EPITA S3 promo 2018
Programmation - Épreuve machine

Marwan Burelle *

Mardi 23 décembre 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 24 points et 13 questions dans cet examen.

*marwan.burelle@lse.epita.fr

Question 1 (1)

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

```
unsigned long msb(unsigned long n);
```

`msb(n)` renvoie le rang (en commencant à 1) du bit de poids le plus élevé (correspond également aux nombres de bits minimum pour représenter la valeur.) Pour calculer le `msb` la méthode simple consiste à diviser le nombre par 2 jusqu'à 0 et de compter le nombre de passage.

Exemple 1.1:

```
> ./question01 0 5
Fixed tests:
  msb(                    1) =  1 (expected 1)
  msb(                    2) =  2 (expected 2)
  msb(                    4) =  3 (expected 3)
  msb(                    8) =  4 (expected 4)
  msb(                   16) =  5 (expected 5)
Random tests:
  msb(          426883455081) = 39
  msb(       121120649469183) = 47
  msb(                  809) = 10
  msb(                 6059) = 13
  msb(      1002196156197347) = 50
```

Question 2 (1)

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

```
void bin_string(unsigned long n, char *output);
```

`bin_string(n, output)` remplit la chaîne `output` avec les valeurs des bits de l'entier `n`. Le bit de poids fort se trouvant à la case 0. Le tableau `output` fait la bonne taille pour stocker tous les bits de n, cette taille (nécessaire pour faire l'opération) s'obtient avec l'expression `(8 * sizeof (unsigned long))`.

On rappelle que pour obtenir la valeur du bit de poids faible d'un nombre, il suffit de prendre le modulo 2 (`n % 2`) et pour obtenir un des caractères numériques, il suffit d'ajouter le chiffre au caractère `'0'` (par exemple le caractère `'1'` s'obtient en faisant `1 + '0'`.)

Exemple 2.1:

```
shell> ./question02 0 5
Fixed test:
  bin_string( 1) = "0000000000000000000000000000000000000000000000000000000000000001" (should be 1)
  bin_string( 2) = "0000000000000000000000000000000000000000000000000000000000000010" (should be 10)
  bin_string( 3) = "0000000000000000000000000000000000000000000000000000000000000011" (should be 11)
  bin_string( 4) = "0000000000000000000000000000000000000000000000000000000000000100" (should be 100)
  bin_string(42) = "0000000000000000000000000000000000000000000000000000000000101010" (should be 101010)
Random test:
  bin_string(15498727785856967622) = "1101011100010110100010101100111000110010011110110010001111000110"
```

```
bin_string(14445630959983478899) = "1100100001111001001100001101001001100110001100110100100001110011"
bin_string(16816925489926593791) = "1110100101100001101110001010001000011001010010010101110011111111"
bin_string( 6183768381126433004) = "0101010111010001001010001001010001100010010101010101100011101100"
bin_string( 5124038999138663629) = "0100011100011100001111100101001001000110111010000111110011001101"
```

Question 3 (1)

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

```
unsigned long read_bin(char *input);
```

`read_bin(input)` lit un entier en binaire dans la chaîne `input`. La chaîne contient tous les bits d'un entier de type `unsigned long`, donc sa taille est `8 * sizeof (unsigned long)` (comme pour l'exercice précédant.) Comme pour l'exercice précédant le bit de poids fort se trouve dans la case 0.

Exemple 3.1:

```
shell> ./question03 0 5
Fixed test:
  read_bin(0000000000000000000000000000000000000000000000000000000000000001) = 1 (should be 1)
  read_bin(0000000000000000000000000000000000000000000000000000000000000010) = 2 (should be 2)
  read_bin(0000000000000000000000000000000000000000000000000000000000000011) = 3 (should be 3)
  read_bin(0000000000000000000000000000000000000000000000000000000000000100) = 4 (should be 4)
  read_bin(0000000000000000000000000000000000000000000000000000000000101010) = 42 (should be 42)
Random test:
  read_bin(1011110011010110000010110001111000111010111101001010100100011101) = 13607075548612569373
  read_bin(0101110101010010100000110100001000011000111010001001110101111111) = 6724581513526549887
  read_bin(1110110101100101110001010101011110001010000010111110011011010111) = 17106295739405559511
  read_bin(1100110100111000000011100101010100110010001110010011000010111001) = 14787585135548641465
  read_bin(0000010000010000011001111010011110111001000000001101010001100001) = 292847945840093745
```

Question 4 (2)

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

```
int* binsearch(int *begin, int *end, int x);
```

`binsearch(begin, end, x)` recherche la valeur `x` dans le tableau commencant à l'adresse `begin` et se terminant à l'addresse `end`. Le tableau est trié en ordre croissant et vous devrez utiliser une dichotomie pour chercher la valeur (attention si vous utilisez une méthode plus lente, vous n'aurez pas tous les points à cause des time-out.) La fonction renvoie le pointeur sur la case du tableau contenant l'entier cherché ou `NULL` s'il n'est pas dans le tableau.

Pour rappel, la dichotomie fonctionne comme suit :

```
binsearch(tab, left, right, x):
  if right - left == 0
    return NULL
  mid = left + (right - left) / 2
  if tab[mid] == x
    return &(tab[mid])
  if tab[mid] < x
    return binsearch(tab, mid + 1, right, x)
  else
    return binsearch(tab, left, mid, x)
```

Exemple 4.1:

```
shell> ./question04 2 10
Fixed Tests:
tab[] =
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
  binsearch(tab, tab+len, 0) -> found (0)
  binsearch(tab, tab+len, 9) -> found (9)
  binsearch(tab, tab+len, 5) -> found (5)
  binsearch(tab, tab+len, 10) -> Not found
Random Tests:
tab[] =
  | -8 | -7 | -3 | -2 | -1 | 0 | 3 | 5 | 7 | 12 |
  binsearch(tab, tab+len, 10) -> Not found
  binsearch(tab, tab+len, -8) -> found (-8)
  binsearch(tab, tab+len, -3) -> found (-3)
  binsearch(tab, tab+len, 7) -> found (7)
  binsearch(tab, tab+len, 10) -> Not found
  binsearch(tab, tab+len, -3) -> found (-3)
  binsearch(tab, tab+len, 12) -> found (12)
  binsearch(tab, tab+len, 7) -> found (7)
  binsearch(tab, tab+len, 1) -> Not found
  binsearch(tab, tab+len, -9) -> Not found
```

Question 5 (3)

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

```
void quick_sort(int *left, int *right);
```

`quick_sort(left, right)` trie en place le tableau d'entier compris entre l'addresse `left` et l'addresse `right` (exclue.) Comme pour l'exercice précédant, vous devez utiliser la méthode décrite dans le sujet au risque d'avoir un algo trop lent.

```
partition(tab, left, right, pivot):
  p = tab[pivot]
  swap(tab[pivot], tab[right - 1])
  pivot = left
  for i = left to right - 2 do
    if tab[i] <= p
      swap(tab[i], tab[pivot])
      pivot = pivot + 1
  done
  swap(tab[pivot], tab[right - 1])
  return pivot

quick_sort(tab, left, right):
  if right - left == 0
    return
  pivot = left + (right - left) / 2
  pivot = partition(tab, left, right, pivot)
  quick_sort(tab, left, pivot)
  quick_sort(tab, pivot + 1, right)
```

Exemple 5.1:

```
shell> ./question05 0 10
Tests with sorted array:
tab[] =
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
After sort:
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
is sorted: OK
Tests with revert sorted array:
tab[] =
  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
After sort:
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
is sorted: OK
Tests with random array:
tab[] =
  | 13 | 16 | 7 | -5 | 23 | 15 | 16 | 22 | 29 | 1 |
```

```
After sort:
  | -5 | 1 | 7 | 13 | 15 | 16 | 16 | 22 | 23 | 29 |
is sorted: OK
```

Question 6 (2)

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

```
void insert(struct list *l, int x);
```

`insert(l, x)` insert dans la liste chaînée `l` (triée dans l'ordre croissant) la valeur `x` à la bonne place si cette valeur n'est pas déjà présente, ne fait rien si celle-ci est présente.

Le type liste est le suivant :

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

Les listes que votre fonction recevera en paramètres commence toujours par une sentinelle (le pointeur ne sera donc jamais `NULL` même si la liste est vide) que vous devrez laisser inchangée (et qui n'intervient pas dans l'ordre des éléments de la liste, sa valeur étant non-spécifiée.)

Exemple 6.1:

```
shell> ./question06 0 10
RANDOM TESTS:
  Inserting into an empty list:
    inserting: 3
    inserting: 6
    inserting: 17
    inserting: 15
    inserting: 13
    inserting: 15
    inserting: 6
    inserting: 2
    inserting: -1
    inserting: -9
Sentinel untouched:  OK
List after insertions:
  (-9)->(-1)->(2)->(3)->(6)->(13)->(15)->(17)->NULL
List is sorted: OK
```

Question 7 (2)

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

```
int circular_list_mem(struct list *l, int x);
```

`circular_list_mem(l, x)` renvoie vrai (1) si `x` est dans la list `l` et faux (0) sinon.

Les listes utilisées sont du même type que pour l'exercice précédant. Mais attention : ces listes n'ont pas de sentinelle et sont **circulaire** ! Vous devrez donc faire attention à ne pas chercher infiniment la fin de la liste, mais vous arrêtez lorsque vous aurez fait le tour de

la liste. La liste vide est représentée par un pointeur `NULL`. Pensez à vérifier que votre algorithme fonctionne pour la liste vide, justement.

Exemple 7.1:

```
Random circular list:
  (5)->(1)->(1)->(14)->(10)->(10)->(17)->(13)->(5)->(36)->...
  circular_list_mem(l, 27):  not found
  circular_list_mem(l, 6):  not found
  circular_list_mem(l, 17):  found
  circular_list_mem(l, 6):  not found
  circular_list_mem(l, 8):  not found
  circular_list_mem(l, 22):  not found
  circular_list_mem(l, 14):  found
  circular_list_mem(l, 19):  not found
  circular_list_mem(l, 3):  not found
  circular_list_mem(l, 13):  found
  circular_list_mem(l, 26):  not found
```

Question 8 (1)

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
```

Question 9 (1)

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

```
int my_strcmp(const char *s1, const char *s2);
```

`my_strcmp(s1, s2)` compare les deux chaînes de caractères (terminée par `'\0'`) `s1` et `s2`. La fonction renvoie 0 si les deux chaînes sont identiques, une valeur négative si la première chaînes est plus *petite* et une valeur positive sinon.

On rappelle que la comparaison de deux chaînes, suit la règle de l'odre lexicographique : on compare les deux chaînes caractère par caractère, si les deux caractères courant sont égaux, on passe au suivant, s'ils sont différents, la chaîne qui a le plus petit caractère est la plus petite, enfin si l'une des chaînes se termine avant l'autre, elle est plus petite.

Il est tout à fait valable de renvoyer la différence entre les 2 premiers caractères qui diffèrent entre les deux chaînes (par exemple pour les chaînes `"aa"` et `"ad"` vous pouvez renvoyer -3.)

Vous pouvez lire la page de manuel (`strcmp(3)`) pour plus d'information.

Exemple 9.1:

```
shell> ./question09 0 10
Fixed Tests:
  s1: "aaaaaa"
  s2: "aaaaab"
my_strcmp(s1,s1) = s1 == s1  (OK)
my_strcmp(s1,s2) = s1 < s2  (OK)
my_strcmp(s2,s1) = s2 > s1  (OK)

Random Tests:
  s1: "n{6\Pavw[:"
  s2: "mO4=ZvMD^b"
my_strcmp(s1,s2) = s1 > s2
```

Question 10 (2)

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

```
int height(struct tree *t);
```

`height(t)` renvoie la hauteur de l'arbre binaire `t`. On rappelle que la hauteur est définie récursivement de la manière suivante : si l'arbre est vide sa hauteur est -1, sinon sa hauter correspond à 1 plus la hauteur maximale entre ses deux fils.

Les arbres sont représentés par le type suivant, l'arbre vide est le pointeur `NULL` :

```
struct tree {
  int           key;
  struct tree  *left, *right;
};
```

Exemple 10.1:

```
shell> ./question10 0 3
Tests:
(1
  (2
    (3)
  )
  (4
    (5)
    (6)
  )
)
height(t) : 2 (OK)
```

Question 11 (3)

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

```
size_t prefix_list(struct tree *t, int keys[]);
```

`prefix_list(t, keys)` remplit le tableau `keys` avec les clefs de l'arbre `t` dans l'ordre de rencontre préfixe d'un parcours en profondeur (main gauche). La fonction devra renvoyer le nombre de nœuds rencontrés lors du parcours.

On notera que vous ne disposez pas d'un compteur de position pour le tableau `keys` : vous devrez, lors des appels récursifs du parcours profondeur, passer ce tableau *décaler* du nombre d'élément ajouter dans le nœud courrant, plus éventuellement ceux du fils gauche lors de l'appel sur le fils droit.

Le type utilisé est le même que pour l'exercice précédant.

Exemple 11.1:

```
shell> ./question11 0 3
Tests:
(1
  (2
    (3)
  )
  (4
    (5)
    (6)
  )
)
Size of t: 6 (in range: OK)
Keys:
  | 1 | 2 | 3 | 4 | 5 | 6 |
```

Pour simplifier les test pendant l'épreuve, les nœuds sont numérotés de façon à être dans l'ordre préfixe, mais attention, pour l'évaluation la moulinette utilisera des valeurs aléatoires.

Question 12 (3)

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

```
int is_white(struct image *img);
```

`is_white(img)` renvoie vraie (1) si l'image est complètement blanche (voir la suite) et faux (0) sinon.

Les images manipulées ont le type suivant :

```
struct image {
  int           white;
  struct image *upleft, *upright, *downleft, *downright;
};
```

Ce sont des images sous forme de *quad-tree* : une image est soit noire (le pointeur est `NULL`) soit blanche (le pointeur n'est pas `NULL` et le champ `white` est à vrai) soit composée de 4 images (les 4 pointeurs de la structure.)

Exemple 12.1:

```
shell> ./question12 0 3
fixed tests:
########
########
########
########
########
########
########
########
is_white: yes (OK)
........
........
........
........
........
........
........
........
is_white: no (OK)
random tests:
########
########
..##....
..##....
######..
######..
......##
```

```
......##
is_white: no
```

Question 13 (2)

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

```
void decipher(char *msg, size_t len, char *key, size_t keylen);
```

`decipher(msg, len, key, keylen)` déchiffre le message dans la chaîne `msg` de longueur `len` avec la clef `key` de longueur `keylen`. Le déchiffrement se fera en place (vous devez remplacer le message original.)

L'algorithme de chiffrement utilisé est *Végenaire*, c'est algorithme très simple, le déchiffrement s'effectue de la manière suivante : pour chaque caractère du message, on obtient le caractère décodé par la formule :

```
clear[i] = (msg[i] - key[i % keylen]) % 128
```

Attention, en C, les calculs arithmétiques se font sur des `int`, il faudra donc s'assurer que vous obtenez bien des éléments de type `char` (pensez aux *casts*.)

Exemple 13.1:
Exercice de Noël : vous découvrirez le résultat par vous même.

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
message
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
test_q13.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
question13.c