DEVOIR SURVEILLÉ I

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 inclut notamment les boucles `for`, l'opérateur `++`, l'allocation dynamique et la manipulation explicite de pointeurs.
  L'utilisation de `else if` est autorisé. La récurrence est autorisée[2].

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[3] 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.*
- 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.[4]

Dans tout le devoir, on supposera que les librairies C suivantes sont déjà incluses : `stdio.h` , `stdlib.h` , `stdbool.h` , `stdint.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. Mais je doute qu'elle vous aide.
3. Le terme « indentation » désigne le décalage horizontal.
4. Car cela ne passe pas aux scanners des concours, et est en conséquence interdit aux concours.

Rappels de cours

Un tableau est une « collections de variables » lesquelles sont nommées des « cases ». On accède à la case d'indice i du tableau T via $T[i]$. On peut la modifier et/ou l'utiliser comme une variable normale. Un tableau à L cases est indicé de 0 à L-1.

Les caractères `char` (aka ce que l'on obtient en appuyant une fois sur une touche du clavier) sont encodés des entiers. On peut faire des opérations arithmétiques avec (par exemple, `'a' + 1` donne `'b'`) ou des comparaisons (on peut tester `variable <= 'z'` , ou encore `variable == 'z'`).
On peut donc écrire du code comme `char var_bis = 'm' + var_de_type_char;`

Afficher (`printf`) et/ou demander une valeur à l'utilisateur (`scanf`) sont des effets secondaires. Sauf demande explicite, il ne faut jamais faire d'affichage et/ou de demande de valeur.

I - Proche du cours

Exercice 1 – *Autour de la recherche dichotomique*

On considère dans un premier temps la fonction ci-dessous :

dichotomie.c

```c
/** Recherche de x dans T
 *
 * Entrées : - x un entier
 *           - tab un tableau d'entiers triés (ordre croiss.),
 *           - len la longueur de tab
 *
 * Sortie  : - un indice auquel se trouve x dans tab,
 *             ou -1 s'il n'en existe pas.
 */
int recherche_dichotomique(int x, int tab[], int len) {
  int deb = 0;
  int fin = len;

  // Au début de chaque itération, si x est présent
  // dans le tableau alors c'est entre
  // les indices deb inclus et fin exclu
  while (deb < fin) {
    int milieu = (deb+fin) / 2;
    if (tab[milieu] == x) { return milieu; }
    else {
      if (x < tab[milieu]) {
        fin = milieu;
      } else { // milieu < x
        deb = milieu +1;
      }
    }
  }

  return -1;
}
```

1. Rappeler le sens de l'opérateur `/` appliqué à des entiers (ligne 21).
2. Prouver la terminaison de la fonction.
3. Justifier très brièvement du fait que si la fonction ne renvoie pas -1, alors elle a bien renvoyé un indice du tableau auquel se trouve l'élément cherché.

On appelle la fonction précédente grâce au code ci-dessous :

dichotomie.c

```c
int main(void) {

  int x = 5;
  int T[] = {-3, 0, 2, 15};
  int indice = recherche_dichotomique(x, T, 4);
  if (indice != -1) {
    printf("L'élément %d a été trouvé à l'indice %d du tableau.\n", x, indice);
  } else {
    printf("L'élément %d n'a pas été trouvé dans le tableau.\n", x);
  }

  return EXIT_SUCCESS;
}
```

4. Faire un schéma de la pile mémoire (on pensera à indiquer le sens des adresses croissantes) lorsque :
   a. L'exécution atteint le début de la ligne 37 pour la première fois.
   b. L'exécution atteint le début de la ligne 22 pour la première fois.
   c. L'exécution atteint le début de la ligne 46 pour la première fois.
5. Notons L le nombre d'itérations de la boucle[i]. Prouver $L \leq \lceil \log_2(\texttt{fin} - \texttt{deb}) \rceil + 1$ fois. (On rappelle que la fonction $\log_2$ est la réciproque sur $\mathbb{R}_+^*$ de $x \mapsto 2^x$.)

   On pourra commencer par montrer le lemme suivant : $\texttt{fin}' - \texttt{deb}' \leq \frac{\texttt{fin}-\texttt{deb}}{2}$

i. Précisons que lorque l'on sort de la boucle, on n'entame pas de nouvelle itération. Par exemple, si `len=1`, on peut prouver qu'il y aura au plus $L = 1$ itération (et non $L = 2$).

Exercice 2 – *Encore ?*

On considère la fonction ci-dessous :

somme-entiers.c

```c
int somme_entiers(int n) {
  int somme = 0;
  int i = 1;

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

  return somme;
}
```

1. Donnez le prototype de la fonction.
2. a. Que renvoie cette sortie sur l'entrée `5` ? Sur l'entrée `-2` ?
   b. Spécifiez cette fonction. On n'omettra pas les effets secondaires.
3. Prouver que cette fonction termine.
4. Représenter le graphe de flot de controle de la fonction.
5. 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.

II - Problème

Le but de ce problème est de vous habituer à lire un énoncé qui contient beaucoup (trop) de mise en contexte, et au fait que la difficulté des questions n'est pas nécessairement croissante.

Exercice 3 – *Codes correcteurs*

Lorsque l'on envoie des messages via un réseau (comme internet), le message est converti en binaire puis transféré le long de fils électriques / ondes radios / fibres optiques. Ces canaux ne sont pas parfaits : du fait de la triste réalité de la physique dans le vrai monde, il peut y avoir des erreurs.
Par exemple, si l'on envoie l'octet `0110 0001` (qui représente `'a'` en ASCII), il peut y avoir une erreur lors du transfert qui fait que l'on obtient `0110 1001` à l'arrivée (qui représente `'i'`). On dit que le bit de poids 3 a été *flipé* pour désigner le fait que sa valeur a changé.

1. On suppose que lors d'une transmission, chaque bit a une probabilité 10^{-4} d'être flippé[i]. Le fait qu'un bit flip ou non n'influe pas sur la probabilité que les autres flippent ou non. On transfert un message de 5 kilooctets.[ii]

Donner l'expression de la probabilité qu'il arrive intact à destination.
Si vous aviez accès à la calculatrice, elle vous indiquerait que cette probabilité vaut environ 2%.

Dans la suite, on nomme « Alice » la personne qui veut envoyer un message et « Bob » celle qui veut le recevoir.

A. Sommes de contrôle

Pour vérifier que le message a été transmis de Alice à Bob correctement, une méthode classique est celle de la *somme de contrôle* (*checksum* en anglais). On suppose que Alice et Bob ont tous deux accès à une fonction `check` qui à un message associe un entier (de k bits ou moins, avec k pas trop gros). Cette fonction est souvent définie à l'aide d'une somme, d'où le nom de la méthode.
On procède alors ainsi :

1) Alice envoie son message m. On nomme m' le message tel que Bob le reçoit. Elle envoie ensuite $check(m)$.
2) Bob calcule $check(m')$ et le compare au $check(m)$ reçu. S'ils sont égaux, il répond "OK" à Alice et est content car il est probable que $m = m'$. Sinon, il répond "PAS OK".
3) Si Alice ne reçoit pas "OK", elle recommence en étape 1).

Cette façon de faire est la base (certes simplifiée) du protocole TLS, utilisé partout sur internet.[iii]

2. Que se passe-t-il si lors du transfert d'Alice à Bob, il n'y a pas d'erreur sur le transfert de m mais une erreur sur celui de $check(m)$? Est-ce probable ?

Remarque. Des mathématiques avancées sont impliquées dans la conception d'une bonne fonction *check* : on veut qu'elle soit "à peu près injective" pour qu'on soit confiant en le fait que $check(m') = check(m) \implies m = m'$. Elle ne peut cependant pas être injective car il y a beaucoup moins de valeurs possibles pour $check(m)$ (2^k valeurs possibles) que pour m (infinité de valeurs possibles puisque le message peut-être aussi long que l'on veut). On ne s'intéressera pas à ces mathématiques ici[iv].

On propose le fonctionnement suivant pour *check*, avec $k = 8$:

a) D'abord calculer la somme de toutes les caractères du message (interpretés comme des entiers).
b) Ensuite renvoyer cette somme modulo 2^8.

3. Sachant que `'m'` correspond à 109, `'p'` à 112, `'2'` à 50 et `'i'` à 105, calculer l'image par *check* du message `"mp2i"`.

4. Le type `char` sert à encoder des caractères ACSII en les encodant comme les entiers auxquels ils correspondent. Il utilise 1 octet. Sachant qu'il y a 127 caractères ASCII et qu'ils correspondent aux entiers $[\![0; 128[\![$, est-ce que leur encodage peut varier en fonction de si `char` encore des entiers de manière signé ou non-signé ?

On suppose que le type `char` est implémenté comme des entiers non-signés sur 8 bits.[v]
En C, on utilisera des tableaux de `char` pour manipuler les messages. Sauf mention contraire, un argument de fonction nommé `len` correspond toujours à la longueur d'un tableau associé.

5. Écrire une fonction `char check(char message[], int len)` qui calcule *check* comme indiqué précédemment.
NB : L'entier calculé faisant moins de 8 bits, on peut bien utiliser le type `char` (qui est un type d'entiers 8 bits non-signés) pour le stocker.

6. *Cette question et les 2 suivantes sont indépendantes du contexte de ce problème.*
   Écrire une fonction `char premiere_lettre(char message[], int len)` qui renvoie la première lettre du message.
7. Écrire une fonction `int nombre_point(char message[], int len)` qui calcule le nombre de fois où le caractère `'.'` apparait dans le message.
   On pourra utiliser `message[indice] == '.'` *pour tester si le caractère d'indice* `indice` *est un point.*
8. Écrire une fonction `int nombre_lol(char message[], int len)` qui calcule le nombre de fois où les 3 caractères `'l'`, `'o'`, `'l'` apparaissent les uns à côté des autres et dans cet ordre dans le message.
   Ainsi, si le message est "xptdrlololer", la valeur à renvoyer est 2.

B. Code à répétition

La méthode par somme de contrôle est très pratique, mais elle a un inconvénient : en cas d'erreur, Alice doit renvoyer tout le message à Bob. En effet, Bob peut détecter des erreurs mais pas les corriger. On voudrait changer cela et faire en sorte que tant qu'il n'y a pas trop d'erreurs, Bob soit capable de les corriger lui-même.
Une première idée pour cela est de tripler chacun des bits envoyés. Pour simplifier, on va plutôt tripler chacun des caractères envoyés. Ainsi, au lieu d'envoyer le message `"mp2i"`, Alice enverrait `"mmmppp222iii"`. On dit qu'on a *encodé* `"mp2i"` en `"mmmppp222iii"`.
Tant qu'au plus une seule des trois copies de chaque caractère subit des flips, il est possible de décoder. Si l'on reçoit par exemple le triplet `"2aa"`, on suppose que le `2` est une erreur et que le bon triplet est `"aaa"` qui correspond au message `"a"`. On dit que l'on a *décodé* `"2aa"` en `"a"`.

9. Supposons que Bob ait reçu le message `"LLLiyyinnnnx CBCOoonnnwwwa.ayy/"`. Quel était le message d'origine le plus probable ?

On appelera `msgAlice` le message d'origine d'Alice, `msg_triple` sa version détriplée, `msg_recu` ce que reçoit Bob, et `msgBob` ce que Bob en déchiffre en corrigeant les erreurs.

10. Écrire une fonction `void encode(char msg_triple[], char msgAlice[], int len)` qui prend en argument un message `msgAlice` de longueur `len` et doit modifier `msg_triple` de sorte à ce qu'il contienne une version détriplée du message. On suppose que `msg_triple` est bien de longueur au moins $3 * \texttt{len}$.
    Par exemple, si lors de l'appel de la fonction `msg` est le tableau `[|m; p; 2; i|]` et `len` = 4, après l'appel de la fonction `msg_triple` doit contenir `[|m; m; m; p; p; p; 2; 2; 2; i; i; i|]`
11. Écrire une fonction `bool decode(char msgBob[], char msg_recu[], int len)` qui prend en argument un message reçu `msg_recu` de longueur `len` et le décode en stockant le message décodé dans `msgBob`.
    Si jamais en décodant il y a un triplet que l'on ne parvient pas à décoder, la fonction renverra immédiatement sans essayer de décoder le reste. Si elle a réussi à tout décoder, elle renverra `true` à la fin.
12. Proposer une modification simple de cette façon de faire afin de diminuer la probabilité que le décodage échoue.

B. Code de Hamming (7,4)

Cette partie sera probablement sur très peu de points, voire aucun. Ne la faites que si vous avez traité le reste.
La méthode précédente était un peu couteuse en taille. On voudrait faire plus compact. On va travailler bit à bit. On se permet d'utiliser l'opération XOR sur les bits (et non sur des booléens), et on la notera $\oplus$:

x \ y	0	1
0	0	1
1	1	0

Valeur de $x \oplus y$.

On admet que l'ordre des variables et le parenthésage dans une suite de $\oplus$ n'a pas d'importance (comme dans une suite de $+$ ou une suite de $\times$ avec des entiers positifs).
On découpe le message à envoyer en paquets de 4 bits. Chaque paquet est encodé ainsi :

a) D'abord les 4 bits b_0, b_1, b_2, b_3 à envoyer.

b) Ensuite le bit $p_0 = b_0 \oplus b_1 \oplus b_3$, puis le bit $p_1 = b_1 \oplus b_2 \oplus b_3$, puis enfin le bit $p_2 = b_0 \oplus b_2 \oplus b_3$.

On appelle cet encodage le « code de Hamming (7,4) » (le 7 correspond au nombre total de bits, et le 4 au nombre de bits faisant partie du message d'origine). Les 3 bits ajoutés sont appelés des « bits de parité ».
Par exemple, les 4 bits "1011" sont encodés comme "1011 001".

13. Bob a reçu "1110 011". Sachant qu'il y a une erreur dans les 4 premiers bits, la corriger.

14. Prouver que s'il n'y a pas d'erreur, dans le message reçu on a $b_0 \oplus b_1 \oplus b_3 \oplus p_0 = 0$.
Proposer de même des inégalités liant p_1 puis p_2 à des b_i.

15. En déduire un algorithme en pseudo-code qui, s'il y a eu au plus 1 flip dans la transmission, peut détecter ce flip et la corriger.
Il doit prendre en entrée $(b_0, b_1, b_2, b_3, p_0, p_1, p_2)$ ayant subi au plus 1 flip et renvoyer un quadruplet contenant les bonnes valeurs pour les 4 premiers bits.
L'éventuel flip peut tout aussi bien avoir effectué un des bits b_i qu'un des bits de parité p_j.

16. Le code de Hamming (7,4) parvient-il à détecter deux flips ? À les corriger ?

17. Proposer un code de Hamming (8,4) qui peut détecter et corriger une deux flips (ou moins).

i. Cette probabilité correspond à un canal de communication pas très bon.
ii. C'est l'ordre de grandeur d'un e-mail de moins d'un paragraphe.
iii. TLS utilise $k = 16$ pour la taille de $check(m)$.
iv. Inutile de râler, vous ferez bien assez d'injectivité au prochain DS de maths.
v. En réalité, pour avoir cette garantie, il faudrait utiliser `unsigned char` . Cependant, c'est long à écrire sur une copie...

III - Humour

Que vaut : [5].

```
1 1 + - - + - + + + - + + + - + + - - - 1
```

Réponse : 2. Évidemment. Duh.

5. À l'origine la blague était en Python, mais elle marche aussi en C et en OCaml. Je pense qu'elle marche dans la plupart des langages existant.