Un autre algorithme possible est l'algorithme de *glouton arrière*. on place toujours les poteaux de la gauche vers la droite, mais on choisit à chaque fois la position légale la plus à droite possible pour le prochain poteau.

Cela donne par exemple ce résultat pour le terrain utilisé comme exemple :



11. Proposer une fonction `int nb_posts_backwards(int n)` prenant `n` pour seul argu-

ment et retournant le nombre de poteaux *intermédiaires* (on ne compte pas les poteaux en $i = 0$ et $i = n - 1$) nécessaires pour poser le cable en utilisant l'algorithme du « glouton arrière ». On ne demande pas de complexité particulière, ni de fournir la position des poteaux, la fonction retournera par exemple simplement « 2 » dans l'exemple précédent.

12. Le nombre de poteaux retournés par la fonction est-il plus petit possible pour une implémentation licite ? Le prouver ou proposer un contre-exemple.