EPITA S3 promo 2018
Programmation - Épreuve machine

Marwan Burelle *

Vendredi 14 novembre 2014

Instructions :

Vous devez lire l'intégralité du sujet, ainsi que l'ensemble de ces instructions. Tout ce qui est explicitement demandé dans ce sujet est obligatoire. La perte de points due au non respect des consignes explicites ne sera pas contestable, en particulier pour les problèmes de compilation de votre code ou l'existence de sous-répertoire dans votre rendu.

Votre répertoire pendant l'épreuve est temporaire, dans ce répertoire vous trouverez un répertoire `subject` *et un répertoire* `rendu`*. Dans le répertoire* `subject` *vous trouverez un sous-répertoire* `Skel` *vous devez copier le* **contenu** *de ce répertoire dans votre répertoire de rendu (*`rendu`*.)*

Vous devez effectuer des rendus réguliers pour être sûr de ne pas perdre votre travail. Pour effectuer un rendu, il vous suffit d'appeler la commande `rendu`.

Pour information, la copie des fichiers du répertoire d'origine vers votre répertoire de rendu, ce fait à l'aide des commandes :

```
> cd
> cd subject
> cp Skel/* ~/rendu/
```

Dans ce répertoire vous trouverez : un fichier `Makefile` *permettant de compiler votre code, un certain nombre de fichiers annexes, un fichier pour chaque question de la forme* `questionXX.c`*. Seuls les fichiers de question peuvent être modifiés, les autres seront remplacés lors de la correction par les originaux.*

Le `Makefile` *permet d'engendrer un petit programme de test pour chaque question. Le programme de test se charge des interactions avec l'utilisateur et appelle votre fonction avec les paramètres attendus. Pour obtenir le programme de test de chaque question il faut faire la commande :* `make questionXX`*.*

La compilation lors de la correction utilisera ce `Makefile`*, par conséquent vous n'aurez les points à la question XX que si la commande "*`make questionXX`*" réussit* ***et bien sûr que le résultat est correct****.*

Le barème est donné à titre indicatif et peut être sujet à modifications.

À la fin de ce document vous trouverez des annexes décrivants :

- *Quelques consignes sur les programmes de tests ;*
- *La liste des fichiers à rendre.*

Il y a 25 points et 12 questions dans cet examen.

*marwan.burelle@lse.epita.fr

C : bases

Question 1 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long power(unsigned long a, unsigned long b);
```

`power(a,b)` renvoie l'entier a^b.

Exemple 1.1:

```
shell> ./question01 0 5
Fixed tests:
  power( 2,  0) =                    1 (expected 1)
  power( 2,  1) =                    2 (expected 2)
  power( 2,  2) =                    4 (expected 4)
  power( 2,  3) =                    8 (expected 8)
  power( 2,  4) =                   16 (expected 16)
Random tests:
  power( 9,  6) =               531441
  power( 3, 15) =             14348907
  power( 3, 15) =             14348907
  power( 4, 12) =             16777216
  power( 3,  1) =                    3
```

Question 2 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long fibo(unsigned long n);
```

`fibo(n)` renvoie le terme de rang n de la suite de Fibonacci.

On rappelle la définition de la suite de Fibonacci :

$$\texttt{fibo}(n) = \begin{cases} 0 & \text{when } n = 0 \\ 1 & \text{when } n = 1 \\ \texttt{fibo}(n-1) + \texttt{fibo}(n-2) & \end{cases}$$

Exemple 2.1:

```
shell> ./question02 10
Fixed tests:
fibo(0) = 0
fibo(1) = 1
fibo(2) = 1
fibo(3) = 2
fibo(4) = 3
fibo(5) = 5
```

```
fibo(6) = 8
fibo(7) = 13
fibo(8) = 21
fibo(9) = 34
```

Question 3 (1)

Écrire la(les) fonction(s) suivante(s):

```
unsigned long fact(unsigned long n);
```

`fact(n)` renvoie l'entier $!n$.

$$\texttt{fact}(n) = \begin{cases} 1 & \text{when } n = 0 \\ n * \texttt{fact}(n-1) \end{cases}$$

Exemple 3.1:

```
shell> ./question03 0 5
fact(13) = 6227020800
fact(01) = 1
fact(12) = 479001600
fact(10) = 3628800
fact(08) = 40320
```

Question 4 (2)

Écrire la(les) fonction(s) suivante(s):

```
unsigned hamming(unsigned a, unsigned b);
```

`hamming(a, b)` calcule la distance de Hamming entre a et b. La distance de Hamming est définie comme le nombre de bits différents entre a et b. Par exemple, entre l'entier 4 (100 en binaire) et l'entier 5 (101 en binaire) la distance de Hamming est de 1.

On rappelle que l'on peut tester le dernier bit (poid faible) d'un entier en utilisant le modulo (`a % 2`) et que diviser un nombre par deux supprime ce dernier bit.

Enfin, il peut être utile de considérer l'opération de *ou-exclusif* bit à bit entre deux entiers : `a ^ b`. Dans le résultat de cette opération, tous les bits qui était différent entre a et b seront à 1 et les autres seront à 0. La distance de hamming se résume donc à compter le nombre de bits à 1 dans `a ^ b`.

Exemple 4.1:

```
shell> ./question04 0 5
Fixed Tests:
a = 00000000000000000000000000000000
```

```
b = 11111111111111111111111111111111
hamming(a, b) = 32
Random Tests:
a = 01101011100010110100010101100111
b = 00110010011110110010001111000110
hamming(a, b) = 15
a = 01100100001111001001100001101001
b = 01100110001100110100100001110011
hamming(a, b) = 11
a = 01110100101100001101110001010001
b = 00011001010010010101110011111111
hamming(a, b) = 17
a = 00101010111010001001010001001010
b = 01100010010101010101100011101100
hamming(a, b) = 16
a = 00100011100011100001111100101001
b = 01000110111010000111110011001101
hamming(a, b) = 16
```

Question 5 (3)

Écrire la(les) fonction(s) suivante(s):

```
unsigned int_sqrt(unsigned x);
```

`int_sqrt(x)` calcule l'approximation entière de la racine carrée de `x`. La racine entière (notée $\lfloor \sqrt{x} \rfloor$) d'un entier est la solution entière à l'équation suivante :

$$\lfloor \sqrt{x} \rfloor^2 \leq x < (\lfloor \sqrt{x} \rfloor + 1)^2$$

On peut la calculer en utilisant la méthode de Héron (variation de la méthode de Newton) : à partir d'une solution *temporaire* y on calcule la prochaine approximation y' grace à la moyenne (entière) :

$$y' = (y + x/y)/2$$

L'algorithme s'arrête lorsque $y \leq y'$. On doit choisir une approximation initiale *haute* (c'est à dire supérieure à la racine cherchée, dans ce cas x est suffisant.)

Exemple 5.1:

```
shell> ./question05 0 5
int_sqrt(1804289385) = 42476 (OK)
int_sqrt(846930888) = 29102 (OK)
int_sqrt(1681692779) = 41008 (OK)
int_sqrt(1714636917) = 41408 (OK)
int_sqrt(1957747795) = 44246 (OK)
```

C : Strings

Question 6 (2)

Écrire la(les) fonction(s) suivante(s):

```
void to_lower(char *s);
```

`to_lower(s)` passe en minuscule toutes les lettres de la chaîne `s`. Le pointeur `s` ne sera pas `NULL`. Bien évidement, votre fonction doit laisser inchanger les caractères qui ne sont pas des lettres (ou sont des lettres déjà en minuscule.) Il peut être opportun de regarder la page de manuel `ascii(7)`.

On notera que le programme de test vérifie que vous n'avez pas dépassé la fin de la chaîne de caractères (le caractère de code ASCII 0.)

Exemple 6.1:

```
shell> ./question06 1 10
FIXED TESTS:
Original:
  "A FULL UPPER-CASE STRING."
After to lower:
  "a full upper-case string."
Off-bound check: OK OK
RANDOM TESTS:
Original:
  "\VA/vK~|%r"
After to lower:
  "\va/vk~|%r"
Off-bound check: OK OK
```

Question 7 (1)

Écrire la(les) fonction(s) suivante(s):

```
size_t mystrlen(char *s);
```

`mystrlen(s)` renvoie le nombre de caractères de la chaîne `s` (le pointeur ne sera pas `NULL`.) Vous devrez respecter le comportement attendu pour la fonction `strlen(3)`.

Exemple 7.1:

```
shell> ./question07 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: OK
shell> ./question07 0 0
s = ""
mystrlen(s) = 0 -- check: OK
```

Question 8 (2)

Écrire la(les) fonction(s) suivante(s):

```
char *mystrncpy(char *dst, char *src, size_t len);
```

`mystrncpy(dst,src,len)` : copie au plus `len` caractères de la chaîne `src` dans la chaîne `dst`. On suppose que `src` et `dst` sont non NULL, `src` est terminée par un `'\0'` et `dst` est de taille suffisante.

Dans tous les cas, `mystrncpy(dst,src,len)` écrit exactement `len` caractères dans `dst`. Si le nombre de caractères de `src` est inférieur à `len`, alors votre fonction devra remplir la fin de `dst` par des `'\0'`. Sinon (`src` plus grand que `len`) votre fonction ne doit pas mettre de `'\0'` à la fin de `dst` (voir `strncpy(3)`.)

Je vous conseille fortement de lire la page de manuel de la fonction `strncpy(3)` qui fournit une description complète de cette fonction.

Exemple 8.1:

```
shell> ./question08 0 5
src = "n{6\P"

test: mystrncpy(dst,src,6)
dst = "n{6\P"
-- check:
  first char: OK
  last char: OK
  0 fill: OK
  overflow: OK

test: mystrncpy(dst,src,2)
dst = "n{"
-- check:
  first char: OK
  last char: OK
  overflow: OK

test: mystrncpy(dst,src,10)
dst = "n{6\P"
-- check:
  first char: OK
  last char: OK
  0 fill: OK
  overflow: OK

test: mystrncpy(dst,src,0)
-- check:
  overflow: OK
```

C : Listes chaînées

Question 9 (2)

Écrire la(les) fonction(s) suivante(s):

```
struct s_list *duplicate_list(struct s_list *l);
```

`duplicate_list(l)` renvoie une **nouvelle** liste contenant les mêmes éléments que `l` dans le même ordre.

Les listes sont définies de la manière suivantes :

```
struct s_list {
  struct s_list        *next;
  int                   val;
};
```

Attention, vous devez renvoyer une nouvelle liste. Lors des tests, la liste originale sera supprimée.

Exemple 9.1:

```
shell> ./question09 5
l =
  00 -> 01 -> 02 -> 03 -> 04
Cloning list ... rl = duplicate_list(l)
Delete (free) l ...
rl =
  00 -> 01 -> 02 -> 03 -> 04
```

Question 10 (4)

Écrire la(les) fonction(s) suivante(s):

```
struct list* merge(struct list *l1, struct list *l2);
```

`merge(l1, l2)` renvoie une liste composée des éléments de `l1` et `l2` dans l'ordre de croissant. Les deux listes originales sont triées dans l'ordre croissant.

Votre fonction **ne** devra **pas** créer de nouvelle liste, mais réutiliser les cellules dans des listes d'origine en ne modifiant que les pointeurs `next`. Vous n'aurez donc **jamais** besoin de `malloc` ou de `free`.

Exemple 10.1:

```
shell> ./question10 5
l1 =
  01 -> 03 -> 05 -> 07 -> 09
l2 =
  00 -> 02 -> 04 -> 06 -> 08
```

```
l = merge(l1, l2)
l =
  00 -> 01 -> 02 -> 03 -> 04 -> 05 -> 06 -> 07 -> 08 -> 09
```

C : Tableaux

Question 11 (3)

Écrire la(les) fonction(s) suivante(s):

```
void select_sort(int tab[], size_t len);
```

`select_sort(tab, len)` trie le tableau `tab` de longueur `len` en utilisant un tri par sélection. On rappelle l'algorithme de tri par sélection :

```
select_sort(tab, len):
  for i <- 0 to len - 1 do
    min <- i
    for j <- i + 1 to len - 1 do
      if tab[j] < tab[min] then min <- j
    done
    tab[i] <-> tab[min]
  done
```

Exemple 11.1:

```
shell> ./question11 0 5
Before sort:
tab = | 83 | 86 | 77 | 15 | 93 |
After sort:
tab = | 15 | 77 | 83 | 86 | 93 |
```

Question 12 (3)

Écrire la(les) fonction(s) suivante(s):

```
void hist(char *str, size_t len, size_t counts[]);
```

`hist(str,len,counts)` compte le nombre d'occurrences de chaque carractère différents dans `str` et stocke le résultat dans le tableau `counts`. Le tableau `counts` est déjà alloué et contient 256 cases, par contre, ces cases n'ont pas été initialisées à 0. `str` contient exactement `len` caractères, ce n'est pas une chaîne de caractères (vous devez lire exactement `len` caractères, même si vous rencontrez un caractères `'\0'`.

Pour des raisons pratiques, seuls les caractères sur 7bits seront testés (de 0 à 127) mais le tableau `counts` fait bien 256 cases et les cases non-utilisées doivent être à zéro.

Exemple 12.1:

```
shell> ./question12 0 40
s = "n{6\Pavw[:mO4=ZvMD^bvU;e'Rrs^4LJ1_$^A{S|"
hist(s, 40, counts)
| 0x00   :      | 0x20   :      | 0x40 @ :      | 0x60 ‘ :      |
| 0x01   :      | 0x21 ! :      | 0x41 A :    1 | 0x61 a :    1 |
| 0x02   :      | 0x22 " :      | 0x42 B :      | 0x62 b :    1 |
| 0x03   :      | 0x23 # :      | 0x43 C :      | 0x63 c :      |
| 0x04   :      | 0x24 $ :    1 | 0x44 D :    1 | 0x64 d :      |
| 0x05   :      | 0x25 % :      | 0x45 E :      | 0x65 e :    1 |
| 0x06   :      | 0x26 & :      | 0x46 F :      | 0x66 f :      |
| 0x07   :      | 0x27 ' :    1 | 0x47 G :      | 0x67 g :      |
| 0x08   :      | 0x28 ( :      | 0x48 H :      | 0x68 h :      |
| 0x09   :      | 0x29 ) :      | 0x49 I :      | 0x69 i :      |
| 0x0a   :      | 0x2a * :      | 0x4a J :    1 | 0x6a j :      |
| 0x0b   :      | 0x2b + :      | 0x4b K :      | 0x6b k :      |
| 0x0c   :      | 0x2c , :      | 0x4c L :    1 | 0x6c l :      |
| 0x0d   :      | 0x2d - :      | 0x4d M :    1 | 0x6d m :    1 |
| 0x0e   :      | 0x2e . :      | 0x4e N :      | 0x6e n :    1 |
| 0x0f   :      | 0x2f / :      | 0x4f O :    1 | 0x6f o :      |
| 0x10   :      | 0x30 0 :      | 0x50 P :    1 | 0x70 p :      |
| 0x11   :      | 0x31 1 :    1 | 0x51 Q :      | 0x71 q :      |
| 0x12   :      | 0x32 2 :      | 0x52 R :    1 | 0x72 r :    1 |
| 0x13   :      | 0x33 3 :      | 0x53 S :    1 | 0x73 s :    1 |
| 0x14   :      | 0x34 4 :    2 | 0x54 T :      | 0x74 t :      |
| 0x15   :      | 0x35 5 :      | 0x55 U :    1 | 0x75 u :      |
| 0x16   :      | 0x36 6 :    1 | 0x56 V :      | 0x76 v :    3 |
| 0x17   :      | 0x37 7 :      | 0x57 W :      | 0x77 w :    1 |
| 0x18   :      | 0x38 8 :      | 0x58 X :      | 0x78 x :      |
| 0x19   :      | 0x39 9 :      | 0x59 Y :      | 0x79 y :      |
| 0x1a   :      | 0x3a : :    1 | 0x5a Z :    1 | 0x7a z :      |
| 0x1b   :      | 0x3b ; :    1 | 0x5b [ :    1 | 0x7b { :    2 |
| 0x1c   :      | 0x3c < :      | 0x5c \ :    1 | 0x7c | :    1 |
| 0x1d   :      | 0x3d = :    1 | 0x5d ] :      | 0x7d } :      |
| 0x1e   :      | 0x3e > :      | 0x5e ^ :    3 | 0x7e ~ :      |
| 0x1f   :      | 0x3f ? :      | 0x5f _ :    1 | 0x7f   :      |
Total count: 40
```

Remarques sur la session exam

À la fin de l'épreuve, vous devez quitter votre session en fermant l'horloge (et uniquement en fermant celle-ci.) Dans tous les cas, votre session sera également fermée à la fin du temps imparti.

Lorsque l'horloge est fermée (volontairement, ou en fin de session), la console qui vous a demandé le token vous demande votre mot de passe. **Vous devez rentrer votre mot de passe UNIX pour que le rendu ai bien lieu.**

Vous pouvez aussi rendre sans quitter : il y a une commande (dans le shell) appellée `rendu` qui déclenche un rendu comme si vous quittiez (demande de mot de passe et envoie du rendu.)

Après la fin du test (dans les minutes qui suivent) vous pouvez redémarrer votre machine et vous connecter pour re-rendre (si par exemple il y a eu une erreur pendant votre premier rendu.)

Remarques sur les squelettes de questions

Pour chaque question, un embryon de code vous est fourni. Ce code correspond au strict minimum pour que la question compile et que l'appel au programme de test échoue avec une erreur identifiable. Par conséquent, vous devez supprimer du corps des fonctions à écrire, le code provoquant l'erreur, sous peine de voir votre programme échouer lors des tests (ne laissez surtout pas la ligne `REMOVE_ME(...)`.)

Exemple 1:
À titre d'exemple, si l'on vous demande la fonction C suivante :

```
int identity(int x);
```

`identity(x)` renvoi `x`.
Vous trouverez dans le fichier de question correspondant le code :

```
int identity(int x) {
  /* FIX ME */
  REMOVE_ME(x);
}
```

Que vous devrez remplacer par :

```
int identity(int x) {
  return x;
}
```

Remarques sur les programmes de test

La commande `make questionXX` produira un binaire `questionXX`. Ce binaire peut être utilisé pour tester votre réponse à la question X du sujet. Tous les binaires attendent des paramètres et affichent une aide succincte s'il sont appelés avec l'option `-help`.

Exemple 2:
Le binaire produit pour la question 1 (il s'agit d'un exemple qui ne correspond pas forcement au sujet)

```
Question 1:
./question01 graine taille
  -help   Display this list of options
  --help  Display this list of options
```

Les deux paramètres sont donc `graine` et `taille`. Les paramètres nommés `graine` (présent pour chaque question, ou presque) font tous référence à l'initialisation d'un générateur de nombre aléatoire. Dans le cas présent la commande `./question03 X Y` va initialiser le générateur de nombre aléatoire avec X (ici, Y servant de taille à la liste générée.) Pour les mêmes valeurs de X on obtiendra les mêmes données (la séquence engendrée par une graine donnée est toujours la même.)

Le reste du temps les noms des paramètres donnés dans l'aide sont explicites (par rapport au sujet.) Si vous avez une doute sur l'utilisation vous pouvez jeter un oeil sur fichier contenant les tests de la question X (`test_qXX.c`.)

Listes des fichiers à rendre

À rendre sans modifications

Ces fichiers **doivent être** dans votre répertoire de rendu (directement sans autre répertoire), mais **ne** doivent **pas** avoir été modifié par rapport à leur version originale :

Makefile
base_test.c
base_test.h
cheaters.h
question01.h
question02.h
question03.h
question04.h
question05.h
question06.h
question07.h
question08.h
question09.h
question10.h
question11.h
question12.h
question13.h
skel.h
test_q01.c
test_q02.c
test_q03.c
test_q04.c
test_q05.c
test_q06.c
test_q07.c
test_q08.c
test_q09.c
test_q10.c
test_q11.c
test_q12.c

Fichiers de réponses

Ces fichiers **doivent être** dans votre répertoire de rendu (directement sans autre répertoire) et peuvent être modifiés (si vous répondez à la question bien sûr.)

question01.c
question02.c
question03.c
question04.c
question05.c
question06.c
question07.c
question08.c
question09.c
question10.c
question11.c
question12.c