Correction du DS 1

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici. De même, les exercices issus des TD et TP ont leur correction déjà publiée.

Exercices

Exercice 1 :

```
int exo1(int n)
{
  if (n <= 0 || n > 1000000000) exit(1); // Vérification faite tout de même.
  while (2*n <= 1000000000) n *= 2;
  return n;
}
```

Exercice 2 :

Terminaison : On prend pour variant $\lfloor \log_2 \frac{10^9}{\texttt{n}} \rfloor$ (mais $10^9 -$ `n` conviendrait aussi). Cette valeur diminue d'un à chaque étape car `n` est doublé, il s'agit d'un entier au vu de la formule et puisque `n` reste toujours inférieur à un milliard au vu de la condition de boucle, le variant proposé est bien positif.

Correction : On se sert de l'invariant de boucle « la valeur actuelle de `n` divisée par sa valeur initiale est une puissance de deux, et la valeur actuelle de `n` est inférieure ou égale à un milliard », qui est clairement vrai avant la boucle, qui reste vrai à chaque tour car `n` est doublé (et au vu de la condition de boucle cela donne une valeur inférieure à un milliard) et dont la véracité après la boucle confirme que la fonction est correcte car à ce moment-là le double de `n` est strictement supérieur à un milliard alors que ce n'était pas le cas auparavant. Mathématiquement, cela correspond à ce qu'on voulait.

Exercice 3 :

Le nombre 10^{10} n'est pas représentable avec le type `int`, donc en changeant la valeur dans la condition de boucle (et la vérification éventuelle) la fonction ne terminerait pas (`n`, de même que `2*n`, resterait de type entier mais la constante `10000000000`, automatiquement de type `long int`, ne pourrait jamais être atteinte).

La solution revient simplement à utiliser des `long int` pour l'argument, et donc la valeur de retour, quitte à caster l'argument en une variable locale si on veut démarrer avec un argument de type `int`.

Une éventuelle adaptation peut aussi donner une fonction qui termine, mais dans ce cas elle ne serait pas correcte (par exemple écrire `10 * 1000000000`, qui donne par ailleurs un comportement indéterminé).

Exercice 4 :

Les chiffres du développement décimal s'obtiennent selon l'algorithme usuel de conversion (en base dix, c'est aussi une formule vue en cours de mathématiques). Avec des fractions, il n'y a aucun problème d'approximation.

La partie compliquée est de délimiter la période. Pour commencer, nous allons utiliser une méthode naïve en cherchant si un résidu modulo le dénominateur a déjà été rencontré, par une recherche séquentielle dans un tableau.

```
int position_dans_tableau(int* tableau, int taille, int v) // Première occurrence, -1 si absent !
{
  for (int i = 0 ; i < taille ; i += 1)
  {
    if (tableau[i] == v) return i;
  }
  return -1;
}

bool simplement_normal(int numer, int denom)
{
  if (numer < 0) numer = -numer; // On évite d'avoir des soucis avec les négatifs.
  int r = numer % denom; // Seule la partie fractionnaire est intéressante.
  int* restes = malloc(denom * sizeof(int));
  int* chiffres = malloc(denom * sizeof(int));
  int pos_debut;
  int indice = 0;
  bool trouve = false;
  while (!trouve)
  {
    r *= 10;
    int chiffre = r / denom;
    r %= denom;
    int pos = position_dans_tableau(restes, indice, r);
    chiffres[indice] = chiffre;
    restes[indice] = r;
    indice += 1;
    if (pos != -1)
    {
      pos_debut = pos+1;
      trouve = true;
    }
  }
  int nombre_occurrences[10] = { 0 };
  for (int i = pos_debut ; i < indice ; i += 1)
  {
    nombre_occurrences[chiffres[i]] += 1;
  }
  free(restes); free(chiffres);
  for (int i = 1 ; i < 10 ; i += 1)
  {
    if (nombre_occurrences[i] != nombre_occurrences[0]) return false;
  }
  return true;
}
```

Exercice 5 :

Exactement le même exercice mais on remplace les occurrences de `10` par `2`. La boucle finale peut alors être remplacée par un simple test et le tableau `nombre_occurrences` par deux variables (voire une) qu'on remplit en utilisant un test.

Exercice 6 :

Cette fois-ci, la taille du tableau est variable, donc quelques modifications s'imposent.

```
bool simplement_normal_base_b(int numer, int denom, int base)
{
  if (numer < 0) numer = -numer; // On évite d'avoir des soucis avec les négatifs.
  int r = numer % denom; // Seule la partie fractionnaire est intéressante.
  int* restes = malloc(denom * sizeof(int));
  int* chiffres = malloc(denom * sizeof(int));
  int pos_debut;
  int indice = 0;
  bool trouve = false;
  while (!trouve)
  {
    r *= base;
    int chiffre = r / denom;
    r %= denom;
    int pos = position_dans_tableau(restes, indice, r);
    chiffres[indice] = chiffre;
    restes[indice] = r;
    indice += 1;
    if (pos != -1)
    {
      pos_debut = pos+1;
      trouve = true;
    }
  }
  int* nombre_occurrences = malloc(base * sizeof(int));
  for (int i = pos_debut ; i < indice ; i += 1)
  {
    nombre_occurrences[chiffres[i]] += 1;
  }
  free(restes);
  free(chiffres);
  for (int i = 1 ; i < base ; i += 1)
  {
    if (nombre_occurrences[i] != nombre_occurrences[0])
    {
      free(nombre_occurrences);
      return false;
    }
  }
  free(nombre_occurrences);
  return true;
}
```

Problème

Question P1 :

On a $u_2 = 4$, soit $\overline{100^2}$. Alors u_3 est un de moins que $\overline{100^3}$, soit $9 - 1 = 8$, qui s'écrit $\overline{22^3}$. Ensuite, u_4 est un de moins que $\overline{22^4}$, soit $10 - 1 = 9$, qui s'écrit $\overline{21^4}$. Ensuite, u_5 est un de moins que $\overline{21^5}$, soit $11 - 1 = 10$, qui s'écrit $\overline{20^5}$. Ensuite, u_6 est un de moins que $\overline{20^6}$, soit $12 - 1 = 11$, qui s'écrit $\overline{15^6}$.

À partir de là, on aura encore cinq valeurs égales à 11 puis une décroissance unité par unité pour atteindre finalement zéro.

Question P2 :

On a $421 = 2 \times 12^2 + 11 \times 12 + 1$, d'où l'écriture `[1, 11, 2]`.

Question P3 :

```
long depuis_base(long* tableau, size_t taille, long base)
{
  long reponse = tableau[taille-1];
  for (int i = taille - 2 ; i >= 0 ; i -= 1)
  {
    reponse = base * reponse + tableau[i];
  }
  return reponse;
}
```

Question P4 :

```
size_t nombre_chiffres_base(long entier, long base)
{
  size_t reponse = 1;
  while (entier >= base)
  {
    entier /= base;
    reponse += 1;
  }
  return reponse;
}
```

Question P5 :

```
long* vers_base(long entier, long base)
{
  size_t taille = nombre_chiffres_base(entier, base);
  long* reponse = malloc(taille * sizeof(long));
  for (size_t i = 0 ; i < taille ; i += 1)
  {
    reponse[i] = entier % base;
    entier /= base;
  }
  return reponse;
}
```

Question P6 :

```
bool nul(long* tableau, size_t taille)
{
  for (size_t i = 0 ; i < taille ; i += 1)
  {
    if (tableau[i] != 0) return false;
  }
  return true;
}

long goodstein(long entier) // Version adaptée à la preuve de terminaison
{
  size_t taille = nombre_chiffres_base(entier, 2);
  long* tableau = vers_base(entier, 2);
  int base = 2;
  while (!nul(tableau, taille))
  {
    base += 1;
    // Ici on peut s'amuser à calculer depuis_base(tableau, taille, base) - 1
    size_t indice = 0;
    while (tableau[indice] == 0) // inutile de rester si indice < taille vu l'autre boucle
    {
      tableau[indice] = base - 1;
      indice += 1;
    }
    tableau[indice] -= 1; // même remarque
  }
  free(tableau); // important
  return base - 2; // vu la consigne
}
```

Question P7 :

On observe que si le tableau ne contient plus qu'une valeur, l'entier décroît à chaque tour de boucle (il n'y a que le « chiffre » des unités). Quand il y a deux valeurs et celle de poids fort est un 1, l'entier est le même à chaque tour de boucle (la base augmente d'un donc $+1$, le chiffre des unités diminue d'un donc -1). Dans les autres cas, l'augmentation de la base fait augmenter l'entier. Ainsi, le maximum est atteint pendant toute la période où le tableau est de la forme `[x, 1]`. En particulier, pour `x` valant `0`, la base et l'entier (qu'on nommera b) sont identiques (tout entier supérieur ou égal à deux s'écrit 10 dans sa propre base). À l'étape suivante, le nombre sera pour la dernière fois encore b mais sur un seul chiffre, et il faudra b étapes que ce nombre pour arriver à zéro, la base étant alors $2b + 1$. Cela donne donc avec la convention de l'énoncé $2b - 1$ étapes en posant b le maximum.

Question P8 :

En considérant le tableau représentant à chaque étape dans la base correspondante, on observe une stricte décroissance de ce tableau selon l'ordre lexicographique. Le tableau peut donc servir de variant selon la définition étendue.