Questions tirées d'anciens examens (solutions) - ICC-C

1. Petites questions indépendantes

Question. Soient x, y et z trois nombres entiers strictement positifs qui peuvent s'exprimer chacun avec **12 bits** maximum.

Sachant qu'un `unsigned int` C contient 32 bits, et qu'un `unsigned long long` en contient 64 : Écrivez, en C, une fonction `pow_mod` qui prend en paramètres les entiers x, y et z décrits ci-dessus. Elle doit renvoyer $x^y \bmod z$.

Rappel : $(a \times b) \bmod c = ((a \bmod c) \times b) \bmod c$.

Réponse. Il est évident qu'on ne peut pas calculer d'abord x^y. Nous aurions besoin de plus de 64 000 bits pour représenter cette quantité. Il faut donc appliquer le $\bmod z$ à chaque itération de la boucle. Puisqu'on part de 12 bits, la multiplication intermédiaire nécessite au maximum 24 bits et tient donc confortablement dans un `unsigned int`. Il n'était pas nécessaire d'utiliser un `unsigned long long`, mais cela n'a pas été pénalisé.

```c
unsigned int pow_mod(unsigned int x, unsigned int y, unsigned int z) {
  unsigned int result = 1;
  for (unsigned int i = 0; i < y; i++) {
    result = (result * x) % z;
  }
  return result;
}
```

La version ci-dessus est presque parfaite. Sa complexité est $\Theta(y)$. On peut significativement l'améliorer en $\Theta(\log y)$ avec l'algorithme d'exponentielle discrète vu en ICC-T :

```c
unsigned int pow_mod(unsigned int x, unsigned int y, unsigned int z) {
  if (y == 1)
    return x;
  else if (y % 2 == 0)
    return pow_mod((x*x) % z, y / 2, z);
  else
    return (x * pow_mod((x*x) % z, (y-1) / 2, z)) % z;
}
```

2. Recherche de sous-chaînes

On s'intéresse à la recherche de sous-chaînes dans des chaînes de caractères en C. On peut penser à la recherche d'un mot dans un texte, par exemple.

On dit qu'une chaîne s de longueur m apparaît comme sous-chaîne de t de longueur n à l'indice p, que l'on note $H(s, t, p)$, si et seulement si :

- $0 \leq p \leq (n - m)$, et
- $\forall k, 0 \leq k < m : s[k] = t[p + k]$.

Question. Écrivez, en C, une fonction `teste_sous_chaine` qui prend en paramètre : une chaîne s, une chaîne t et un indice p, et qui renvoie `true` si $H(s, t, p)$ est vrai, et `false` sinon.

Réponse. Le type des indices doit être `size_t`, puisqu'il s'agit d'indices dans des chaînes de caractères, qui sont des tableaux.

```
bool teste_sous_chaine(const char *s, const char *t, size_t p) {
  size_t m = strlen(s);
  size_t n = strlen(t);
  if (p > n - m) { // il faut que 0 <= p <= n-m pour être vrai
    return false;
  } else {
    // si il existe 0 <= k < m tel que s[k] != t[p + k], alors c'est faux
    for (size_t k = 0; k < m; k++) {
      if (s[k] != t[p + k]) {
        return false;
      }
    }
    // sinon c'est vrai
    return true;
  }
}
```

Question. Écrivez, en C, une fonction `trouve_sous_chaines` qui prend en paramètre : une chaîne s et une chaîne t, et qui renvoie une liste de tous les indices p tels que $H(s, t, p)$ est vrai.

Vous pouvez supposer l'existence des structure et fonctions suivantes pour représenter une liste chaînée d'indices :

```
typedef struct plist { ... } plist_t;
plist_t make_empty_plist();
void plist_add_last(plist_t *list, P p);
```

où `P` est le type que vous avez choisi pour représenter un indice.

Réponse.

```
plist_t trouve_sous_chaines(const char *s, const char *t) {
  plist_t result = make_empty_plist();
  size_t n = strlen(t);
  for (size_t p = 0; p <= n; p++) { // attention : <= ici !
    if (teste_sous_chaine(s, t, p)) {
      plist_add_last(&result, p);
    }
  }
  return result;
}
```

3. Estimation de π

On souhaite écrire un programme C qui estime la valeur de π. Pour ce faire, nous procédons par simulation avec la technique suivante.

On considère un point (x, y) avec x et y uniformément distribués sur l'intervalle (réel) $[-1, 1]$. La probabilité p que ce point se situe sur le disque unité est égale à l'aire du disque unité ($\pi r^2 = \pi$, puisque $r = 1$) divisée par l'aire du carré circonscrit ($2 \cdot 2 = 4$). Nous avons donc $p = \frac{\pi}{4}$.

Nous allons estimer la valeur de p par simulation, et par là, estimer celle de $\pi = 4p$.

Important ! Dans cette question, vous n'avez pas le droit d'utiliser les fonctions et constantes mathématiques de C, sauf là où c'est explicitement autorisé ! Vous ne pouvez utiliser que les opérations élémentaires sur les entiers et les nombres à virgule flottante. C'est logique, puisqu'on cherche à estimer π nous-mêmes.

Dans chaque sous-question, vous pouvez utiliser les fonctions définies pour les sous-questions précédentes.

Question. Écrire (en C) la fonction `is_in_unit_disc` qui prend en entrée deux réels x et y et qui indique (retourne) si oui ou non le point (x, y) est sur le disque unité, c'est-à-dire si $x^2 + y^2 \leq 1$.

Réponse.

```c
bool is_in_unit_disc(double x, double y) {
  return x*x + y*y <= 1.0;
}
```

Question. Écrire (en C) la fonction `simulate_one` qui ne prend aucun argument. Elle génère aléatoirement un point tel que décrit ci-dessus, puis retourne si oui ou non ce point est dans le disque unité. On suppose qu'il existe une fonction `rand_double()` qui renvoie un `double` aléatoire dans l'intervalle $[-1, 1]$, que vous pouvez utiliser pour cette question.

Réponse.

```c
bool simulate_one() {
  double x = rand_double();
  double y = rand_double();
  return is_in_unit_disc(x, y);
}
```

Question. Écrire (en C) la fonction `estimate_pi` qui prend en entrée un entier strictement positif n, et qui retourne une estimation de la valuer de π. Pour ce faire, elle doit utiliser la technique décrite au début de cette section. Elle teste n points aléatoires et compte quelle proportion $\hat{p}$ d'entre eux tombent dans le disque unité. L'estimation de π est alors $\hat{\pi} = 4\hat{p}$.

Réponse.

```c
double estimate_pi(int n) { // ou unsigned int
  int in_disc = 0;
  for (int i = 0; i < n; i++) {
    if (simulate_one()) {
      in_disc += 1;
    }
  }
  double p_hat = ((double) in_disc) / ((double) n);
  return 4.0 * p_hat;
}
```

Question. Finalement, nous aimerions savoir quelle valeur de n serait suffisamment grande pour avoir une estimation « suffisamment bonne » de π. Rappelez-vous que les nombres en virgule flottante permettent de garantir des bornes sur les *erreurs relatives*. Nous voulons donc trouver le plus petit entier n tel que l'*erreur relative* de $\hat{\pi}$ par rapport à π soit inférieure à une certaine tolérance.

Écrire (en C) une fonction `main` qui détermine la plus petite valeur de n telle que l'erreur relative respecte une tolérance de 10^{-6}. Elle affiche ensuite cette valeur n à l'écran ainsi que la valeur correspondante de $\hat{\pi}$. Dans cette question, vous pouvez utiliser la constante prédéfinie `M_PI` pour avoir la valeur « exacte » de π (celle par rapport à laquelle vous devez calculer vos erreurs relatives).

(Évidemment, étant donnée la nature aléatoire de la simulation, exécuter plusieurs fois `main` peut résulter en différentes valeurs de n et de $\hat{\pi}$. On ne fait pas attention à cela pour cette question.)

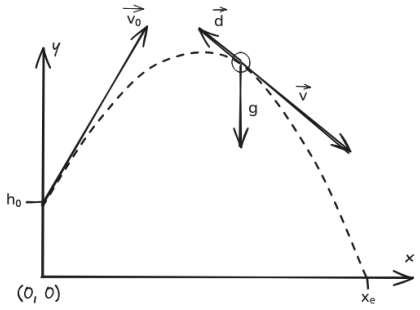
Vous pouvez utiliser la fonction `abs(x)` pour cette question.

Réponse.

```c
int main() {
  const int TOLERANCE = 1e-6; // ou 0.000001

  int n = 1;
  double pi_hat = estimate_pi(n);
  while (abs((pi_hat - M_PI) / M_PI) > TOLERANCE) {
    n++;
    pi_hat = estimate_pi(n);
  }

  printf("n = %d\n", n);
  printf("pi_hat = %f\n", pi_hat);

  return 0;
}
```

4. Balistique

On cherche à simuler, en C, la trajectoire d'un projectile soumis à la gravité et aux frottements de l'air. On peut représenter la scène comme suit.



Les paramètres du système sont les suivants :

- $p_{y0} \geq 0$: la hauteur initiale du projectile (en abscisse, il démarre toujours en 0, c'est-à-dire $p_{x0} = 0$)
- (v_{x0}, v_{y0}), sa vitesse initiale (les deux valeurs étant > 0).
- $0 \leq d < 1$, le coefficient de frottement.
- $g > 0$, la constante de gravité.

On cherche à calculer (approximativement) x_e, la distance depuis l'origine où le projectile tombera au sol. À tout moment, le projectile est caractérisé par sa position (p_x, p_y) et sa vitesse (v_x, v_y). Pour simuler (plutôt que résoudre analytiquement), on utilise un pas de temps Δt petit mais pas infinitésimal. Par exemple, on pourrait avoir $\Delta t = 10^{-3}$ s. On peut alors calculer les nouvelles valeurs caractéristiques du projectile par intégration numérique, comme suit :

$$v_{x(n+1)} = (v_{xn} \cdot (1-d)) \cdot \Delta t v_{x(n+1)} = (v_{xn} \cdot (1-d)) \cdot \Delta t$$

$$v_{y(n+1)} = \big(v_{yn} \cdot (1-d) - g\big) \cdot \Delta t v_{y(n+1)} = \big(v_{yn} \cdot (1-d) - g\big) \cdot \Delta t$$

$$p_{x(n+1)} = p_{xn} + v_{x(n+1)} \cdot \Delta t p_{x(n+1)} = p_{xn} + v_{x(n+1)} \cdot \Delta t$$

$$p_{y(n+1)} = p_{yn} + v_{y(n+1)} \cdot \Delta t p_{y(n+1)} = p_{yn} + v_{y(n+1)} \cdot \Delta t$$

Question. Écrivez, en C, une fonction simulation qui simule le système ci-dessous. Elle doit prendre en paramètre les différents paramètres du système. Elle doit renvoyer la distance x_e à laquelle le projectile tombe au sol. N'hésitez pas à définir des fonctions annexes, si cela peut vous aider.

Réponse. Il s'agit essentiellement de traduire les équations en code C. On ajoute une boucle `do..while` pour s'arrêter quand on a touché le sol. Attention à ne pas utiliser `py == 0.0` pour cette condition ! En fait, on ne peut même pas utiliser `abs(py) < EPSILON`, car la vitesse verticale par pas de temps pourrait devenir supérieure à `EPSILON`. Un test plus simple et correct est donc `py <= 0.0` pour s'arrêter (ou `py > 0.0` pour continuer).

```c
const double Dt = 0.001; // Δt

double simulation(double py0, double vx0, double vy0, double d, double g) {
  // optionnel : quelques assert pour vérifier les contraintes
  assert(py0 > 0.0 and vx0 > 0.0 and vy0 > 0.0);
  assert(0.0 <= d and d < 1.0);
  assert(g > 0.0);

  // initialisation
  double px = 0.0, py = py0, vx = vx0, vy = vy0;
```

```
  // simulation
  do {
    vx = vx * (1.0 - d) * Dt;
    vy = (vy * (1.0 - d) - g) * Dt;
    px += vx * Dt;
    py += vy * Dt;
  } while (py > 0.0);

  // une fois qu'on a touché le sol, renvoyer la distance horizontale
  return px;
}
```

Notez que le code ci-dessus ne fonctionnerait pas tout à fait avec une boucle `while`. Si $p_{y0} = 0$ (qui est un cas valide), on n'entrerait jamais dans la boucle car on « toucherait le sol » au lancement. Le `do..while` évite ce problème.

Question. (difficile) En balistique, on sait qu'évaluer les distances est difficile. Si on dispose d'un lanceur qui lance un projectile toujours à la même vitesse scalaire v_0, et qu'on doit toucher une cible à une distance donnée x_c, la seule chose qu'on peut contrôler est l'angle $0 < \theta < \pi/4$ du lanceur par rapport à l'horizontale.

Écrivez, en C, une fonction balistique qui cherche le bon angle de lancement. Elle prend en paramètre les paramètres du système, sauf v_{x0} et v_{y0} ; à la place, elle reçoit la vitesse scalaire v_0 et la distance x_c de la cible. Elle doit renvoyer θ tel que la cible est touchée (à une petite constante près).

On procède par dichotomie : on sait que $0 < \theta < \pi/4$. On teste donc d'abord $\theta = \frac{0+\pi/4}{2} = \pi/8$. Si on vise trop loin, on teste ensuite $\theta = \frac{0+\pi/8}{2} = \pi/16$. Si on vise trop près, on teste $\theta = (\pi/8 + \pi/4) = 3\pi/16$. On continue ainsi de suite jusqu'à tomber « suffisamment près » de la cible.

On pourra supposer que la vitesse initiale est suffisamment grande pour qu'il existe un $\theta \in \mathbb{R}$ qui atteigne la cible. Pour rappel, étant donnés v_0 et θ, la vitesse initiale sera $(v_0 \cos\theta, v_0 \sin\theta)$.

Réponse. La principale difficulté ici était d'adapter la logique de dichotomie, que nous avions étudiée sur des entiers, à des nombres en virgule flottante. Une fois cela accepté, l'implémentation est raisonnable. Remarquez qu'on utilise 1 mètre complet comme tolérance pour toucher la cible. On ne parle pas ici de gérer l'erreur des double ; il s'agit plutôt d'une notion du domaine : si je vise une cible de 2 m de diamètre, par exemple, elle sera touchée même si je vise à 1 m de son centre théorique.

```
const double TARGET_TOLERANCE = 1.0;

double balistique(double py0, double v0, double d, double g, double xc) {
  double thetaMin = 0.0;
  double thetaMax = PI/4.0;

  double theta, xe; // initialisées dans la boucle mais on en a besoin après
  do {
    // tentative de "tir"
    theta = (thetaMin + thetaMax) / 2.0;
    double vx0 = v0 * cos(theta);
    double vy0 = v0 * sin(theta);
    xe = simulation(py0, vx0, vy0, d, g);

    // réduire l'intervalle d'exploration
    if (xe < xc)
      thetaMin = theta;
```

```
    else
      thetaMax = theta;
  } while (abs(xe - xc) > TARGET_TOLERANCE);

  return theta;
}
```

5. Développement en série

On souhaite implémenter nous-mêmes, en C, le calcul du logarithme naturel. Pour ce faire, nous allons nous baser sur le développement en série de Maclaurin de $\ln(1+x)$, qui est :

$$\ln(1+x) = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{i} x^i$$

Puisque nous ne pouvons pas réellement aller jusqu'à ∞, nous allons limiter la série jusqu'à un entier n donné, et donc calculer une approximation de la fonction jusqu'au rang n :

$$\ln(1+x) \approx f_n(x) = \sum_{i=1}^{n} \frac{(-1)^{i+1}}{i} x^i$$

Comment savoir quelle valeur de n est suffisamment grande pour obtenir une approximation suffisamment bonne ? Nous allons écrire un programme pour le déterminer.

Important ! Dans cette question, vous n'avez pas le droit d'utiliser les fonctions mathématiques de C, sauf là où c'est explicitement autorisé ! Vous ne pouvez utiliser que les opérations élémentaires sur les entiers et les nombres à virgule flottante. C'est logique, puisqu'on cherche à implémenter ln nous-mêmes.

Dans chaque sous-question, vous pouvez utiliser les fonctions définies pour les ous-questions précédentes.

Question. Écrire (en C) la fonction `int_pow` qui prend en entrée un réel x et un entier positif ou nul y et qui retourne x^y.

Réponse. Voici deux versions possibles, l'une récursive, l'autre itérative :

```c
double int_pow(double x, int y) {
  if (y == 0) {
    return 1.0;
  } else {
    return x * int_pow(x, y - 1);
  }
}
```

```c
double int_pow(double x, int y) {
  double r = 1.0;
  for (int i = 1; i <= y; i++) {
    r = r * x;
  }
  return r;
}
```

Question. Écrire (en C) la fonction `log1p_approx` qui prend en entrée un réel x et un entier strictement positif n, et qui retourne la valeur de $f_n(x)$ (l'approximation au rang n de $\ln(1+x)$ avec le développement en série de Maclaurin). Vous devez respecter la définition de $f_n(x)$ ci-dessus.

Réponse. Voici une version possible :

```c
double log1p_approx(double x, int n) {
  double result = 0.0;
  for (int i = 1; i <= n; i++) {
    result += (int_pow(-1.0, i + 1) * int_pow(x, i) / i);
  }
  return result;
}
```

Question. Il faut maintenant pouvoir tester si un n donné est « suffisamment bon ». Écrire (en C) la fonction `suffisamment_bon` qui prend en entrée un entier strictement positif n et un réel strictement positif `tolerance`, et qui retourne `true` si et seulement si n est « suffisamment bon ». Nous dirons que n est suffisamment bon si $|f_n(x) - \ln(1+x)| \leq$ `tolerance` pour tous les $x \in [0,\ 1[$ (0 inclus, 1 exclu). En pratique, on testera seulement les valeurs par incrément de 1/256 (donc, $x \in \left\{0, \frac{1}{256}, \frac{2}{256}, \ldots, \frac{255}{256}\right\}$).

Pour calculer la valeur « exacte » de $\ln(1+x)$, utilisez la fonction C `log1p(x)` (ceci est donc la seule fonction mathématique de C que vous avez le droit d'utiliser).

Réponse. Voici une version possible :

```
bool suffisamment_bon(int n, double tolerance) {
  for (double x = 0.0; x < 1.0; x += (1.0/256.0)) {
    double difference = log1p_approx(x, n) - log1p(x);
    if (difference > tolerance || difference < -tolerance) {
      return false;
    }
  }
  return true;
}
```

Question. Finalement, nous voulons trouver le plus petit n qui nous donne une fonction f_n avec une tolérance donnée. Écrire (en C) une fonction `main` qui affiche le plus petit entier strictement positif n tel que n soit « suffisamment bon » avec la tolérance $1/1000$.

Réponse. Voici une version possible :

```
int main() {
  int n = 1;
  while (!suffisamment_bon(n, 1e-3)) { // ou 0.001
    n++;
  }
  printf("%d\n", n);
  return 0;
}
```

6. Conception de programme pour le Père Noël

Le Père Noël souhaite que vous l'aidiez à gérer sa distribution de cadeaux. Il vous confie sa division suisse.

Pour la Suisse, il possède plusieurs fabriques de cadeaux. Une fabrique est caractérisée par son nom (une chaîne de caractères), le poids total des cadeaux qu'elle peut (encore) produire cette année (comment ça, ça n'a aucun sens ? chut, c'est magique), et la liste des cadeaux produits mais pas encore distribués.

Un cadeau est caractérisé par son nom, son poids, et le code postal de la commune où il doit être distribué. Une commune est caractérisée par son nom, son code postal, et la liste des cadeaux qui y ont été distribués. Afin d'optimiser la logistique, on voudrait – dans la mesure du possible – fabriquer dans la même fabrique les cadeaux à destination de la même commune.

Question. Donnez un code C possible pour les structures de données permettant de modéliser :

- un cadeau (`cadeau_t`)
- une fabrique
- une commune

Vous pouvez bien sûr définir d'autres types de données, si vous le souhaitez.

Vous pouvez supposer qu'il existe une structure `cadeau_list_t` représentant une liste chaînée de `cadeau_t`.

```c
typedef struct cadeau_node {
  cadeau_t cadeau;
  struct cadeau_node *next;
} cadeau_node_t;
typedef struct cadeau_list {
  cadeau_node_t *first;
  cadeau_node_t *last;
} cadeau_list_t;
```

Réponse.

```c
// Deux typedef optionnels pour que le reste soit plus lisible
typedef unsigned int poids_t; // aussi possible : int, float, double
typedef unsigned int codepostal_t; // aussi possible : int

typedef struct cadeau {
  char *nom;
  poids_t poids;
  codepostal_t destination;
} cadeau_t;

typedef struct fabrique {
  char *nom;
  poids_t poids_restant; // éventuellement : poids_total_possible
  cadeau_list_t cadeaux_produits;
} fabrique_t;

typedef struct commune {
  char *nom;
  codepostal_t code_postal;
  cadeau_list_t cadeaux_distribues;
} commune_t;
```

Question. Les fonctionnalités suivantes seront nécessaires à la gestion des cadeaux :

1. Tester si un cadeau peut encore être produit dans une fabrique donnée.
2. Tester si une fabrique est idéale pour un cadeau ; c'est le cas si elle a déjà un cadeau à destination du même code postal dans ses produits non distribués.
3. Produire un cadeau dans une fabrique donnée. Si cette fabrique ne peut pas/plus produire le cadeau, renvoyer `false`.
4. Trouver, parmi un tableau de fabriques, une fabrique qui peut produire un cadeau donné. Si possible, choisir une fabrique « idéale ». Produire le cadeau dans la fabrique sélectionnée. Si aucune fabrique ne peut accueillir le cadeau, renvoyer `false`.
5. Étant données une fabrique F et une commune C, distribuer à C tous les cadeaux produits dans F qui sont à destination de C.

Écrire, en C, les **prototypes** de 5 fonctions réalisant les fonctionnalités ci-dessus. Ne pas implémenter ces fonctions.

Réponse. Il s'agit d'une question de conception, ici. Il y a plusieurs bonnes réponses possibles. On veillera cependant à utiliser les `const *` de manière adéquate.

```c
bool peut_produire(const frabrique_t *fabrique, const cadeau_t *cadeau);

bool est_ideale(const frabrique_t *fabrique, const cadeau_t *cadeau);

bool produire(fabrique_t *fabrique, const cadeau_t *cadeau);

bool produire(fabrique_t fabriques[], size_t fabrique_count, const cadeau_t *cadeau);

void distribuer(fabrique_t *fabrique, communte_t *commune);
```

Question. Implémentez, en C, les fonctions correspondant aux fonctionnalités numéros **2 et 4** ci-dessus. Vous pouvez appeler les autres fonctions sans les implémenter.

Réponse.

```c
bool est_ideale(const frabrique_t *fabrique, const cadeau_t *cadeau) {
  for (cadeau_node_t *node = frabrique->cadeaux.first;
      node != NULL; node = node->next) {
    if (node->cadeau.destination == cadeau->destination) {
      return true;
    }
  }
  return false;
}

bool produire(fabrique_t fabriques[], size_t fabrique_count, const cadeau_t *cadeau){
  // D'abord, on tente de trouver une fabrique idéale
  for (size_t i = 0; i < fabrique_count; i++) {
    fabrique_t *f = &fabrique[i];
    if (peut_produire(f, cadeau) && est_ideale(f, cadeau)) {
      produire(f, cadeau);
      return true;
    }
  }

  // Sinon, on utilise n'importe quelle fabrique disponible
  for (size_t i = 0; i < fabrique_count; i++) {
    fabrique_t *f = &fabrique[i];
```

```
    if (peut_produire(f, cadeau)) {
      produire(f, cadeau);
      return true;
    }
  }

  // On n'a rien trouvé
  return false;
}
```

La duplication des deux boucles est dommage. Avec les outils dont nous disposons actuellement, il est cependant difficile de faire mieux. Une possibilité, mais qui sort de ce qui était attendu ici, est d'utiliser un pointeur séparé pour retenir l'adresse de la « meilleure » fabrique trouvée jusque là.

```
bool produire(fabrique_t fabriques[], size_t fabrique_count, const cadeau_t *cadeau){
  fabrique_t *selected = NULL;
  for (size_t i = 0; i < fabrique_count; i++) {
    fabrique_t *f = &fabrique[i];
    if (peut_produire(f, cadeau)) {
      /* Si on n'a encore sélectionné aucune fabrique, alors on
       * sélectionne celle-ci même si elle n'est pas idéale.
       * Par contre, si on a déjà sélectionné une fabrique, on
       * ne la remplace que si la nouvelle est idéale.
       */
      if (selected == NULL || est_ideale(f, cadeau)) {
        selected = f;
      }
    }
  }

  if (selected == NULL) {
    return false;
  } else {
    produire(selected, cadeau);
    return true;
  }
}
```

7. Divisibilité par 7

On souhaite tester si un nombre est divisible par 7. Cependant, on travaille sur une machine avec un processeur particulier. Ce processeur possède seulement une instruction pour diviser par 10, mais pas par d'autres nombres. Heureusement, on peut déterminer si un nombre entier (relatif) est divisible par 7 avec la formule suivante :

$$p(x) = \begin{cases} \text{VRAI} & \text{si } x = 0 \text{ ou } x = 7 \\ \text{FAUX} & \text{si } 0 < x < 10 \text{ et } x \neq 7 \\ p(-x) & \text{si } x < 0 \\ p(\lfloor x/10 \rfloor - 2 \cdot (x \bmod 10)) & \text{si } x \geq 10 \end{cases}$$

Question. Écrivez, en C, une fonction **récursive** `div7rec(x)` qui prend en paramètre un entier relatif `x`, et renvoie `true` s'il est divisible par 7, `false` sinon. Seules les instructions élémentaires sont autorisées (les fonctions mathématiques sont interdites). De plus, les divisions et modulos ne peuvent être faits que par 10. Donc `a / 10` et `a % 10` sont autorisées (avec `a` entier). Toute autre forme de division est interdite, puisqu'on ne pourrait pas la faire fonctionner sur notre machine.

Réponse. Pour la version récursive, on peut directement traduire la formulation mathématique en fonction C.

```
bool div7rec(int x) {
  if (x == 0 || x == 7) {
    return true;
  } else if (0 < x && x < 10) { // implied: x != 7
    return false;
  } else if (x < 0) {
    return p(-x);
  } else {
    return p((x / 10) - 2 * (x % 10));
  }
}
```

En réarrangeant un peu l'ordre des tests, on peut obtenir une version plus courte, bien que fondamentalement équivalente.

```
bool div7rec(int x) {
  if (x < 0) {
    return p(-x);
  } else if (x >= 10) {
    return p((x / 10) - 2 * (x % 10));
  } else {
    return (x == 0 || x == 7);
  }
}
```

Question. Écrivez, en C, une fonction **non récursive** `div7(x)` qui fait la même chose, avec les mêmes contraintes.

Réponse. La version non récursive est plus difficile. On met l'équivalent des appels récursifs dans la boucle. Les cas de fin, quand $0 \leq x < 10$, sont traités après la boucle.

```
bool div7(int x) {
  while (x < 0 || x >= 10) {
    if (x < 0) {
      x = -x;
    } else {
      x = (x / 10) - 2 * (x % 10);
```

```
    }
  }
  return (x == 0 || x == 7);
}
```

8. Routage IP

Rappelez-vous le principe des tables de routage IP vus en ICC-T. Nous allons implémenter certains aspects d'un programme qui s'exécuterait sur un routeur, afin de recevoir, traiter, et envoyer des paquets IP.

On suppose qu'il existe un type de données C `adresseip_t` qui représente une adresse IP (si cela vous aide, vous pouvez considérer qu'il s'agit d'un entier non signé).

Question. Définissez, en C, un type de données `table_routage_t` qui permet de stocker la table de routage d'un nœud. Expliquez brièvement vos choix d'implémentation.

Réponse.

```c
typedef struct ligne_routage {
  adresseip_t destination;
  adresseip_t voisinSuivant;
  unsigned int distance; // ou int
} ligne_routage_t;

typedef struct table_routage {
  size_t size;
  ligne_routage_t *lignes;
} table_routage_t;
```

Question. On suppose qu'il existe une fonction C

```c
void envoieSurLien(adresseip_t voisin, adresseip_t destination, const char *paquet);
```

que vous ne devez pas implémenter. Elle transfère au nœud voisin direct donné, via la couche de lien, un paquet à destination de destination. On représente le contenu d'un paquet comme une chaîne de caractères.

Implémentez, en C, la fonction `envoieSurReseau` qui effectue le transfert d'un paquet au niveau de la couche réseau. Elle reçoit la table de routage du nœud courant (telle que vous l'avez définie à la sous-question précédente), un destinaire et un paquet. Le nœud courant est le routeur sur lequel on exécute le programme.

Elle doit transférer le paquet au prochain nœud via la couche de lien. On suppose que le destinaire du paquet n'est pas le nœud courant.

Conseil : n'oubliez pas qu'il vous est toujours possible de définir des fonctions annexes. Dans ce cas, vous devez bien sûr fournir l'implémentation de celles-ci.

Réponse.

```c
void envoieSurReseau(const table_routage_t *table,
    adresseip_t destination, const char *paquet) {
  // Chercher la destination dans la table
  for (size_t i = 0; i < table->size; i++) {
    ligne_routage_t *ligne = &table->lignes[i];
    if (ligne->destination == destination) {
      // Quand on l'a trouvée, transférer le paquet au voisin suivant
      envoieSurLien(ligne->voisinSuivant, destination, paquet);
      // Optionnel : éviter de chercher dans le reste de la table
      return;
    }
  }
}
```

9. Nombres premiers

Écrivez un *programme* C (`main` et potentiellement des fonctions annexes) qui crée un fichier nommé `premiers.csv` contenant la liste, un par ligne, des nombres premiers et de leur carré, séparés par une virgule.

Le début du fichier `premiers.csv` sera donc :

```
2,4
3,9
5,25
7,49
```

Votre programme devra commencer par demander jusqu'à quel nombre maximum (pas nécessairement premier) on souhaite aller (inclus si c'est un nombre premier qui est donné). Par exemple :

```
Jusqu'à quel nombre souhaitez-vous aller ?
1234
```

Dans ce cas, la fin du fichier premiers.csv sera alors :

```
1229,1510441
1231,1515361
```

(car le prochain nombre premier après 1231 est 1237).

On supposera de plus qu'une fonction

```c
unsigned int prochain_premier(unsigned int n);
```

est fournie (vous n'avez donc pas à l'écrire). Cette fonction donne le prochain nombre premier strictement plus grand qu'un nombre `n`. Par exemple `prochain_premier(5)` retourne `7`, et prochain_premier(1229) retourne 1231.

(Rappel : le premier nombre premier est 2.)

Réponse. Voici une première version, qui n'obtient pas tous les points.

```c
int main() {
  printf("Jusqu'à quel nombre souhaitez-vous aller ?\n");
  unsigned int max = 0;
  scanf("%u", &max);

  FILE *f = fopen("premiers.csv", "wb");

  int p = 2;
  while (p <= max) {
    fprintf(f, "%u,%u\n", p, p*p);
    p = prochain_premier(p);
  }

  fclose(f);

  return 0;
}
```

Voici une version plus aboutie, qui gère correctement les erreurs liées à l'écrire dans le fichier.

```c
int main() {
  printf("Jusqu'à quel nombre souhaitez-vous aller ?\n");
  unsigned int max = 0;
  scanf("%u", &max);
```

```
  FILE *f = fopen("premiers.csv", "wb");
  if (f == NULL) {
    printf("Impossible de créer le fichier\n");
    return 1;
  }

  int p = 2;
  bool failure = false;
  while (p <= max && !failure) {
    if (fprintf(f, "%u,%u\n", p, p*p) < 0) {
      failure = true;
    }
    p = prochain_premier(p);
  }

  if (failure) {
    printf("Une erreur s'est produite pendant l'écriture du fichier\n");
    fclose(f);
    return 1;
  }

  fclose(f);
  return 0;
}
```