Solutions des exercices

Solution de l'Exercice 1 – *Autour de la recherche dichotomique*

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

1. Il s'agit de la division entière, c'est à dire que l'on divise puis tronque le résultat. Dans les entiers positifs, cela revient à la partie entière inférieure de la division.
2. La fonction ne réalise aucun appel de fonction, et a une seule boucle. Montrons qu'elle termine en en exhibant un variant.

   Prouvons que $\texttt{fin} - \texttt{deb}$ est un variant. Cette quantité est :

   - Entière : cette quantité est la différence de deb et `fin`, qui sont entières (lignes 14 et 15).
   - Minorée : d'après la conditon de la boucle ona `deb < fin` (ligne 20), c'est à dire $\texttt{fin} - \texttt{deb} \geq 0$.
   - Strictement décroissante :

     Au début d'une itération quelconque, d'après la condition de la boucle (ligne 20) on a $\texttt{deb} < \texttt{fin}$ d'où $\texttt{deb} = \frac{\texttt{deb}+\texttt{deb}}{2} < \frac{\texttt{deb}+\texttt{fin}}{2} < \frac{\texttt{fin}+\texttt{fin}}{2} = \texttt{fin}$. On a encadré la fraction entre eux entiers. Avec $\texttt{milieu} = \left\lfloor \frac{\texttt{deb}+\texttt{fin}}{2} \right\rfloor$, on obtient grâce aux propriétés de la partie entière inférieure : $\texttt{deb} \leq \texttt{milieu} < \texttt{fin}$

     À la fin de cette itération quelconque, on a alors :

     – soit on sort de la boucle (on renvoie `milieu` ligne 22)

     – soit $\begin{cases} \texttt{deb'} = \texttt{milieu} + 1 > \texttt{deb} \\ \texttt{fin'} = \texttt{fin} \end{cases}$

     d'où $\texttt{fin'} - \texttt{deb'} < \texttt{fin} - \texttt{deb}$

     – soit $\begin{cases} \texttt{deb'} = \texttt{deb} \\ \texttt{fin'} = \texttt{milieu} < \texttt{fin} \end{cases}$

     d'où encore $\texttt{fin'} - \texttt{deb'} < \texttt{fin} - \texttt{deb}$

   `fin-deb` est donc bien un variant, d'où la terminaison de la boucle, et donc :

   La fonction `recherche_dichotomique` termine.

3. La seule ligne où la fonction peut renvoyer autre chose que -1 est la ligne 22. À cette ligne, elle renvoie `milieu` juste après avoir vérifié que `tab[milieu] == x` : la valeur renvoyée est donc bien un indice où trouver `x` dans le tableau.
4. Je ferai trois jolis dessins au tableau. Un dessin propre en pdf, c'est long èé.
5. Commençons par montrer le lemme suivant : « $\texttt{fin'} - \texttt{deb'} \leq \frac{\texttt{fin}-\texttt{deb}}{2}$ ». En lisant les cas dans le même ordre que précédemment, on a :

   - soit :

   $$\begin{aligned} \texttt{fin'} - \texttt{deb'} &= \texttt{fin} - \left(\left\lfloor \frac{\texttt{deb} + \texttt{fin}}{2} \right\rfloor + 1 \right) \\ &\leq \texttt{fin} - \frac{\texttt{deb} + \texttt{fin}}{2} \\ &\leq \frac{\texttt{fin} - \texttt{deb}}{2} \end{aligned}$$

   Or, comme $\texttt{fin'} - \texttt{deb'}$ est un entier, cette dernière inégalité implique :

   $$\texttt{fin'} - \texttt{deb'} \leq \left\lfloor \frac{\texttt{fin} - \texttt{deb}}{2} \right\rfloor$$

   - soit :

$$\begin{aligned}\texttt{fin'} - \texttt{deb'} &= \left\lfloor \frac{\texttt{deb} + \texttt{fin}}{2} \right\rfloor - \texttt{deb} \\ &\leq \frac{\texttt{deb} + \texttt{fin}}{2} - \texttt{deb} \\ &\leq \frac{\texttt{fin} - \texttt{deb}}{2}\end{aligned}$$

De même, cela entraine :

$$\texttt{fin'} - \texttt{deb'} \leq \left\lfloor \frac{\texttt{fin} - \texttt{deb}}{2} \right\rfloor$$

On a ainsi prouvé le lemme annoncé.

Notons L le nombre d'itérations de la boucle et numérotons celles-ci de 0 inclus à L exclu. Notons $(V_n)_{n\in[\![0;L[\![}$ la suite (finie) des valeurs du variant `fin-deb` au début des itérations. On a donc :

- $V_0 = \texttt{len} - \texttt{0} = \texttt{len}$
- d'après le lemme précédent, pour tout $n \in [\![0; L[\![: V_n \leq \frac{V_{n-1}}{2}$. On reconnait une majoration qui suit une suite géométrique.

Si $n = \lceil \log_2(\texttt{fin} - \texttt{deb}) \rceil \geq L$, le résultat attendu est immédiat. Sinon, étudions V_n :

$$\begin{aligned}V_n &\leq \frac{\texttt{fin} - \texttt{deb}}{2^{\lceil \log_2(\texttt{fin}-\texttt{deb}) \rceil}} \\ &\leq \frac{\texttt{fin} - \texttt{deb}}{2^{\log_2(\texttt{fin}-\texttt{deb})}} \\ &\leq \frac{\texttt{fin} - \texttt{deb}}{\texttt{fin} - \texttt{deb}} \\ &\leq 1\end{aligned}$$

Il s'ensuit que $V_n \leq 1$. Par l'absurde, si $n+1 <\leq L$, on aurait $V_{n+1} \leq \frac{1}{2}$. Comme la suite est une suite d'entiers, on aurait $V_{n+1} = 0$, et donc `deb` $\geq$ `fin` au début de l'itération $n+1$. Absurde d'après la condition du `while`. Il s'ensuit que l'itération $n+1$ n'existe pas, c'est à dire que $L \leq n+1$. On a bien prouvé :

$$\boxed{L \leq \lceil \log_2(\texttt{fin} - \texttt{deb}) \rceil + 1}$$

Solution de l'Exercice 2 – *Encore ?*

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. Le prototype est : `int somme_entiers(int n)` .
2. **a.** Sur l'entrée 5, cette fonction renvoie $0 + 1 + 2 + 3 + 4 + 5 = 15$.
   Sur l'entrée -2, elle renvoie 0 (on ne rentre jamais dans la boucle).
   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.
3. 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 `k` est :
   - Entière car `k` est déclaré comme `int k` (ligne 5).
   - Majorée par la valeur de `n` d'après `k <= n` (ligne 8). Comme la valeur de `n` ne change pas dans le code, ceci est bien une majoration par une borne constante.

- Strictement croissante cat (ligne 10) `k'` $=$ `k` $+1>$ `k`

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

La fonction `somme_entiers` termine.

4. Cf schéma que je ferai au tableau. Les schémas propres à l'ordinateur, ça prend du temps èé.
5. 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 $\sum_{k=0}^{n} l \geq 2^3 1$, 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 polynome 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 est positif, le polynôme estnégatif 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 3 – *Codes correcteurs*

1. Chaque bit a une probabilité $(1 - 10^{-4})$ de ne pas être flippé. Il y a 40000 bits. Les flips sont indépendants. Donc :

La probabilité qu'aucun bit ne soit flippé est $\left(1 - 10^{-4}\right)^{40000}$.

   C'est le 400001ème terme d'une suite géométrique de raison < 1 (dont le premier terme n'est pas *trop* proche de 1), il est donc logique que sa valeur annoncée par l'énoncée (0.018) soit faible.
2. Dans ce cas, il est probable que le $check(m)$ reçu et le $check(m')$ reçus diffèrent, et que Bob croit à tort que le message reçu n'est pas le bon.

   Cependant, cela est rare car $check(m)$ tient sur k bits avec k petit et a donc peu probablement subi des flips.
3. D'après les valeur snumériques de l'énoncé, la somme vaut $109+112+50+105 = 221+50+105 = 271+105 = 376$. Modulo $2^8 = 256$ cela fait 120.

La somme de contrôle est 120.

4. Dans les règles de l'encodage signé sur 8 bits, les 127 premiers codes (de "00...0" à "01...1") sont utilisés pour encoder $[\![0; 127[\![$ de manière non-signée. Autrement dit, peu importe que `char` soit signé ou non, les 127 ASCII seront encodés de la même façon.
5. Voici une implémentation de *check*. On y utilise les rappels de l'énoncé pour faire des calculs directement dans le type `char` :

prblm-code-corr.c
```c
char check(char message[], int len) {
  char checksum = 0;
  int indice = 0;
  while (indice < len) {
    checksum = checksum + message[indice];
    indice = indice +1;
  }
  return checksum;
}
```

6. On remarque que l'argument `len` ne sert pour l'instant à rien. Plus tard dans l'année, on pourra à 'laide d'un `assert` garantir que `len` > 0.

prblm-code-corr.c
```c
char premiere_lettre(char message[], int len) {
  return message[0];
}
```

7. On parcourt le tableau. Lorsque la case en cours est un `'.'`, on augmente de 1 le compteur de `'.'` rencontrés. On renvoie ce compteur à la fin :

prblm-code-corr.c
```c
int nombre_point(char message[], int len) {
  int nb_pt = 0;
  int indice = 0;
  while (indice < len) {
    if (message[indice] == '.') {
      nb_pt = nb_pt +1;
    }
    indice = indice +1;
  }
  return nb_pt;
}
```

8. Le principe est le même qu'en question précédente, si qu'on teste si la case en cours *et les deux suivantes* forment `"lol"`. Il faut faire attention à la borne de fin de `indice` pour que `indice+2` ne dépasse pas :

prblm-code-corr.c
```c
int nombre_lol(char message[], int len) {
  int nb_lol = 0;
  int indice = 0;
  while (indice+2 < len) {
    if (   message[indice] == 'l'
        && message[indice+1] == 'o'
        && message[indice+2] == 'l'
       ) {
      nb_lol = nb_lol +1;
    }
    indice = indice +1;
  }
  return nb_lol;
}
```

9. En prenant dans chaque triplet le caractère le plus courant, on obtient :

Le message était « Lynn Conway ».

10. La difficulté est de correctement manipuler les indices. On peut utiliser un indice pour chacun des tableaux, ou réaliser que quand un indice avance de 1 (on lit une lettre) l'autre avance de 3 (on l'écrit en triple), et qu'une multiplication par 3 convient donc :

prblm-code-corr.c

```c
void encode(char msg_triple[], char msgAlice[], int len) {
  int indice = 0;
  while (indice < len) {
    msg_triple[3*indice] = msgAlice[indice];
    msg_triple[3*indice+1] = msgAlice[indice];
    msg_triple[3*indice+2] = msgAlice[indice];
    indice = indice +1;
  }
  return;
}
```

11. À chaque itération de la boucle, on lit un triplet. Si deux des caractères de ce triplet sont les mêmes, c'est e caractère à écrire. Si par contre les 3 caractères sont distincts, on ne peut pas décoder et on renvoie `false`. La manipulation des indices est la même que précédemment :

prblm-code-corr.c

```c
bool decode(char msgBob[], char msg_recu[], int len) {
  int indice = 0;
  while (indice < len) {

    // On écrit dans msgBob le caractère qui est (au moins) en double :
    if (   msg_recu[3*indice] == msg_recu[3*indice+1]
        || msg_recu[3*indice] == msg_recu[3*indice+2]
       ) {
      msgBob[indice] = msg_recu[3*indice];
    }
    else {
      if (msg_recu[3*indice+1] == msg_recu[3*indice+2]) {
        msgBob[indice] = msg_recu[3*indice+1];
      }
      else { // Sinon, aucun caractère en double
        return false;
      }
    }

    indice = indice +1;
  }

  return true;
}
```

12. Au lieu de tripler chaque lettre, on quintuple. Ou plus. L'inconvénient étant que l'on consomme de plus en plus de taille.

13. On devrait avoir :

- $p_0 = b_0 \oplus b_1 \oplus b_3 = 1 \oplus 1 \oplus 0 = 0$. C'est bien le cas.
- $p_1 = b_1 \oplus b_2 \oplus b_3 = 1 \oplus 1 \oplus 0 = 0$. Ce n'est pas le cas.
- $p_2 = b_0 \oplus b_2 \oplus b_3 = 1 \oplus 1 \oplus 0 = 0$. Ce n'est pas le cas.

Ainsi, on peut conclure que le bit erronné est commun aux deux derniers cas : c'est donc b_2 ou b_3. Mais comme p_0 est correct, b_3 n'est pas erronné : c'est b_2 qui est erronné.

Le message corrigé est `1100 011`.

14. $p_1 \oplus b_1 \oplus b_2 \oplus b_3 = 0$ et $p_2 \oplus b_0 \oplus b_2 \oplus b_3 = 0$.

15. Supposons qu'il y ait au plus 1 flip.
S'il y a un flip, la valeur d'un des bits de parité n'est pas cohérente comme dans l'exemple précédente. En conséquence, si toutes les valeurs des bits de parité sont cohérentes, on conclut qu'il n'y a pas d'erreur à corriger.

Sinon, on peut remarquer que chacun des b_i apparait dans plusieurs p_j. On peut donc trouver d'où provient l'erreur en « intersectant » comme dans l'exemple précédent :

- Si uniquement la valeur de p_0 est incohérente, c'est p_0 lui-même qui a été flippé. En effet, le flip b_0, b_1 ou b_3 aurait aussi affecté l'un des autres p_j. Il faut donc dé-flipper la valeur de p_0.
- Idem pour p_1 et p_2.
- Sinon, si les valeurs de p_0, p_1 et p_2 sont toutes trois incohérentes, alors c'est b_3 qui a été flippé car c'est le seul bit à apparaitre dans les 3. Il faut donc le déflipper.
- Sinon, si uniquement les valeurs de p_0 et p_1 sont incohérentes, c'est b_1 qui a été flippé. En effet, seuls b_1 et b_3 sont en communs, mais ce n'est pas b_3 car p_2 est correct. Il faut donc déflipper b_1.
- Idem pour les deux autres cas.

Le disjonction de cas précédente permet de créer un algorithme qui convient : on déflippe l'éventuel bit flippé, puis on renvoie les 4 premiers bits.

16. Allez voir sur Wikipédia. Hamming (7,4) peut détecter mais pas corriger.
Attention, Wikipédia organise ses bits d'une façon différente de moi (et plus pertinente informatique). Le but est que la position des bits permettre d'écrire plus facilement les "intersections d'erreurs".

17. Allez voir sur Wikipédia.