Devoir d'informatique

1 Enveloppes

1. On compte dans un accumulateur `count` les enveloppes avec la propriété souhaitée :

```c
int nb_inf(void) {
    int count = 0;
    for (int i=0; i<N; ++i) {
        // count contient le nombre d'enveloppes avec la propriété
        // parmi les i premières enveloppes
        if (content(i) < i) { count++; }
    }
    return count;
}
```

2. Il s'agit ici d'une variation sur la recherche d'un maximum d'un ensemble :

```c
int max_diff(void) {
    int max = 0;
    for (int i=0; i<N; ++i) {
        // count contient la plus grande différence en valeur absolue
        // parmi les i premières enveloppes
        if (abs(content(i)-i) > max) { max = abs(content(i)-i); }
    }
    return max;
}
```

3. Puisque l'on souhaite la plus grande enveloppe vérifiant la propriété, il est plus simple de faire la recherche en commençant par les enveloppes portant les nombres les plus élevés, donc de l'enveloppe $N-1$ jusqu'à trouver (auquel cas on renvoie le résultat de suite), ou bien les avoir toutes examinées jusqu'à l'enveloppe 0 incluse :

```c
int max_over(int v) {
    for (int i=N-1; i>=0; --i) {
        if (content(i) >= v) { return i; }
    }
    return -1;
}
```

4. Pour une paire (i, j), il faut $i \neq j$ avec $\texttt{content}(i) = j$ et $\texttt{content}(j) = i$, soit $\texttt{content}(\texttt{content}(i)) = i$.

Pour éviter de compter deux fois la même paire, on ne comptera que les cas où $j > i$, soit $\texttt{content}(i) > i$:

```c
int nb_pairs(void) {
    int count = 0;
    for (int i=0; i<N; ++i) {
        // count contient le nombre de paires (i1, j1) avec i1<j1
        // pour lesquelles i1 figure parmi les i premières enveloppes
        if (content(i)>i && content(content(i))==i) { count++; }
    }
    return count;
}
```

5. On considère ici tous les a_0 possible, et pour chacun on détermine la longueur de la séquence strictement croissante. Attention aux indices ! Bien évidemment, on effectue des calculs inutiles en recomptant des morceaux de séquences déjà comptées, mais comme le sujet le suggère, on ne s'en préoccupe pas ici.

```c
int max_incr(void) {
    int max = 0;
    for (int i=0; i<N; ++i) {
        // max contient la longueur de la plus longue séquence
        // débutant par une des i premières enveloppes
        // (ou 0 si i=0, par défaut)
        int count=1;
        for (int j=i; content(j)>j; j=content(j)) {
            count++;
        }
        // count contient la longueur de la plus longue séquence
        // strictement croissante débutant par l'enveloppe i
        if (count > max) { max = count; }
    }
    return max;
}
```

6. Aucune séquence ne peut être plus longue que n, et on considère n points de départ. La complexité dans le pire des cas est ici quadratique ($O(n^2)$), cette limite étant effectivement atteinte si chaque enveloppe i contient $i+1$, à l'exception de l'enveloppe $n-1$. Dans ce cas, les points de départ 0 à $n/2$ nécessitent chacun au moins $n/2$ itérations, ce qui conduit bien à une complexité quadratique.

7. On remarquera que si `content`$(i) = i$, il ne s'agit pas d'un cycle (on renverra 0). Par ailleurs, aucun cycle ne peut être supérieur strictement à n, donc si on n'est pas revenu au point de départ après n enveloppes, c'est que l'enveloppe initiale n'est pas dans un cycle. Cela donne par exemple :

```
int len_cycle(int v) {
    int i = content(v);
    if (i!=v) {
        for (int k=2; k<=n; ++k) {
            i = content(i);
            if (i==v) { return k; }
        }
    }
    return 0;
}
```

8. Le r recherché est le PPCM de tous les cycles. On commence par définir une fonction calculant le PGCD :

```
int gcd(int a, int b) {
    while (a!=0) {
        int tmp = a; a = b%a; b = tmp;
    }
    return b;
}
```

Puis un PPCM :

```
int lcm(int a, int b) {
    return a/gcd(a, b)*b;
}
```

Et enfin (attention à ignorer les 0 renvoyés par `len_cycle`!) :

```
int order(void) {
    int r = 1;
    for (int i=0; i<N; ++i) {
        p = len_cycle(i);
        if (p>0) { r = lcm(r, p); }
    }
    return r
}
```

2 Correction algorithmique du gnome sort

1. Si le tableau est déjà trié par ordre croissant, la condition du **`if`** sera toujours vérifiée, et `i` incrémenté à chaque itération. Il y aura donc exactement `n` itérations de la boucle **`while`**.

2. On peut proposer l'invariant suivant : « les `i` premiers éléments de `tab` (dans les cases d'index 0 à $i-1$ (inclus) si $i > 0$) sont rangés par ordre croissant ». Cette proposition est vraie au début et à la fin de chaque itération de la boucle **`while`**. Pour le montrer, remarquons d'abord qu'initialement, au début de la première itération de la boucle **`while`** c'est nécessairement vrai ($i = 0$). Ensuite, si c'est vrai au début de la boucle :

- si $i = 0$ au début de la boucle, on passe à $i = 1$ à l'issue de la boucle, et un élément seul est toujours trié ;
- sinon, si les éléments dans les cases d'index 0 à $i-1$ (inclus) sont triés par ordre croissant, et que `tab[i] >= tab[i-1]`, on peut en déduire que les éléments d'index 0 à i (inclus) sont rangés par ordre croissant, ce qu'exprime l'incrémentation de `i` ;
- et enfin, si `tab[i] < tab[i-1]`, la permutation ne touche pas aux positions des éléments dans les cases d'index 0 à $i-2$, lesquels sont par conséquent toujours triés par ordre croissant, ce qui est en accord avec une décrémentation de `i`.

3. Il y a $\binom{2}{n} = n(n-1)/2$ couples $(i, j) \in [\![0\,..\,n-1]\!]^2$ vérifiant `i<j`. Ils peuvent tous vérifier `tab[i]<=tab[j]` (tableau trié par ordre croissant) ou tous vérifier `tab[i]>tab[j]` (trié par ordre décroissant), donc le nombre d'inversions est nécessairement compris entre 0 et $n(n-1)/2$ (inclus).

4. Si la condition du test « **`if`** » est vraie, rien ne change dans le tableau, donc le nombre d'inversions n'évolue pas. Si elle est fausse, il y avait une inversion entre les cases i et $j = i-1$, qui disparaît avec l'échange. Pour les autres couples (i, j), la situation n'a pas changé (même si l'un des éléments du couple a pu être décalé d'une case par l'inversion, il ne peut se retrouver de l'autre côté du second élément), donc le nombre d'inversions décroit exactement de 1. Dans tous les cas, le nombre d'inversions dans le tableau décroit (au sens large).

5. On ne peut pas prendre directement le nombre d'inversions comme variant de boucle, car bien qu'étant toujours un entier positif, sa décroissance n'est pas stricte à chaque itération ! Cela dit, on peut s'en sortir en remarquant qu'entre deux décrémentation de cette grandeur, à chaque itération, `i` augmente de 1 (et ne peut dépasser n), donc il ne peut y avoir plus de n itérations sans décrémentation du nombre d'inversions, ce qui garantit que l'algorithme va nécessairement terminer.

On peut être plus malin en remarquant que $2\mathrm{N} - i$, où N est le nombre d'inversions, est un variant de boucle : si la condition du « **`if`** » est vraie, N ne change pas et `i` est incrémenté, donc $2\mathrm{N} - i$ est décrémenté. Si la condition du « **`if`** » est fausse, N est décrémenté et i est décrémenté, de sorte que $2\mathrm{N} - i$ est décrémenté. $2\mathrm{N} - i$ étant une valeur dans $[\![-n\,..\,n(n-1)+n]\!]$, il ne peut y avoir une infinité d'itération de la boucle **`while`**, donc la fonction termine.

6. Comme on ne décrémente pas N à chaque itération, il n'est pas immédiat d'affirmer que le nombre d'itérations est lié au nombre d'inversions (d'ailleurs, ce n'est pas vrai pour un tableau trié).

Si on a été assez astucieux pour penser au $2N - i$ dans la question précédente, on peut affirmer que la complexité dans le pire des cas est $\Theta(n(n-1)+n) = \Theta(n^2)$ car on part de $2N - i$, soit $n(n-1)$ si la liste est triée par ordre décroissant, et on termine à $-n$, en décémentant à chaque itération de la boucle.

Sinon, on peut s'apercevoir que le tri est en fait un tri par insertion, avec toutefois une petite différence : après avoir inséré l'élément initialement en position i jusqu'à la position $i' < i$, avant de traiter l'élément en position $i+1$, il faut « revenir » à cette position $i+1$ en effectuant autant de tests dans l'autre sens. On a donc grossièrement deux fois plus de comparaisons que pour le tri par insertion, ce qui donne une complexité dans le pire des cas en $\Theta(n^2)$.

3 Pose de poteaux

1. En utilisant le théorème de Thalès, on peut par exemple écrire [1] :

```
double hauteur(int i, int j, int k) {
    if (i==j) { return H; }
    double slope = (h(j)-h(i))/(j-i);
    return h(i) + H + slope * (k-i) - h(k);
}
```

Remarque : par « hauteur par rapport au sol », on entendait la différence entre la hauteur du fil à l'abscisse k et $h(k)$ (l'altitude référence $h = 0$ pourrait se rapporter à n'importe quoi, il n'y a pas de raison qu'elle fasse référence au « sol »). Le sujet impose bien une hauteur du fil au moins égale à D par rapport au sol, avec une représentation graphique de cette marge, ce qui permet de lever une éventuelle ambiguïté sur le terme.

Si, suite à un malentendu, on retourne l'altitude par rapport à l'altitude de référence 0, en ne retirant pas `h(k)` du résultat, il faudra penser à tenir compte de ce $h(k)$ dans les fonctions ultérieures.

2. Entre deux abscisses entières, l'évolution de l'altitude du fil (tendu) et celle du sol sont deux fonctions affines de l'abscisse. La hauteur du fil par rapport au sol est donc une fonction affine également. Si cette hauteur est supérieure ou égale à D en i et $i+1$, alors elle sera nécessairement supérieure ou égale à D pour toute abscisse comprise entre i et $i+1$, il est donc bien inutile de le vérifier.

1. On s'est prémuni ici du cas `i=j` qui aurait pu conduire à une forme indéterminée, même si dans la suite la fonction ne sera normalement jamais appelée avec `i=j`.

3. On teste toutes les abscisses entières k entre $i+1$ et $j-1$ (pas besoin de tester i et j puisque $H \geqslant D$). Cela donne :

```
bool direct_possible(int i, int j) {
    for (int k=i+1; k<=j-1; ++k) {
        if (hauteur(i, j, k) < D) {
            return false;
        }
    }
    return true;
}
```

4. On peut utiliser la fonction précédente pour chaque section du fil, entre chaque paire de poteaux successifs [2] :

```
bool allowed(int idx[], int p) {
    for (int i=0; i<p-1; ++i) {
        if (!direct_possible(idx[i], idx[i+1])) {
            return false;
        }
    }
    return true;
}
```

5. Rien de bien compliqué ici, on applique le théorème de Pythagore (attention, il n'y a pas d'opérateur puissance en C!), avec la même remarque sur les paramètres de la boucle **for** :

```
double length(int idx[], int p) {
    double len = 0.0;
    for (int i=0; i<p-1; ++i) {
        double dx = idx[i+1]-idx[i];
        double dy = h(idx[i+1])-h(idx[i]);
        len = len + sqrt(dx*dx + dy*dy);
    }
    return len;
}
```

6. Il nous faut ici tester toutes les positions k possibles pour le pilier intermédiaire.

2. On ne se préoccupera pas ici de contrôler que l'utilisateur a bien mis 0 dans la première case et $n-1$ dans la dernière, on ne pourrait d'ailleurs de toute façon pas accéder à la valeur de n. En revanche, on prendra bien soin, en choisissant les paramètres de la boucle, de tester la condition jusqu'à la dernière case du tableau `idx` mais pas au-delà! Ainsi, dans la fonction proposée, lors de la dernière itération on a `i=p-2`, ce qui conduit à un test entre les positions `idx[p-2]` et `idx[p-1]`, cette dernière position étant bien la dernière présente dans le tableau.

Il serait parfaitement possible d'utiliser la fonction `allowed`, en créant un tableau de taille 3 contenant 0, k et $n-1$, où k prend successivement les valeurs 1 à $n-2$.

Attention, il n'existe pas de moyen, en C, d'écrire directement, dans les arguments, un tableau que l'on pourrait passer à la fonction, il faut le déclarer explicitement avant l'appel. Plutôt que de déclarer un tableau à chaque itération, on a pris le parti ici de déclarer un unique tableau pour *toute* la fonction, et de modifier le contenu de la seconde case pour chaque itération.

```c
int possible_3(int n) {
    int idx[3] = {0, 1, n-1};
    for (int k=1; k<=n-2; ++k) {
        idx[1] = k;
        if (allowed(idx, 3) {
            return k;
        }
    }
    return -1;
}
```

Cela étant dit, pour cette question, il est probablement plus simple d'utiliser deux appels à `direct_possible` :

```c
int trois_possible(int n) {
    for (int k=1; k<=n-2; ++k) {
        if (direct_possible(0, k) && direct_possible(i, n-1)) {
            return k;
        }
    }
    return -1;
}
```

7. Lors de chaque itération, on effectue deux appels à `direct_possible` (directement, ou indirectement à travers la fonction `allowed`), dont la complexité est linéaire en la distance entre les poteaux concernés. La somme des deux distances $0 \to k$ et $k \to n-1$ est égale à n. Chaque itération a donc une complexité linéaire. Puisque dans le pire des cas on effectue $n-2$ itérations, on a donc pour `possible_3` une complexité quadratique ($O(n^2)$) en n.

8. On applique l'algorithme proposé par l'énoncé : on considère toutes les positions k de 1 à $n-1$ et, s'il n'est pas possible d'attendre la position $k+1$ en partant du dernier poteau posé (initialement le poteau en $n=0$), il faut poser un poteau en k.

Pour s'assurer de ne pas écrire les choses de travers, il est recommandé de réfléchir à un invariant de boucle !

Notons qu'il est impératif qu'il y ait au moins un appel à `direct_possible` dont le second argument est $n-1$. Il faut en effet envisager de poser un poteau en $n-2$! On sait en revanche qu'il est toujours possible d'atteindre l'abscisse 1 depuis le poteau initial.

Cela donne par exemple :

```c
int nb_posts_forward(int n) {
    int last = 0;
    int count = 0;
    for (int k=1; k<n-1; ++i) {
        // Inv. : On peut atteindre l'abscisse k en ligne directe
        // depuis l'abscisse last correspondant au dernier poteau posé
        if (!direct_possible(last, k+1)) {
            last = k;
            count++;
        }
    }
    return count;
}
```

Remarque : il peut être intéressant, pour s'assurer que la fonction est bien écrite, de regarder combien il est possible de comptabiliser de poteaux intermédiaires pour cette fonction. Si la condition du **if** n'est jamais vérifiée, le compte sera de 0. Si elle l'est toujours, il sera de $n-2$. Il s'agit bien des nombres minimum et maximum de poteaux que l'on peut avoir à poser, c'est un indice supplémentaire que la fonction peut être correcte.

9. Si l'on pose un poteau en k et un poteau supplémentaire en k', les itérations de k à k' auront une complexité quadratique en $k'-k$ du fait des appels à `direct_possible`. La complexité de `nb_intermediaires_avant` est donc de l'ordre de la somme des carrés des longueurs de chaque section du cable. Cette somme est, dans le pire des cas, de l'ordre de n^2 (par exemple s'il est inutile de poser un poteau intermédiaire). On a donc une complexité quadratique ($O(n^2)$) en n.

10. On peut obtenir une complexité linéaire en remarquant qu'il est possible, lorsque l'on cherche le k' où poser le prochain poteau après celui en k, qu'il n'est en fait nécessaire de vérifier qu'une seule altitude à chaque itération.

Si l'on note M_p le point correspondant à position la plus basse tolérée pour le fil à l'abscisse p, et I_k la position du sommet du dernier poteau posé, il est aisé de voir que parmi tous les tests pour chaque p, si celui pour le p tel que l'angle $\overrightarrow{\mathrm{I}_k\mathrm{M}_p}$, mesuré par rapport à l'horizontale dans le sens direct, est le plus grand possible est réussi, alors tous les autres le seront également.

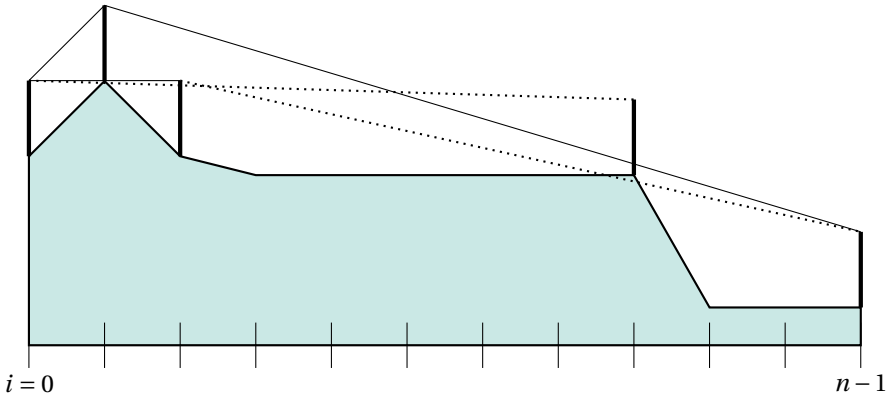
Il suffit donc de mémoriser et tenir à jour à chaque itération le p correspondant, ce qui permet de ne faire qu'un seul test à chaque itération de cette même boucle.

11. À nouveau, on implémente l'algorithme proposé :

```c
int nb_posts_backward(int n) {
    int last = 0;
    int count = 0;
    while (last < n-1) {
        // On cherche la position k la plus à droite
        // possible que l'on puisse atteindre depuis
        // le dernier poteau posé
        int k = n-1;
        while (!direct_possible(last, k) { k--; }
        // On pose un nouveau poteau à la position identifiée
        last = k;
        count++;
    }
    return count - 1;
}
```

La boucle **while** va forcément trouver une position k qui convienne, puisque dernier_pose+1, au moins, est une possibilité valide. Comme last est un entier qui augmente strictement à chaque itération et majoré par n-1, on a la terminaison de notre fonction (non demandée). Notons que l'on retire 1 au compteur à l'issue de la fonction, car le poteau en $n-1$ a été compté dans la boucle **while**.

12. Le nombre de poteaux retournés par la fonction n'est pas nécessairement le petit possible pour une implémentation licite, mais le contre-exemple n'est pas trivial à construire ni même à tracer proprement. On peut par exemple supposer D = 0 pour simplifier, et considérer la situation suivante :



Elle correspond au tableau des hauteurs ci-dessous [3] avec H = 1 :

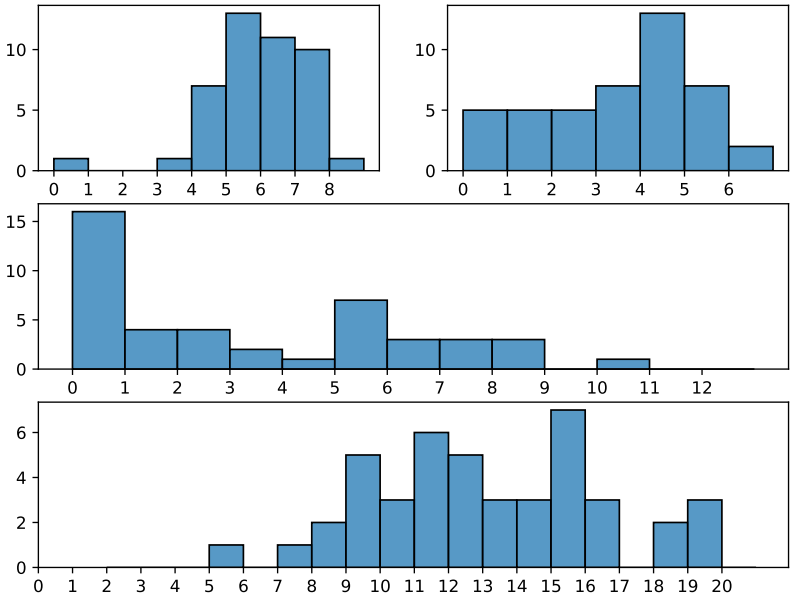
idx	0.0	0.995	0.0	-0.25	-0.25	-0.25	-0.25	-0.25	-0.25	-2.0	-2.0	-2.0

L'algorithme considéré va placer un premier poteau en $k = 2$ (on peut vérifier qu'il est possible de l'atteindre depuis le point de départ, et que toutes les positions d'abscisse $k > 2$ ne le sont pas).

Seulement, il n'est pas possible d'atteindre directement le poteau en $n-1$ depuis un poteau placé en $k = 2$, car on a un souci en $k = 8$. Il faudra donc un second poteau intermédiaire (qui sera placé par l'algorithme à cette même abscisse $k = 8$).

Et pourtant, il était possible de lier les deux extrémités en n'utilisant qu'un seul poteau intermédiaire, placé en $k = 1$.

Résultats



3. Le schéma suffisait, il n'est pas nécessaire de fournir un tableau de hauteurs. Il n'est indiqué ici que pour ceux qui voudraient vérifier numériquement que le cas proposé est effectivement problématique.