Devoir surveillé #1 (avec solutions)

Vendredi 2025-11-07 ; durée : trois heures

Petits exercices

Exercice 1. *Lecture de types*

Soit l'extrait de programme C suivant :

```c
struct intpair
{
    int a;
    int b;
};

struct intpair fun(int a, int b)
{
    int x;
    unsigned *p;
    int u[10];
    bool v;
    // ...
}
```

1. Donnez la signature de la fonction y apparaissant, et expliquez ce qu'elle signifie (seulement en tant que signature ; on ne cherche pas particulièrement à deviner ce que fait réellement la fonction).

   La signature est `struct intpair fun(int a, int b)` : cette fonction prend deux paramètres de type `int` et renvoie un résultat de type `struct intpair`

2. Pour chacune des variables apparaissant dans cette fonction, donnez son **type C** et expliquez brièvement ce en quoi il consiste.

   — `x` est de type `int` : un type entier signé de taille fixe.
   — `p` est de type `unsigned *` : le type des pointeurs vers `unsigned` (`p` contient l'adresse d'un entier non signé de taille fixe).
   — `u` est de type `int[10]` : le type des tableaux contenant dix `int`s.
   — `v` est de type `bool` : un type booléen pouvant prendre seulement deux valeurs `true` et `false`.

Exercice 2. *Exécution de programme*

Soit le code source C suivant :

```c
#include <stdlib.h>
#include <stdio.h>

void fun(unsigned n) {
    for (unsigned i = 0; i < n; i = i + 2) {
        printf("%d ", i);
    }
    printf("\n");
}

int main(void) {
    fun(10);
    return EXIT_SUCCESS;
}
```

1. Donnez une suite possible de commandes permettant de le compiler et d'exécuter le programme résultant depuis un terminal UNIX (avec un environnement similaire à ceux disponibles en TP). Pour cela, on supposera que le code source est contenu dans un fichier `pe.c` se trouvant dans le même répertoire que le répertoire de travail du terminal.

   De façon minimaliste (sans aucune option) :

   ```
   > cc pe.c
   > ./a.out
   ```

2. Sans justification, que produit l'exécution de ce programme ?

   Ce programme affiche `0 2 4 6 8` sur la sortie standard.

Exercice 3. *Trouvez les bugs !*

On constate à l'exécution que la boucle suivante ne termine pas :

```
1 for (unsigned i = n; i >= 0; i--) {
2     n = n + i;
3 }
```

1. Expliquez pourquoi.

   La variable de boucle `i` étant de type entier non signé, la condition de sortie `i >= 0` est *toujours* trivialement vraie. Puisque le corps de boucle ne provoque aucune sortie anticipée, on ne sort jamais de la boucle.

On se donne la fonction suivante, dont on souhaite qu'elle renvoie `true` si `x` est un élément de `a`, et `false` sinon :

```
1  bool search(size_t n, int a[n], int x) {
2      for (size_t i = 0; i < n; i++) {
3          if (a[i] == x) {
4              return true;
5          }
6          else {
7              return false;
8          }
9      }
10 }
```

2. Expliquez en quoi elle n'est pas correcte pour ses spécifications.

   La boucle `for` effectuera au plus une itération, puisque que la fonction toute entière terminera quelque soit le résultat du test à la ligne 3. Ainsi, `search` renverra par exemple `false` quand appelée avec des arguments `2`, `int a[2] = {0, 1}` et `1`, alors que ses spécifications imposent qu'elle devrait renvoyer `true`.

3. Proposez une correction (on ne demande pas de justifier que la fonction corrigée soit correcte).

   ```
   bool search(size_t n, int a[n], int x) {
       for (size_t i = 0; i < n; i++) {
           if (a[i] == x) {
               return true;
           }
       }
       return false;
   }
   ```

4. Quelle erreur est causée par l'extrait de code suivant :

   ```
   1 int *p = NULL;
   2 *p = 12;
   ```

La ligne 2 effectue un déréférencement de pointeur nul, qui est illégal.

5. Quelle erreur est causée par l'extrait de code suivant :

```
1 int a[10];
2 for (size_t i = 0; i <= 10; i++) {
3     a[i] = i;
4 }
```

La boucle `for` effectue 11 itérations, et la 11ème cherchera à accéder à la case `a[10]` du tableau `a`. Or celui-ci est de taille `10`, et cet accès est donc illégal.

Exercice 4. *Terminaison 1*

On se donne la fonction `fun` suivante :

```
1 unsigned fun(unsigned x) {
2     unsigned r = 1;
3     while (r < x) {
4         r = 2 * r;
5     }
6     return r;
7 }
```

On suppose dans un premier temps que le type `unsigned` permet de représenter l'ensemble des entiers naturels.

1. Prouvez la terminaison de cette fonction.

Il suffit de prouver la terminaison de l'unique boucle `while`. Celle-ci admet l'entier naturel `r` comme variant :
Majoration : Par la condition de boucle, `r` est majoré par `x`, qui reste constant au cours de l'exécution du programme.
Stricte croissance : `r` est initialisé à `1` à la ligne 2, et est multiplié par `2` à la ligne 4 : il croît strictement à chaque itération de la boucle.
La présence de ce variant garantit la terminaison de la boucle, et donc de la fonction.

On considère maintenant le « vrai » type `unsigned`.

2. Expliquez pourquoi la terminaison de la fonction `fun` n'est plus garantie.

Ce n'est plus vrai que `r` croît strictement, car les calculs avec `r` se font maintenant modulo `UINT_MAX + 1`. La boucle n'admet plus de variant (qui soit évident), et la terminaison n'est pas garantie.

Pour les questions suivantes on pourra admettre l'invariant (global) suivant : la valeur de `r` est une puissance de deux ou nulle.

3. Donnez un exemple d'argument pour lequel `fun` ne termine pas (on ne demande pas nécessairement une valeur numérique exacte ; une description suffisamment précise suffit).

On a que la valeur maximale représentable par le type `unsigned` `UINT_MAX` n'est pas une puissance de deux. La fonction ne terminera pas si `x` est strictement supérieur à la plus grande puissance de deux représentable par le type `unsigned` (ce qui est donc possible par ce qui précède ; dans ce cas on a également `x` strictement positif). En effet, par l'invariant ci-dessus, on ne pourra dans ce cas jamais avoir `r >= x`, et la boucle ne terminera jamais.

4. Proposez une *précondition* sur l'argument de cette fonction qui, quand elle est satisfaite, garantit sa terminaison. On cherchera à avoir une précondition *la moins restrictive possible*, mais l'on pourra se contenter d'une argumentation sommaire.

Il est suffisant d'imposer que `x` doit être au plus égal à la plus grande puissance de deux représentable par un entier `unsigned`. Soit `M` cette valeur ceci revient à avoir la condition de sortie de boucle `(r < x && r < M)`, ce qui fait que la boucle admet `r` comme variant sur les `unsigned`.

Exercice 5. *Terminaison 2*

On se donne la fonction `fun` suivante :

```
1  unsigned fun(unsigned a, unsigned b) {
2      unsigned r = 0;
3      while (a > 0 || b > 0) {
4          if (a > b) {
5              a = a - 1;
6          }
7          else {
8              b = b - 1;
9          }
10         r = r + 1;
11     }
12     return r;
13 }
```

1. Prouvez sa terminaison.

   Il suffit de prouver la terminaison de l'unique boucle `while`, ce que l'on fait en utilisant le variant `a + b`.
   Minoration : Par la condition d'entrée de boucle, `a + b` doit être strictement positif (`a` et `b` ne peuvent pas être négatifs de par leur type, et l'on entre dans la boucle seulement si au moins l'une de ces deux variables est strictement positive), et est donc minoré par une constante.
   Stricte décroissance : Lors d'une itération de la boucle, quelque soit le résultat du test de la ligne 4 on a que `a` décroît de `1` (ligne 5 ; dans ce cas $a > b \geq 0$ et la soustraction de 1 ne provoque pas de dépassement de capacité) ou `b` décroît de `1` (ligne 8 ; dans ce cas $b \geq a$, et par la condition d'entrée de boucle au moins l'une des deux variables est > 0, et donc les deux le sont ; il n'y a à nouveau pas de problème de dépassement de capacité). Ainsi, `a + b` décroît strictement à chaque itération.
   La présence de ce variant garantit la terminaison de la boucle, et donc de la fonction.

Exercice 6. *Popcount*

Soit un entier non signé `x` de type `uint32_t`, on définit son *population count* comme le nombre de bits à 1 dans son écriture en base deux.

1. Écrivez une fonction C de signature :
   `uint32_t popcount(uint32_t x)`
   qui calcule et renvoie le *population count* de son argument.

   On propose :
   ```
   uint32_t popcount(uint32_t x)
   {
       uint32_t r = 0;
       for (; x > 0; x = x / 2)
       {
           if (x % 2 == 1)
           {
               r = r + 1;
           }
       }
       return r;
   }
   ```

2. Proposez un jeu de 3 tests pour cette fonction.

   On peut par exemple vérifier que `popcount` renvoie respectivement `0`, `2` et `4` pour les arguments `0`, `0xA` et `0xAC`.

Exercice 7. *Somme de tableaux*

1. Écrivez une fonction `sum` de signature :
   `unsigned *sum(size_t n, unsigned a[n], unsigned b[n])`

qui alloue une zone mémoire de durée de stockage « alloué » permettant de stocker `n` entiers `unsigned` consécutifs (on pourra supposer `n` non nul), l'initialise de façon à ce que le $i$ème de ces entiers soit égal à la somme des éléments d'indice i de `a` et `b`, et renvoie l'adresse du début de la zone.
On pourra considérer que l'allocation mémoire réussit toujours.

On propose :

```
unsigned *sum(size_t n, unsigned a[n], unsigned b[n])
{
    assert(n > 0);
    unsigned *r = malloc(n * sizeof(unsigned));
    for (size_t i = 0; i < n; i++)
    {
        r[i] = a[i] + b[i];
    }
    return r;
}
```

Problèmes

Exercice 8. *Optimalité de la recherche dichotomique*

Le but de cet exercice est de prouver dans un certain sens que la recherche par dichotomie d'un élément dans un tableau trié est optimale (la plus efficace possible), à constantes près.
On s'intéresse donc à des algorithmes résolvant le problème suivant : soit en entrée un tableau a trié de n entiers et un entier x, on veut renvoyer :

— un entier positif $i \in [\![0, n-1]\!]$ tel que $a[i] = x$ si un tel entier i existe
— -1 sinon

1. Expliquez le principe d'une recherche par dichotomie, et comment elle permet de résoudre le problème ci-dessus.

Une recherche dichotomique exploite le fait que le tableau est trié : si l'on trouve un indice j t.q. par exemple $a[j] < x$, alors on sait que pour tout $j' < j$, $a[j'] < x$ et il est inutile de tester ces autres éléments de a. On peut alors initialement comparer x à l'élément au milieu du tableau a et en fonction de si celui-ci est égal, strictement inférieur ou strictement supérieur à x, s'arrêter en renvoyant l'indice correspondant, ou continuer en comparant x au milieu de la moitié haute du tableau, ou au milieu de la moitié basse. On continue ainsi jusqu'à trouver un élément égal à x ou déduire qu'il n'y en a aucun dans a.

2. Justifiez brièvement que le coût d'une recherche par dichotomie est un $O(\log n)$ dans le pire cas.

À chaque étape d'une telle recherche, la longueur du tableau dans lequel peut se trouver x s'il appartient à a est divisée par (environ) 2. Ainsi, après au plus environ $\log n$ étapes (si l'on a pas encore trouvé x), ce tableau est de longueur au plus 1 ce qui permet de conclure en une étape. Chaque étape pouvant être réalisée pour un coût constant, le coût total est un $O(\log n)$ dans le pire cas (qui correspond à celui où x n'appartient pas à a).

On considère maintenant des algorithmes (par exemple écrits en C) qui ne peuvent pas directement accéder aux éléments de a, et ne peuvent apprendre de l'information à leur sujet qu'à travers une fonction de signature :
`int cmp(size_t i, int x)`
qui renvoie respectivement `-1`, `0` ou `1` si `a[i]` est strictement inférieur, égal ou strictement supérieur à `x`. On interdit également à l'algorithme de faire appel à toute autre fonction. Un algorithme résolvant le problème de recherche a alors par exemple une signature :
`int search(size_t n, int x)`
(où l'on suppose `n` représentable par un `int`).
On donne ci-dessous un exemple d'un tel algorithme résolvant le problème par une recherche linéaire :

```
int search(size_t n, int x) {
    for (size_t i = 0; i < n; i++) {
        if (cmp(i, x) == 0) {
```

```
            return i;
        }
    }
    return -1;
}
```

Le but va maintenant être de montrer que tout algorithme résolvant correctement le problème doit effectuer au moins K appels à la fonction `cmp` dans le pire cas, avec $K = \Omega(\log n)$ (autrement dit, K est asymptotiquement supérieur ou égal à $\log n$, à constantes près) ; ceci permettra immédiatement de conclure que le coût pire cas de l'algorithme lui même est un $\Omega(\log n)$.
Dans tout ce qui suit, on note $\mathbb{A}$ un algorithme résolvant (totalement) correctement le problème de recherche, et n et x respectivement la longueur du tableau dans laquelle la recherche s'effectue et l'élément recherché.

3. Montrez que pour toute valeur x fixée, $\mathbb{A}$ peut renvoyer exactement $n+1$ réponses différentes, en considérant l'ensemble des tableaux de n entiers (par exemple de type `int`) sur lesquels il peut s'exécuter.

> Pour toute valeur de x, il existe des tableaux de n éléments où x n'est pas présent (dans ce cas $\mathbb{A}$ doit terminer et renvoyer `-1`) et des tableaux où x est présent en un unique indice $i \in [\![0, n-1]\!]$ (dans ce cas $\mathbb{A}$ doit terminer et renvoyer i, qui peut prendre n valeurs distinctes possibles). $\mathbb{A}$ peut donc renvoyer exactement $n+1$ valeurs distinctes.

Soit une exécution (qui termine) de $\mathbb{A}$, on appelle *transcript* la suite finie des valeurs renvoyées par les appels successifs à `cmp` (qui dépend en principe de n, x, et les valeurs contenues dans le tableau).

4. Montrez qu'on peut majorer le nombre de transcripts distincts de longueur positive ou nulle inférieure ou égale à c par 3^{c+1}.

> Il y a 3^i transcripts de longueur i, et donc $\sum_{i=0}^{c} 3^i$ transcripts de longueur $\leq c$, soit $\frac{3^{c+1}-1}{2} < 3^{c+1}$ transcripts distincts.

On admet maintenant que pour des valeurs de n et x fixées, la réponse renvoyée par $\mathbb{A}$ dépend *uniquement* du transcript de son exécution (c'est à dire, des valeurs contenues dans le tableau). (On peut justifier cela par le fait qu'une fois les arguments de $\mathbb{A}$ fixés, la seule variabilité dans son exécution peut provenir des valeurs renvoyées par les fonctions appelées par $\mathbb{A}$, et que ces appels ne peuvent être qu'à `cmp`.)

5. Conclure.

> On fixe n et x à des valeurs quelconques. On a supposé $\mathbb{A}$ totalement correct, et il termine donc quelques soient les valeurs contenues dans le tableau. Soit K le nombre maximum d'appels à `cmp` effectués par $\mathbb{A}$ lors d'une exécution, par Q. 4 on a que $\mathbb{A}$ admet au plus 3^{K+1} transcripts possibles.
> Par Q. 3, $\mathbb{A}$ doit pouvoir renvoyer $n+1$ valeurs distinctes (en fonction des valeurs contenues dans le tableau).
> Or par l'hypothèse ci-dessus, pour pouvoir renvoyer deux valeurs distinctes, $\mathbb{A}$ doit avoir produit des transcripts différents ; il est donc nécessaire que $\mathbb{A}$ puisse produire au moins $n+1$ transcripts distincts.
> Finalement, on doit donc avoir $n+1 \leq 3^{K+1}$, soit $\log_3(n+1) \leq K+1$, et K est bien un $\Omega(\log n)$.

Exercice 9. *Implémentation jouet de la fonction de hachage de mot de passe* `scrypt`

Dans tout cet exercice, on suppose disposer d'une fonction `die` qui prend en argument une chaîne de caractère, affiche celle-ci sur la sortie standard, et termine l'exécution du programme avec un code d'erreur. Une implémentation possible de cette fonction est :

```
void die(char *msg) {
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}
```

et un exemple d'utilisation :

```
die("erreur à l'exécution");
```

Le but de cet exercice est d'implémenter une version jouet de la *fonction de hachage de mot de passe* `scrypt`. On définit celle-ci de la façon suivante (sans pour l'instant préciser les types exacts des objets manipulés) :
soit H une *fonction de hachage* prenant pour argument un mot de passe m et une clef k et renvoyant un entier de 64 bits, et r un entier strictement positif (potentiellement grand) inférieur à 2^{64}, on définit :

— $h_0 = H(k, m)$, puis $h_i = H(k, h_{i-1})$ pour $i \in [\![1, r-1]\!]$.
— $b_0 = H(k, h_{r-1})$, puis $b_i = H(k, b_{i-1} \boxplus h_{b_{i-1} \bmod r}$ pour $i \in [\![1, r-1]\!]$, (où « $\boxplus$ » désigne l'addition « modulo 2^{64}, et « $a \mod b$ » le reste positif de la division de a par b, tous deux positifs).
— La valeur de `scrypt` pour ces arguments et cette fonction de hachage est définie comme l'entier de 64 bits représentant b_{r-1}.

Pour la suite, on suppose les mots de passe représentés comme des tableaux d'entiers `uint64_t`, et la fonction de hachage H implémentée par une fonction de signature :
`uint64_t hash(uint64_t k, size_t n, const uint64_t m[n])`
Le qualifieur de type `const` indique le fait que cette fonction ne modifie pas les éléments de son argument `m` (ce que l'on pourra donc supposer chaque fois que l'on utilise la fonction).

1. Écrivez un extrait de code C qui appelle `hash` avec un argument `k` quelconque et comme argument `m` un tableau `a` (supposé déjà déclaré) de longueur `100`.

```
hash(0, 100, a);
```

2. Expliquez ce que fait l'extrait de code suivant :
```
uint64_t a = 123456;
uint64_t h = hash(0, 1, &a);
```

Cet extrait de code appelle la fonction `hash` (et stocke son résultat dans une variable `h`) avec une clef `k` valant `0`, sur le mot de passe stocké dans la variable `a`. Comme cette variable est de type `uint64_t`, l'appel se fait en fournissant son adresse et une longueur de `1` : `&a` peut s'assimiler à un tableau d'`uint64_t` de longueur `1`.

3. Écrivez une fonction `scrypt_1` de signature :
`uint64_t scrypt_1(uint64_t k, size_t n, uint64_t m[n], uint64_t r)`
qui calcule et renvoie le résultat de `scrypt` telle que définie ci-dessus, où `k` correspond à k, `m` (de longueur `n`) à m, et `r` à r.
Cette fonction ne devra calculer chaque valeur h_i qu'une unique fois au cours de son exécution, et pourra allouer de la mémoire avec `malloc`.
Prenez garde à correctement traiter les éventuelles erreurs pouvant avoir lieu à l'exécution, et à éviter toute fuite mémoire.

On propose :
```
uint64_t scrypt_1(uint64_t k, size_t n, uint64_t m[n], uint64_t r)
{
    assert(r > 0);

    uint64_t *h = malloc(r * sizeof(uint64_t));
    if (h == NULL)
    {
        die("script_1: not enough memory");
    }

    h[0] = hash(k, n, m);
    for (uint64_t i = 1; i < r; i++)
    {
        h[i] = hash(k, 1, &h[i-1]);
    }

    uint64_t b = hash(k, 1, &h[r-1]);
    for (uint64_t i = 1; i < r; i++)
    {
        uint64_t t = b + h[b % r];
        b = hash(k, 1, &t);
    }

    free(h);
    return b;
}
```

4. Écrivez une fonction `scrypt_2` de signature :
`uint64_t scrypt_2(uint64_t k, size_t n, uint64_t m[n], uint64_t r)`
qui calcule et renvoie le résultat de `scrypt` telle que définie ci-dessus, où `k` correspond à k, `m` (de longueur `n`) à m, et `r` à r.
Cette fonction ne devra pas utiliser `malloc`, mais pourra calculer plusieurs fois chaque valeur h_i au cours de son exécution.

On propose :

```
1  uint64_t scrypt_2(uint64_t k, size_t n, uint64_t m[n], uint64_t r)
2  {
3      assert(r > 0);
4
5      uint64_t b = hash(k, n, m);
6      for (uint64_t i = 1; i <= r; i++)
7      {
8          b = hash(k, 1, &b);
9      }
10
11     for (uint64_t i = 1; i < r; i++)
12     {
13         uint64_t b2 = hash(k, n, m);
14         for (uint64_t i = 1; i <= b % r; i++)
15         {
16             b2 = hash(k, 1, &b2);
17         }
18         uint64_t t = b + b2;
19         b = hash(k, 1, &t);
20     }
21
22     return b;
23 }
```

5. Pour chacune de vos fonctions `scrypt_1` et `scrypt_2`, donnez (asymptotiquement et à constante près, exprimé sous forme de O) :

— le nombre F d'appels à `hash` ;
— le nombre M d'accès à de la mémoire allouée par `malloc` (en ignorant les éventuels accès mémoire fait par d'autres fonctions, comme `hash`) ;

en fonction de r.

Pour `scrypt_1` : On ignore les appels & accès hors des boucles ; chaque itération des deux boucles effectue un nombre constant non nul d'accès mémoire à `h` (qui a été alloué avec `malloc`) et un appel à `hash`. Chacune des boucles effectuant $O(r)$ itérations, on a :

— $F = O(r)$
— $M = O(r)$

Pour `scrypt_2` : On ignore les appels & accès hors des boucles. La fonction n'alloue pas de mémoire avec `malloc`, et l'on a trivialement $M = 0 = O(1)$.
La boucle de la ligne 6 effectue $O(r)$ appels à `hash` au total ($O(1)$ pour chacune des $O(r)$ itérations). Pour la même raison, celle de la ligne 11 effectue $O(r)$ appels en dehors de la boucle interne de la ligne 14 ; le nombre d'itérations de cette dernière boucle étant *a priori* variable à chaque itération de la boucle externe, on ne peut pas conclure grand chose à part le majorer par r (puisqu'il est majoré par $b \mod r < r$) ; c'est donc un $O(r)$. Au total, la boucle de la ligne 11 effectue donc $O(r^2)$ appels à `hash` ($O(r)$ pour chacune des $O(r)$ itérations), et l'on a $F = O(r^2)$.

6. Sans l'implémenter, proposez un *compromis temps-mémoire* qui permette d'implémenter `scrypt` de façon à avoir M (défini comme à la question précédente) pouvant prendre n'importe quelle valeur strictement positive, et qui est telle que $FM = O(r^2)$ (avec F également défini comme à la question précédente).

Une solution possible consiste à allouer un tableau h de longueur $M > 0$ où l'on stocke les valeurs h_i pour $i \in \mathcal{M} := \{\lceil r/M \rceil, 2 \times \lceil r/M \rceil, \ldots, (M-1) \times \lceil r/M \rceil\}$. On peut alors modifier `scrypt_2` à la ligne 13 en initialisant

`b2` à h_j, avec $j = \max_{k \in \mathcal{M}}\{k \leq b_{i-1} \mod r\}$, et la boucle de la ligne 14 de façon à ce qu'elle ne fasse plus que $\lceil r/M \rceil$ itérations dans le pire cas. Au total, le nombre d'appels F à `hash` devient alors un $O(r \times \lceil r/M \rceil)$, et l'on a bien $FM = O(r^2)$.