Bug bounty : toute erreur signalée dans le corrigée peut rapporter des points, en fonction de sa gravité. Pour les erreurs les plus mineures (fautes de frappe / français), je ne paye pas à la faute mais à l'octet (arrondi à la valeur supérieure) de fautes.

Solutions des exercices

Solution de l'Exercice 1 – *Here we go again*

Dans cet exercice, et dans tout ce sujet, on note sans prime (e.g. `k`) la valeur d'une quantité au début d'une itération, et avec prime (e.g. `k'`) la valeur en fin de cette itération (c'est à dire au début de l'itération suivante, si elle existe).

1. **a.** Le prototype est : `int somme_entiers(int n)`.

---

b. Voici la spécification de `somme_entiers` :

- Entrées : `n` un entier `int`.
- Sorties : $\sum_{k=0}^{n} k$. Si $n \leq 0$, cette somme est nulle.
- Effets secondaires : Aucun.

---

2. Cette fonction ne réalise aucun appel de fonction et contient une seule boucle. Prouvons que celle-ci termine en en exhibant un variant.
La quantité `i` est :

- Entière car `i` est déclaré comme `int i` (ligne 10).
- Majorée par la valeur de `n` d'après `i <= n` (ligne 12). Comme la valeur de `n` ne change pas dans le code, ceci est bien une majoration par une borne constante.
- Strictement croissante car (ligne 14) `i' = i + 1 > i`.

Cette quantité est donc un variant de la boucle `while`, qui termine donc. D'où :

`somme_entiers` termine.

---

3. La fonction n'a pas d'effets secondaires, comme attendu. Montrons qu'elle renvoie la bonne valeur, c'est à dire qu'elle renvoie $\sum_{i=0}^{n} i$. Il suffit de montrer qu'en sortie de boucle, `somme` vaut cette valeur (puisque c'est celle qui est renvoyée).
Pour prouver cela, montrons que I : «`s` $= \sum_{k=0}^{\mathtt{i}-1} k$» est un invariant.

- Initialisation : avant la première itération de la boucle, on a `s` $= 0$. On a également `i` $- 1 < 0$, d'où la somme de I est vide et vaut donc 0. On a bien $0 = 0$: l'invariant est initialisé.
- Conservation : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que I' : « `s'` $= \sum_{k=0}^{\mathtt{i}'-1} k$ ».
  On a :

$$\begin{cases} \mathtt{s}' = \mathtt{s} + \mathtt{i} & \text{(ligne 13)} \\ \mathtt{i}' = \mathtt{i} + 1 & \text{(ligne 14)} \end{cases}$$

  Or d'après I, on a `s` $= \sum_{k=0}^{\mathtt{i}-1} k$. Donc :

$$\begin{aligned} \mathtt{s}' &= \mathtt{s} + \mathtt{i} \\ &= \left(\sum_{k=0}^{\mathtt{i}-1} k\right) + \mathtt{i} && \text{d'après } I \\ &= \sum_{k=0}^{\mathtt{i}} k \\ &= \sum_{k=0}^{\mathtt{i}'-1} k && \text{car } \mathtt{i}' = \mathtt{i} + 1 \end{aligned}$$

  On a prouvé que I' est vrai : l'invariant se conserve.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme à la fin de la dernière itération on a `i` $= \mathsf{n}+1$, on a en sortie de boucle :

$$\begin{aligned}\mathsf{s} &= \sum_{k=0}^{\mathsf{n}+1-1} k \\ &= \sum_{k=0}^{\mathsf{n}} k\end{aligned}$$

Il s'ensuit que :

`somme_entiers` est partiellement correcte.

---

4. Puisque la fonction termine et est partiellement correcte :

`somme_entiers` est totalement correcte.

---

5. La fonction ne contient pas d'appels de fonctions. En dehors de sa boucle, la fonction effectue 0 additions. Comptons le nombre d'additions effectuées par la boucle.

Chaque itération de la boucle effectue 2 additions (ligne 13 et ligne 14). Chaque évaluation de la condition effectue 0 condition. La boucle itère sur `i` $\in [\![1;\mathsf{n}]\!]$. Le nombre d'addition total de la boucle est donc $\mathsf{n}.2 + (\mathsf{n}+1).0 = 2(\mathsf{n}+1)$.

Donc :

`somme_entiers` effectue $2\mathsf{n}$ additions.

---

6. D'après les données rappelées dans l'énoncé, le type `int` peut stocker les valeurs $[\![-2^{32/2}; 2^{32}/2[\![$ c'est à dire $[\![-2^{31}; 2^{31}[\![$.

Il y aura donc un surpassement de capacité lorsque $\sum_{k=0}^{n} l \geq 2^{31}$, c'est à dire lorsque $\frac{n(n+1)}{2} \geq 2^{31}$, donc quand :

$$n^2 + n - 2^{32} \geq 0$$

Le discriminant de ce polynôme est :

$$\Delta = 1 + 2^{34} > 0$$

Ses deux racines sont donc :

$$\frac{-1-\sqrt{1+2^{34}}}{2} \simeq \frac{-1-2^{17}}{2} \simeq -2^{16} \text{ et } \frac{-1+\sqrt{1+2^{34}}}{2} \simeq 2^{16}$$

Comme le coefficient dominant du polynôme de degré 2 est positif, le polynôme est négatif entre ses racines et positif en dehors. Donc :

Pour $n \geq 0$, la variable `somme` débordera à peu près à partir de $n = 2^{16}$.

Solution de l'Exercice 2 – *Palindrome*

1. On applique l'indication de l'énoncé :

palindrome.c

```c
/** Renvoie la longueur d'une chaine de caractère. */
unsigned int length(char* str) {
  unsigned l = 0;
  while (str[l] != '\0') {
    l += 1;
  }
  return l;
}
```

---

2. La fonction ci-dessous teste si le premier caractère au égal au dernier, le second à l'avant-dernier, etc ; autrement dit au fait que le caractère d'indice `0` est égal au caractère d'indice `l-1`, puis que le caractère d'indice 1 est égal au caractère d'indice `l-2`, etc.

palindrome.c

```c
/** Renvoie true ssi str est un palindrome. */
bool est_palindrome(char* str) {
  unsigned l = length(str);
  unsigned i = 0;
  while (i <= l/2) {
    if (str[i] != str[l-1-i]) { return false; }
    i = i +1;
  }
  return true;
}
```

---

3. Si l'on fait cette modification, le pointeur pointe désormais vers des `char const` c'est à dire vers des caractères non modifiables. Cela permet de garantir que la fonction ne modifie par le texte passé en argument.

   NB : la fonction ne modifie déjà pas - et il n'y aurait donc pas besoin de la modifier outre son prototype -, mais l'ajout de `const` permettra de le *garantir*.

   NBB : une chaîne de caractère crée en l'écrivant entre guillemets doubles (comme `"Camille !"`) est déjà non-modifiable, mais ce sont là des arcanes de C que je ne vous ai pas encore enseignées. Cependant, rien ne garantit que le pointeur passé à `est_palindrome` a été créé ainsi, et l'ajout de `const` reste donc pertinent.

Solution de l'Exercice 3 – *Trouver la star*

1. Raisonnons par l'absurde et supposons qu'il existe au moins deux stars. Alors par définition deux stars distinctes se connaissent mutuellement (car tout le monde les connait), mais ne sont donc plus des stars (car elles connaissent quelqu'un d'autre). Absurde.

---

2. **a.** Voici une fonction qui vérifie si une personne est une star en $n-1$ questions :

Fonction Vérification
Entrées : `G` un groupe, `x` une personne
Sorties : `true` si la personne est une star, `false` sinon

```
tant que on n'a pas interrogé toutes les personnes autres que x faire
    y ←une personne non interrogée
    si y ne connait pas x ou x connait y alors
        renvoyer false
renvoyer true
```

Avant de passer à la question suivante, calculons rapidement la complexité de cette fonction (cela servira). La boucle itère autant de fois qu'il y a de personnes autre que `x`, donc `n-1` fois. À chaque itération, on effectue 2 questions. Il n'y a pas de questions dans la condition ou hors de la boucle : cette fonction effectue donc $2(n-1)$ questions.

---

b. Voici une fonction qui résout le problème en $(n-1)n = \Theta(n^2)$ questions. Pour cela, on teste pour chacune des personnes si elle est une star :

Fonction StarNaif

Entrées : G un groupe
Sorties : la star de G si elle existe, ou « pas de star » sinon

```
1 tant que on n'a pas testé toutes les personnes faire
2 |  x ←—une personne non testé
3 |  si Vérification(x) alors
4 |  |  renvoyer x
5 renvoyer « pas de star »
```

c. La fonction n'effectue pas de question hors de sa boucle. Sa boucle itère une fois par personne. À chaque itération, les seules questions posées le sont par l'appel de fonction, qui effectue $2(n-1)$ questions. Au final :

La fonction `StarNaif` effectue $\frac{n(n-1)}{2}$ questions.

3. a. Si x connait y, alors x n'est pas une star. Sinon, y n'est pas connu par x et n'est donc pas une star.

b. La fonction `Sélection` « élimine » à chaque tour une personne grâce au raisonnement indiqué par la question précédente. Une star ne peut pas être éliminée par ce raisonnement : ainsi, s'il existe une star, elle n'est jamais éliminée, reste jusqu'à la fin et est renvoyée.

La version ci-dessus était suffisante pour avoir les points. Si une preuve formelle rigoureuse avec invariant a été menée, j'ai valorisé en donnant des points supplémentaires. Voici une preuve d'invariant qui peut permettre de conclure :

Supposons qu'il existe une star, et montrons l'invariant de boucle suivant : « la star n'est pas sur le côté ».

- Initialisation : Avant la première itération, personne n'a été envoyé sur le côté. D'où l'initialisation.
- Hérédite : Supposons l'invariant vrai au début d'une itération quelconque, et montrons-le vrai à la fin de celle-ci, c'est à dire montrons qu'à la fin de cette itération la star n'est pas sur le côté.

  D'après l'invariant, la star n'était pas sur le côté au début. L'itération met une personne sur le côté, selon le principe expliqué à la question précédente. Or, d'après celle-ci, la personne en question n'est pas une star.

  Il s'ensuit qu'à la fin de l'itération la star n'a toujours pas été mise sur le côté.

  Conclusion : la star n'est pas mise sur le côté. Or, à la fin de la boucle, il reste une seule personne dans le groupe : il s'agit donc de la star.

Comme cette dernière personne est celle que l'on renvoie, on a bien prouvé que s'il existe une star `Sélection` la renvoie.

c. Il suffit de combiner la fonction de sélection et la fonction de vérification. En effet, s'il existe une star, la sélection la renvoie et la vérification confirme ; alors que s'il n'existe pas de star, la sélection renvoie quelqu'un mais la vérification contredit.

Fonction StarNaif

Entrées : G un groupe
Sorties : la star de G si elle existe, ou « pas de star » sinon

```
1 candidate ←—Sélection(G)
2 si Vérification(candidate) alors
3 |  renvoyer candidate
4 sinon
5 |  renvoyer « pas de star »
```

d. Les seules questions posées le sont dans les deux appels de fonction. On sait déjà que `Vérification` effectue $2(n-1)$ questions.

La boucle de `Sélection` itère n-1 fois (on commence avec n personnes, on en met une de côté à chaque itération, on s'arrête quand il n'en reste qu'une) et pose 1 question à chaque itération. Il n'y a pas de questions dans sa condition de boucle ou hors de sa boucle : elle effectue $n-1$ questions.

Donc :

Cette solution résout le problème en $3(n-1)$ questions.

---

4. **a.** Si l'on prouve cela, on prouve que pour tout algorithme sa complexité en questions $Q(n)$ vérifie $Q(n) \geq \left\lfloor \frac{n}{2} \right\rfloor > \frac{n}{2} - 1$. Or, l'ordre de grandeur de cette *minoration* est n, et on a donc montré que pour tout algorithme, $Q(n) = \Omega(n)$.

J'enlève une partie des points si vous ne faites pas explicitement apparaitre que $\frac{n}{2}$ est une minoration. Par contre, je tolère le fait de traiter la partie entière « avec les mains ».

---

b. Soit I_0 une instance à n personnes, ayant une star. Comme par hypothèse l'algorithme la résout en moins de $\left\lfloor \frac{n}{2} \right\rfloor$, et que chaque question n'implique que deux personnes, il existe au moins une personne p_0 qui n'a jamais fait partie d'une question : on ne sait pas si elle connait quelqu'un, ni si quelqu'un la connait.

On crée I_1 en remplaçant cette personne p_0 par une personne p_1 qui ne connait personne d'autre et que personne ne connait. Il s'ensuit que dans I_1 il n'y a pas de star.

Cependant, comme l'algorithme s'exécute sur I_0 sans jamais interroger p_0, et que I_1 diffère de I_0 uniquement par cette personne, l'algorithme s'exécutera de la même façon sur I_0 et I_1.

---

c. À la question précédente, on a montré que l'algorithme s'exécute de la même façon sur I_0 et I_1. Or, la première instance a une star tandis que la seconde n'en a pas : l'algorithme supposé totalement correct se trompe sur l'une de ces deux instances, absurde.

Donc, d'après la question 4.a. :

Tout algorithme résolvant le problème utilise $\Omega(n)$ questions.

NB : on aurait pu faire cette preuve en montrant qu'il faut interroger chaque personne (pas simplement que chaque personne soit impliquée dans une question), et donc que $Q(n) \geq n$, mais cela demande de faire un brin plus attention lors de la création de I_1. Cette version-ci a moins de pièges de rigueur.

NBB : on peut en fait montrer que tout algo effectue au moins $3n - 3 - \lfloor \log_2 n \rfloor$.

---

d. La solution trouvée en question 3. est en $O(n)$. On vient de prouver que toute solution est en $\Omega(n)$.

L'ordre de grandeur linéaire est optimal.

Solution de l'Exercice 4 – *Facteur équilibré*

1. **a.** `T[2:4[` est `[|1; 1 |]` .

---

b. Il y a trois paires `(i, j)` telles que `T[i:j[` vaille `[|1|]` :

(`i` = 1, `j` = 2) et (`i` = 2, `j` = 3) et (`i` = 3, `j` = 4)

---

2. **a.** Avant la première itération, `compteur` vaut `[|0; 0|]` et `k` vaut `1` .

Ensuite, à la fin de chaque itération (je note avec des primes les quantités en fins d'itération) :

`k'`	`compteur'`
2	`[|0; 1|]`
3	`[|0; 2|]`
4	`[|1; 2|]`

Après cette dernière itération, on quitte la boucle car on a $k \geq j$.

---

b. On propose l'invariant suivant, où *Card* désigne le cardinal (le nombre d'éléments de l'ensemble).

$\forall c \in \{0;1\}$, `compteur[c]` $= Card(\{$`tab[m]` tels que $i \leq m < k$ et `tab[m]` $= c\})$

En français, cet invariant s'écrit :

> « `compteur[0]` est le nombre de `0` dans `tab` entre les indices `i` inclus et `k` exclu. De même, `compteur[1]` est le nombre de `1` dans `tab` entre les indices `i` inclus et `k` exclu. »

La version mathématique et la version française sont toutes les deux acceptées, tant qu'elles sont rigoureuses.

c. On admet qu'il s'agit bien d'un invariant. En particulier, comme à la fin de la boucle `k = j`, il nous apprend qu'en ligne 25 `compteur[0]` est le nombre de `0` dans `tab[i:j[` et que `compteur[1]` est le nombre de `1` de ce même facteur. Le facteur est équilibré si et seulement si ces deux valeurs sont égales : c'est exactement ce que l'on renvoie.

La valeur de sortie est donc la bonne. Comme en plus il n'y a pas d'effets secondaires et qu'aucun effet secondaire n'est attendu :

La fonction est partiellement correcte.

Des points partiels voire complets peuvent être donnés même si l'invariant n'en est en fait pas un, tant que le raisonnement est rigoureux et correct.

---

3. **a.** Pour que la fonction fonctionne (haha) comme expliqué, il faut que pour chacune des valeurs de `i` on teste toutes les valeurs de `j` : on teste ainsi toutes les paires `(i, j)`.

Or, le code donné est bogué : la valeur de `j` n'est pas remise à 0 à chaque nouveau `i`. Ainsi, après la première itération de la grande boucle (lignes 45-58), `j` vaut `len + 1`... et vaut encore cette valeur au début de la seconde itération ! Ce qui fait que dans cette seconde itération on ne rentre pas dans la boucle itérieure (lignes 48-55), et qu'on ne parcourt aucun `j` lorsque `i = 1`. De même pour `i = 2`, etc.

Pour déboguer, il suffit de déplacer `int j = 0` de la ligne 43 à la ligne 47.

NB : on peut aussi faire commencer `j` à `i + 1` pour ne pas calculer plein de fois le facteur vide ; mais ce n'est pas un bug : c'est de l'optimisation.

b. Commençons par noter que le seul endroit où ont lieu des lectures du tableaux est dans l'appel de fonction (ligne 51). D'après l'indication, chaque appel effectue $O(\texttt{len})$ lectures.

Or, la boucle intérieure (lignes 48-55) effectue `len + 1` itérations (elle itère pour $\texttt{j} \in [\![0; \texttt{len}[\![$) : au total, elle effectue donc $O((\texttt{len} + 1)\texttt{len}) = O(\texttt{len}^2)$ lectures.

Cette boucle est dans le corps de la grande boucle (lignes 45-58), laquelle itérate `len` fois ($\texttt{i} \in [\![0; \texttt{len}[\![$) : elle est donc en $O(\texttt{len}.\texttt{len}^2) = O(\texttt{len}^3)$.

Comme les seules lectures sont celles-ci, on peut conclure :

`llbf_naif` (déboguée) effectue $O(\texttt{len}^3)$ lectures de cases.

NB : on peut arriver au même résultat sans utiliser la majoration $2(\texttt{j} - \texttt{i}) = O(\texttt{len})$, mais c'est plus calculatoire.

NBB : initialiser `j` à `i+1` au lieu de 0 ne change pas cet ordre de grandeur (mais divise par 2 la constante cachée par le O()).

---

c. D'après les explications précédentes, dans la version boguée la boucle intérieure est exécutée une seule fois. Comme il s'agit du seul endroit où on lieu des lectures, la complexité est celle d'une exécution de la boucle intérieure soit $O(\texttt{len}^2)$.

---

4. On ne peut renvoyer qu'une valeur. Par contre, on peut passer par pointeur les emplacements où stocker les valeurs de "sortie" supplémentaire. On ajoute ainsi deux pointeurs en arguments à la fonction. Le nouveau prototype est : `int llbf_naif(int const tab[], int len, int* i_opt, int* j_opt)` .

Il ne reste qu'à modifier la fonction pour qu'au fil de ses itérations, en plus de mémoriser la meilleure longueur elle mémorise la paire `(i, j)` associée. À la fin, avant de renvoyer la longueur, la fonction stockerait les `i` et `j` associés à cette longueur dans respectivement `*i_opt` et `*j_opt` .

NB : on a déjà utilisé cette astuce en TP, pour faire une fonction qui "renvoie" le minimum *et* le maximum d'un tableau.

---

5. **a.** Comme `T[i:j[` et `T[i:j+1[` ne diffèrent que par le fait que `T[j]` est présent dans le second facteur mais pas dans le premier, on a :

$$\boxed{\begin{pmatrix} n'_0 \\ n'_1 \end{pmatrix} = \begin{pmatrix} n_0 + \mathbb{1}_{\texttt{T[j] = 0}} \\ n_1 + \mathbb{1}_{\texttt{T[j] = 1}} \end{pmatrix}}$$

Où j'ai utilisé la fonction notation mathématique suivante (appelée *indicatrice* ou *fonction indicatrice*) :

$$\mathbb{1}_P = \begin{cases} 1 & \text{si } P \text{ est vraie} \\ 0 & \text{sinon} \end{cases}$$

---

b. L'idée est la suivante :

- On initialise `j` à `i` au lieu de 0.
- Chaque itération de la grande boucle va considérer tous les facteurs `T[i:j[` et utiliser deux variables `n_0` et `n_1` pour compter leur nombre de 0 et de 1.
- Quand on passe de `j` à `j` +1, on met à jour ces deux variables comme indiqué à la question précédente.

Ainsi, chaque itération de la sous boucle est en temps constant et non en temps linéaire (on ne recalcule pas en entier le nombre de 1 et de 0 à chaque fois, on le met simplement à jour en temps constant), et on gagne une puissance sur la complexité. On obtient le code suivant :

lbf.c

```c
/** Renvoie la longueur d'un plus grand facteur équilibré de tab. */
int llbf_quadratique(int const tab[], int len) {
  int long_max = 0;

  int i = 0;

  // Boucle parcourant toutes les valeurs de i
  while (i < len) {

    int j = i;
    int n_0 = 0;
    int n_1 = 0;
    // Boucle parcourant toutes les valeurs de j
    while (j <= len) {
      // Si tab[i:j[ est équilibré, faire long_max = max(long_max, j-i)
      if (n_0 == n_1 && j-i > long_max) {
        long_max = j-i;
      }

      // maj pour le prochain tour de boucle
      if (tab[j] == 0) { n_0 = n_0 +1;}
      else             { n_1 = n_1 +1;}
      j = j +1;
    }

    i = i +1;
  }

  return long_max;
}
```

NB : il eut été un brin plus élégant et efficace d'utiliser un tableau à deux cases au lieu de deux variables `n_0` et `n_1` (comme dans `est_equilibre`) car cela aurait éviter le `if-else` lignes 110-112, mais j'ai voulu faire deux variables comme dans la question précédentes.

---

c. Les seules lectures sont toujours celles dans la boucle itérieure. Mais cette fois-ci, chaque itération de la boucle intérieure effectue une seule lecture du tableau (ligne 110). La boucle intérieure itère sur $\texttt{j} \in [\![i; \texttt{len}[\![$ donc $\texttt{len} - i + 1$ fois. Cette boucle effectue donc $\texttt{len} - i + 1$ lectures.
Or, la boucle extérieure itère sur $\texttt{i} \in [\![0; \texttt{len}[\![$, et la complexité en lecture L_{quadra} de la fonction vérifie donc :

$$\begin{aligned} L_{quadra}(\texttt{len}) &= \sum_{\texttt{i}=0}^{\texttt{len}-1} (\texttt{len} - i + 1) \\ &= \texttt{len}^2 + \texttt{len} - \sum_{\texttt{i}=0}^{\texttt{len}-1} i \\ &= \texttt{len}^2 + \texttt{len} - \frac{(\texttt{len}-1)\texttt{len}}{2} \\ &= \frac{1}{2}\texttt{len}^2 + \frac{3}{2}\texttt{len} \\ &= \Theta(\texttt{len}^2) \end{aligned}$$

En particulier :

> La complexité de `llbf_quadratique` est un $O(\texttt{len}^2)$.

NB : on peut aussi majorer $\texttt{len} - \texttt{i}$ par $O(\texttt{len})$ pour simplifier les calculs. Cependant, ne pas faire cette simplification me permet de montrer que ce n'est pas uniquement au plus de l'ordre de $\texttt{len}^2$ mais aussi au moins de cet ordre : c'est plus précis ! (Ce n'était pas attendu.)

---

6. **a.** Le déséquilibre maximal est atteint quand toutes les cases valent 1 et ce déséquilibre vaut $+\texttt{len}$; et le déséquilibre minimal est atteint quand toutes les cases valent 0 et ce déséquilibre vaut $-\texttt{len}$.

---

b. Voici une fonction qui réalise ce qui est demandé. Pour obtenir une complexité linéaire, on calcule le déséquilibre d'un préfixe en temps constant à partir du déséquilibre du préfixe précédent (cf page suivante) :

lbf.c

```c
/** Renvoie pour chaque décalage possible l'indice
  * du premier préfixe qui réalise ce décalage.
  *
  * Entrée :
  *   - tab un tableau de 0/1, et len sa longueur
  *
  * Sortie :
  *   - un pointeur prem (obtenu via malloc) tel que
  *     pour tout d entre -len et +len inclus,
  *     prem[d+len] est l'indice terminant le premier
  *     suffixe ayant un déséquilibre d.
  *
  * Effets secondaires :
  *   - alloue avec malloc (c'est le prem renvoyé)
  *
  * Complexité : len lectures de cases
  */
int* premier_indice(int const tab[], int len) {

  // on cree la zone et on l'initialise à INT_MAX
  int* prem = (int*) malloc((2*len+1)*sizeof(int));
  int indice = 0;
  while (indice < 2*len+1) {
    prem[indice] = INT_MAX;
    indice = indice +1;
  }

  // On parcourt les préfixes de tab de gauche
  // à droite. Le déséquilibre d'un préfixe se
  // calcule à partir du déséquilibre du précédent
  int j = 0;
  int d = 0; // desequilibre de tab[0:indice[
  while (j < len) {

    // si c'est la première fois qu'on voit
    // ce déséquilibre, le stocker
    if (prem[d+len] == INT_MAX) {
      prem[d+len] = j;
    }

    // lire tab[j] et maj le déséquilibre en csq
    if (tab[j] == 1) {d = d +1;}
    else             {d = d -1;}
    j = j +1;
  }
  // /!\ Il faut traiter le dernier déséquilibre calculé
  if (prem[d+len] == INT_MAX) {
    prem[d+len] = j;
  }

  return prem;
}
```

Avant de passer à la question suivante, calculons tout de suite sa complexité (en anticipation de la question 6.f.) : il y a une seule ligne où une case de `tab` est lue, la ligne 163. Celle-ci est comprise dans une boucle qui itère `len` fois. Donc cette fonction effectue exactement `len` lectures.

---

c. Notons N_0 la fonction qui a un facteur associe son nombre de 0, et N_1 le nombre de 1. Ces fonctions vérifient une relation de Chasles :

$$N_0(T[i:j[) + N_0(T[j:k[) = N_0(T[i:k[)$$

Et idem pour N_1.

Le déséquilibre d'un facteur `T[i:j[` est par définition N_1(`T[i:j[`) − N_0(`T[i:j[`).
Avec les notations de l'énoncé, comme `d` est le déséquilibre du préfixe en cours, on a :

$$N_1(\texttt{tab[0:j[}) - N_0(\texttt{T[0:j[}) = \texttt{d}$$

Et donc :

$$\begin{aligned}
&N_1(\texttt{tab[prem[d]:j[}) - N_0(\texttt{T[prem[d]:j[}) \\
&= (N_1(\texttt{tab[0:j[}) - N_1(\texttt{tab[0:prem[d][})) - (N_0(\texttt{T[0:j[}) - N_0(\texttt{T[0:prem[d][})) \\
&= (N_1(\texttt{tab[0:j[}) - N_0(\texttt{T[0:j[})) - (N_1(\texttt{tab[0:prem[d][}) - N_0(\texttt{T[0:prem[d][})) \\
&= d - d \\
&= 0
\end{aligned}$$

Donc :

Le facteur `tab[prem[d]:j[` est équilibré.

---

d. Raisonnons par l'absurde et supposons qu'il existe $i <$ `prem[d]` tel que `T[i:j[` soit équilibré.

Alors avec les mêmes calculs qu'à la question précédente, on prouve que `T[0:j[` et `T[0:i[` ont le même déséquilibre.

Or, le déséquilibre de `T[0:j[` est `d`, et le premier préfixe l'ayant ce termine en `prem[d]` > `j`. Absurde.

---

e. En appliquant les raisonnements des questions précédentes, on obtient le code ci-dessous (page suivante) :

lbf.c

```c
/** Renvoie la longueur d'un plus long facteur équilibre.
  *
  * Complexité linéaire.
  */
int llbf(int const tab[], int len) {
  int* prem = premier_indice(tab, len);

  int long_max = 0;
  int j = 0;
  int d = 0;
  while (j < len) {

    // T[i:j[ est le plus grand facteur équilibré
    // qui se termien à l'indice j
    int i = prem[len+d];
    if (i != INT_MAX && j-i > long_max ) {
      long_max = j-i;
    }

    if (tab[j] == 1) {d = d +1;}
    else             {d = d -1;}
    j = j +1;
  }
  // /!\ il faut traiter le dernier facteur
  int i = prem[len+d];
  if (i != INT_MAX && j-i > long_max ) {
    long_max = j-i;
  }

  free(prem);
  return long_max;
}
```

---

f. On a déjà calculé la complexité de `premier_indice` : exactement `len` lectures.

Le même raisonnement prouve que la boucle de `llbf` effectue aussi `len` lectures.

Comme enfin les seules lectures sont celles effectuées par la boucle et celles effectuées par l'unique appel à `premier_indice` (hors de la boucle, ligne 181), on conclut :

> `llbf` effectue 2`len` lectures de cases du tableau `tab`.

---

g. Raisonnons par l'absurde et supposons qu'il existe une fonction totalement correcte $\mathscr{A}$ qui ne lit pas toutes les cases de son entrée pour résoudre le problème. Soit `tab` un tableau équilibré, et notons `len` sa longueur. Alors par hytpohèse, il existe une case de `tab` qui n'est plus lue.

On crée `autre_tab` en changeant le contenu de cette case pas lue. Comme on a changé une seule case depuis un tableau équilibré, `autre_tab` n'est pas équilibré. C'est bien un tableau de 0 et de 1, et on peut donc bien lui appliquer $\mathscr{A}$.

Cependant, comme $\mathscr{A}$ s'exécute sur `tab` sans jamais lire la case en question, il s'exécutera de la même façon sur `tab` et `autre_tab` (qui ne diffère que par la case non-lue) et renverra la même réponse. Or, dans le premier la réponse attendue est `len` (le plus grand facteur équilibré est le tableau tout entier), alors que dans l'autre il n'existe aucun facteur équilibré de longueur `len`. $\mathscr{A}$ se trompe donc sur l'une de ces deux entrées : absurde puisqu'elle est totalement correcte.

On conclut :

> Toute fonction résolvant le problème utilise doit au moins lire chaque case de son entrée.