DEVOIR SURVEILLÉ II

Informatique

Durée : 2 heures

Consignes :

- La candidate attachera la plus grande importance à la **clarté**, à la **précision** et à la **concision** de la rédaction.
  2 points de la note finale sur 20 sont reservés au soin de la copie.[1]
- Si une candidate est amenée à repérer ce qui peut lui sembler être une erreur d'énoncé, elle le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'elle a été amenée à prendre.
- **L'usage du cours, de notes, d'une calculatrice ou de tout autre appareil électronique est strictement interdit.**
- L'usage d'éléments du langage C non-encore vus en cours/TP/TD est interdit. Cela concerne principalement les boucles `for`, l'opérateur `++`, ainsi que les `struct`.
  L'utilisation de `else if` est autorisé.

Pour une bonne présentation de la copie, il est notamment demandé à la candidate de :

- Mettre en valeurs les résultats finaux.
- Tirer un trait horizontal entre chaque question afin que le début et la fin d'une question soient facilement identifiables.
- Indenter[2] ses codes.
- Si elle écrit un programme non-trivial, le préfixer d'un résumé de son fonctionnement en deux ou trois lignes. *Je m'autorise à mettre 0 à un code compliqué non expliqué, quand bien même il serait correct.*
- Indiquer au début de sa copie avoir lu ces consignes.
- Prendre l'initiative de couper le code demandé en plusieurs fonctions si cela aide à la lecture.
- Numéroter ses copies en bas à droite sous le format numéro_de_la_page / nombre_total_de_pages.
- Ne pas utiliser de correcteur blanc.[3]

Dans tout le devoir, on supposera que les librairies C suivantes sont déjà incluses : `stdio.h` , `stdlib.h` , `stdbool.h` , `stdint.h` , `limits.h` . Quelques erreurs de syntaxe ponctuelles pourront être tolérées.

Les exercices sont indépendants.

Ne retournez pas la page avant d'y être invitées.

1. Ce n'est pas une lubie personnelle, cela se fait aux concours. Par exemple, CCINP maths réserve 1 point sur 20.
2. Le terme « indentation » désigne le décalage horizontal.
3. Car cela ne passe pas aux scanners des concours, et est en conséquence interdit aux concours.

Compléments de maths

Une suite est dite *arithmético-géométrique* s'il existe a et b tels que pour tous $n > 0$, $u_{n+1} = au_n + b$. Pour calculer la valeur des termes d'une telle suite :

- On commencer par calculer le point fixe fixe de l'équation de récurrence, c'est à dire par résoudre l'équation suivante (d'inconnue l) : $l = al + b$.
- On pose la suite (appelée suite auxiliaire) $v_n = u_n - l$. On prouve qu'elle est géométrique, et on calcule la valeur de ses termes.
- On en déduit la valeur des termes de $u_n = v_n + l$.

I - Proche du cours

Exercice 1 – *Here we go again*

On considère la fonction ci-dessous :

somme-entiers.c

```c
/** Renvoie la somme des entiers de 1 à n (inclus).
  * Pas d'effets secondaires.
  */
int somme_entiers(int n) {
  int somme = 0;
  int i = 1;

  while (i <= n) {
    somme = somme + i;
    i = i +1;
  }

  return somme;
}
```

1. **a.** Donnez le prototype de la fonction.
   b. Spécifiez cette fonction.
2. Prouver que cette fonction termine.
3. Prouver qu'elle est partiellement correcte (on ignore les éventuels débordements).
4. Prouver qu'elle est totalement correcte.
5. Calculer la complexité en nombre d'additions de cette fonction.
6. On rappelle que le type `int` est un type d'entiers signés. On le suppose de taille 4 octets (32 bits). Donner une valeur approximative du premier entier `n` positif pour lequel la fonction ne renvoie pas la bonne réponse. *On pourra passer rapidement sur les détails des calculs.*

Exercice 2 – *Palindrome*

En C, pour représenter un texte, on utilise le type `char*` : un texte est en fait un pointeur vers une zone de la mémoire qui contient les lettres du texte, les unes après les autres. Pour marquer la fin du texte, on ajoute à la fin le caractère spécial `'\0'`.
Ainsi, la ligne de code C ci-dessous :

palindrome.c

```c
char* txt = "Camille !";
```

Correspond à :

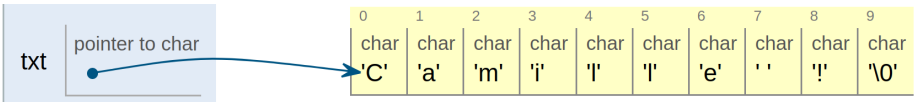

Figure II.1 – Visualisation de la mémoire du code précédent, proposée par `Pythontutor`. À gauche se trouve la pile mémoire, à droite la zone de stockage des données statiques.

On voit bien sur cette visualisation que les caractères sont stockés les uns après les autres, jusqu'à atteindre le `'\0'` qui marque la fin.

1. Écrire une fonction `unsigned int length(char* str)` qui prend en argument un pointeur comme décrit ci-dessus et renvoie le nombre de caractères du texte (sans compter `'\0'`).
   On pourra faire une boucle `while` *qui parcourt les cases jusqu'à tomber sur le caractère de fin.*

On dit qu'un texte est un **palindrome** s'il est symétrique par rapport à son milieu, c'est à dire si sa première lettre est égale à sa dernière, et que sa seconde lettre est égale à son avant-dernière, etc[i]. Ainsi, « `kayak` » est un palindrome, « `abc ff cba` » est aussi un palindrome, mais « `tant` » n'est pas un palindrome et « `kayek` » non plus.

2. Écrire une fonction `bool est_palindrome(char* str)` qui teste si un texte est un palindrome.
3. On propose de remplacer `char*` par `char const*` dans le prototype. Qu'est-ce que cela changerait ?

i. Cela revient aussi à dire que le texte se lit pareil de gauche à droite et de droite à gauche.

II - Problème

Exercice 3 – *Trouver la star*

Les élèves ayant un tiers temps passent cet exercice, en écrivant « Exercice Star : TT » sur leur copie
On considère un groupe de n personnes. Ces personnes peuvent ou non se connaitre mutuellement. Ce n'est pas forcément symétrique : x peut connaitre y alors qu'y ne connait pas x. Chaque personne se connait elle-même.

Définition 1 (Star).
Une **star** est quelqu'un que le monde connait mais qui ne connait personne (à part elle-même).

L'objectif est de trouver une star s'il en existe une, ou d'indiquer qu'il n'y en a pas sinon.
Pour apprendre si une personne x connait une personne y, on demande à x « connais-tu y? ». C'est la seule et unique question que l'on a le droit de poser.
La complexité sera comptée en nombre de questions. On travaillera en pseudo-code (et on peut inventer des opérations comme « demander à telle personne de se mettre sur le côté »).

1. En raisonnant par l'absurde, prouver qu'il y a au plus une star.
2. **a.** Proposer une fonction `Vérification` qui prend en entrée un groupe et une personne de ce groupe, et qui teste si la personne est une star.
   b. En déduire un algorithme trouvant une star en $\Theta(n^2)$ questions.
   c. Justifier brièvement de sa complexité.
3. **a.** On pose à une personne x la question « connais-tu y? » (avec x et y deux personnes distinctes). Justifier qu'on peut déduire de sa réponse que soit x soit y n'est pas une star.
   b. Prouver que s'il existe une star, la fonction `Sélection` ci-dessous la renvoie.

Fonction Sélection

Entrées : `G` un groupe de personne
Sorties : un personne du groupe dite « candidate »

```
1  candidate ←une personne du groupe
2  tant que G n'est pas réduit à 1 personne faire
3  |  y ←une personne de G
4  |  si candidate connait y alors
5  |  |  Demander à la personne candidate de quitter G (et se mettre sur le côté)
6  |  |  candidate ←y
7  |  sinon
8  |  |  Demander à la personne y de se mettre sur le côté
9  (Demander à toutes les personnes de revenir dans le groupe)
10 renvoyer candidate
```

c. En déduire un algorithme qui renvoie une star s'il en existe une ou qui répond qu'il n'existe pas de star sinon, en temps linéaire

d. Prouver sa complexité.

4. On veut prouver que toute solution au problème utilise $\Omega(n)$ questions.

a. Justifiez que si l'on prouve que toute solution utilise au moins $\left\lfloor \frac{n}{2} \right\rfloor$ questions, on aura prouvé le résultat attendu.

b. On raisonne par l'absurde et on suppose qu'il existe un algorithme totalement correct résolvant le problème en utilisant strictement moins de $\left\lfloor \frac{n}{2} \right\rfloor$ questions. On note I_0 une instance du problème dans laquelle se trouve une star.

Expliquez comment modifier I_0 en I_1 de sorte à ce que « l'algorithme ne voit pas la différence mais se trompe sur I_1 ».

c. Conclure.

d. En déduire si l'ordre de grandeur de complexité de la solution trouvée en question 3. est satisfaisant.

Exercice 4 – *Facteur équilibré*

Définition 2 (Facteur, préfixe, suffixe).
Soient `T` est un tableau de longueur ℓ, $\texttt{i} \in [\![0;\ell[\![$ et $\texttt{j} \in [\![0;\ell]\!]$.
On note `T[i:j[` le tableau `[| T[i]; T[i+1]; ...; T[j-1] |]`,c'est à dire `T` des indices `i` inclus à `j` exclu. On appelle `T[i:j[` un **facteur** de `T`.
De plus :

- Si $\texttt{i} = 0$, on dit que le facteur est un **préfixe**.
- Si $\texttt{j} = \ell$, on dit que le facteur est un **suffixe**.
- Si $\texttt{i} \geq \texttt{j}$, on dit que le facteur est **vide** (et il ne contient rien).

Exemple. Dans le tableau `T` valant `[| 0; 1; 1; 1; 0; 0 |]` (de longueur $\ell = 6$) :

- le facteur `T[0:0[` est vide (et est un préfixe).
- le facteur `T[0:3[` est `[|0; 1; 1|]` (et est un préfixe).
- le facteur `T[3:6[` est `[|1; 0; 0|]` (et est un suffixe).
- le facteur `T[1:5[` est `[|1; 1; 1; 0 |]` (et n'est ni un préfixe, ni un suffixe).
- le facteur `T[0:6[` est le tableau en entier (et est à la fois un préfixe et un suffixe).

⚠ Cette notion de facteur *n'est pas* implémentée dans le langage C, et vous ne pouvez donc pas écrire `T[i:j[` en C.

1. **a.** Dans le tableau `T` précédent, quel est le facteur `T[2:4[` ?

b. Donner toutes les paires `(i, j)` telles que `T[i:j[` soit `[|1|]` .

Dans tout cet exercice, on ne considère que des tableaux contenant uniquement des 0 et des 1. On comptera la complexité en nombre de lectures de cases du tableau pris en argument.

Définition 3 (Facteur équilibré).
Un facteur est dit **équilibré** s'il contient autant de 0 que de 1. En particulier, le facteur vide est toujours équilibré.

On s'intéresse au problème Length-Longuest-Balanced-Factor (abrégé LLBF) suivant :

- Entrée : T un tableau de 0 et de 1
- Tâche : calculer la longueur d'un plus long facteur équilibré.

2. On considère tout d'abord la fonction `est_equilibre` ci-dessous :

lbf.c

```c
/** Renvoie true ssi tab[i:j[ est équilibré.
  *
  * Entrées :
  *   - tab un tableau de 0 et de 1
  *   - i et j définissant un facteur de tab
  *
  * Sortie :
  *   - true si tab[i:j[ est équilibré, false sinon
  *   - (aucun effets secondaires)
  *
  * Complexité (en nombre de lectures de cases du tableau) : 2(j-i).
  */
bool est_equilibre(int const tab[], int i, int j) {
  int compteur[2] = {0,0};
  int k = i;
  while (k < j) {
    compteur[tab[k]] = compteur[tab[k]] +1;
    k = k +1;
  }
  return compteur[0] == compteur[1];
}
```

   a. Exemple. On exécute cette fonction sur le tableau `[|0; 1; 1; 0; 1; 0; 1|]`, avec `i` = 1 et `j` = 4. Représentez les valeurs de `compteur` et `k` juste avant la première itération ainsi qu'à la fin de chaque itération. Indiquer la sortie de la fonction : est-elle correcte ?
   b. Proposez un invariant de boucle permettant de prouver qu'à la fin de la boucle, `compteur[0]` contient le nombre de 0 du facteur `tab[i:j[` et `compteur[1]` le nombre de 1 de ce facteur.
   c. En admettant qu'il s'agit bien d'un invariant, en déduire la correction partielle de la fonction.

On admet que `est_equilibre` est totalement correcte et a bel et bien la complexité indiquée.

3. La fonction `llbf_naif` ci-dessous est sensée résoudre le problème.
   Elle procède ainsi : la première boucle (lignes 45-58) parcourt toutes les valeurs de i possibles. Pour chacune d'entre elles, la seconde boucle (lignes 48-55) parcourt toutes les valeurs de j possibles : on parcourt ainsi toutes les paires (i, j) possibles. Enfin, pour chacune de ces paires, le corps de la seconde boucle teste si le facteur correspondant est équilibré.

lbf.c

```c
/** Renvoie la longueur d'un plus long facteur équilibré de tab.
  *
  * Entrées :
  *   - tab un tableau de 0 et de 1
  *   - len sa longueur
  *
  * Sortie :
  *   - la longueur maximale d'un facteur équilibré de tab
  *   - (aucun effets secondaires)
  */
int llbf_naif(int const tab[], int len) {
  int long_max = 0;

  int i = 0;
  int j = 0;

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

    // 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 (est_equilibre(tab, i, j) && j-i > long_max) {
        long_max = j-i;
      }
      j = j +1;
    }

    i = i +1;
  }

  return long_max;
}
```

a. Cette fonction est buguée. Trouvez son ou ses bugs et indiquez comment les résoudre (on mentionnera les numéros des lignes impliquées dans les modifications afin d'éviter tout ambguïté).

b. Prouvez que la version corrigée est en $O(\texttt{len}^3)$. On pourra utiliser le fait que `j-i` $= O($`len`$)$ afin de simplifier les calculs.

c. Quelle était la complexité de la version buguée ? *On attend une explication mais pas nécessairement une preuve rigoureuse.*

4. Si l'on voulait renvoyer non pas uniquement la longueur d'un plus grand facteur, mais aussi des indices `i` et `j` associés, comment pourrait-on faire ?
On rappelle qu'il n'existe pas de type de paire/triplet en C, et que l'usage de `struct` *est interdit.*

5. On veut accélérer la solution trouvée en question 3.. Pour cela, on va éviter les appels à `est_equilibre` .

a. Soit `T[i:j[` un facteur avec $0 \leq$ `i` $\leq$ `j` $<$ `len`. On note n_0 le nombre de 0 dans ce facteur, et n_1 le nombre de 1 dans ce facteur.
Exprimer n'_0 et n'_1, qui sont respectivement le nombre de 0 et de 1 du facteur `T[i:j+1[` , en fonction de n_0 et n_1.

b. En déduire une fonction `int llbf_quadratique(int const tab[], int len)` qui teste si un facteur est équilibré en temps quadratique.

c. Prouver sa complexité.

6. On veut accélérer encore plus. Pour cela, on utilise une astuce courante en informatique : celle des « sommes partielles ». L'idée est la suivante : on a besoin d'une information sur les *facteurs* `T[i:j[` , et on cherche à exprimer cette information à partir d'une information sur les *préfixes*.
Ici, l'information qui va nous intéresser est le déséquilibre :

Définition 4 (Déséquilibre).
Soit `T[i:j[` un facteur. Le déséquilibre de ce facteur est son nombre de 1 moins son nombre de 0. Notez qu'un facteur est équilibré si et seulement si son déséquilibre est nul.

Exemple. Dans le tableau `T` valant `[| 0; 1; 1; 1; 0; 0 |]` (de longueur $\ell = 6$, il s'agit de l'exemple initial) :

- le déséquilibre de `T[0:0[` est 0. Ce facteur est donc équilibré.
- le déséquilibre de `T[0:3[` est $+1$.
- le déséquilibre de `T[3:6[` est -1.
- le déséquilibre de `T[1:5[` est est $+2$.
- le déséquilibre de `T[0:6[` est 0. Ce facteur est donc équilibré.

a. Quel est le déséquilibre possible maximal et minimal pour un facteur d'un tableau `tab` de longueur `len` ? *On n'attend pas de justification.*

b. On va pré-calculer pour chaque valeur possible de déséquilibre dans le tableau `tab` l'indice du premier préfixe ayant ce déséquilibre.

Écrire une fonction `int* premier_indice(int const tab[], int len)` qui effectue cela. Elle doit renvoyer un pointeur `prem` qui pointe vers une zone mémoire (créée avec `malloc`) telle que pour tout `d` $\in [\![-$`len`$;+$`len`$]\!]$, `prem[len+d]` est le premier indice `j` tel que le préfixe `tab[0:j[` ait un déséquilibre valant `d`. Elle doit avoir une complexité linéaire (il est possible de l'écrire en lisant une et une seule fois chaque case du tableau).

Si jamais une valeur de déséquilibre ne correspond à aucun préfixe, on stockera `INT_MAX` dans la case correspondante. Il s'agit d'une valeur pré-codée en C[i] valant le plus grand entier signé représentable, et on peut raisonner comme si elle valait $+\infty$.

Des points partiels pourront être accordés pour une fonction en complexité quadratique.

c. Prouver que si le préfixe `tab[0:j[` a un déséquilibre `d` alors, avec les notations de la question précédente, le facteur `tab[ prem[len+d] : j [` est équilibré.

⚠ Dans les TD, j'avais laissé passé une erreur de signe grave. Il s'agit bien de `prem[len+d]` (avec un `+`), c'est à dire de la première fois où le décalage `d` a été vu (et non `-d`, qui serait faux).

d. Prouver qu'il s'agit du plus grand facteur équilibré se terminant à l'indice `j`.

e. En déduire une fonction C `int llbf(int const tab[], int len)` qui résout le problème LLBF en temps linéaire.

On n'oubliera pas de libérer la mémoire allouée.

f. Prouver la complexité de cette fonction, en indiquant le nombre exact de lectures effctuées.

g. Prouver que tout algorithme résolvant le problème LLBF doit au moins lire chaque case du tableau pris en entrée.

Pour prouver cela, on peut procéder ainsi :

- Raisonner par l'absurde, et supposer qu'il existe une fonction totalement correcte qui résout le problème en ne lisant pas toutes les cases.
- Introduire un tableau `tab` de longueur `len` dont le plus grand facteur équilibré est de longueur `len` (çad que le tableau est équilibré).

  Le modifier en un tableau `autre_tab` de même longueur que `tab` mais qui n'est pas équilibré, et tel que la fonction s'exécute de la même façon sur `tab` et sur `autre_tab`.
- Conclure.

On conclut ainsi que, si la solution de la question 6. a été écrité précautionneusement, l'ordre de grandeur de son nombre de lectures est optimal[ii]

i. En réalité, elle est stockée dans `limits.h`, que l'on suppose inclus.

ii. On peut même l'écrire en exactement `len` lectures, en fusionnant `premier_indice` et `llbf`. Mais j'ai préféré découper pour simplifier.