EPITA S3 promo 2019
Programmation - Épreuve machine

Marwan Burelle *

Vendredi 30 octobre 2015

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 dûe au non respect des consignes explicites ne sera pas contestable.

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

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 `submission`.

À titre de rappel, voici les commandes pour commencer votre épreuve :

```
> cd
> cd subject
> cp -R Skel/* ~/submission/
```

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.

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 fact(unsigned long n);
```

`fact(n)` calcule factorielle de n.
On rappelle que la suite factorielle est définie par :

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

Exemple 1.1:

```
shell> ./question01 0 5
./question01 0 5
Fixed tests:
fact( 0) = 1
fact( 1) = 1
fact( 2) = 2
fact( 3) = 6
fact( 4) = 24
fact( 5) = 120
Random tests:
fact( 7) = 5040
fact( 6) = 720
fact( 9) = 362880
fact( 3) = 6
fact( 1) = 1
```

Question 2 (1)

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

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

`fibo(n)` calcule le rang n de la suite de Fibonacci.
On rappelle que la suite de Fibonacci est définie par :

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

Exemple 2.1:

```
shell> ./question02 0 5
```

```
Fixed tests:
fibo( 0) = 0
fibo( 1) = 1
fibo( 2) = 1
fibo( 3) = 2
fibo( 4) = 3
fibo( 5) = 5
Random tests:
fibo(33) = 3524578
fibo(36) = 14930352
fibo(27) = 196418
fibo(15) = 610
fibo(43) = 433494437
```

Question 3 (1)

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

```
unsigned long my_intsqrt(unsigned long n);
```

`my_intsqrt(n)` calcule la racine carrée entière de n.
La racine carrée entière est la solution (entière) x à l'inéquation :

$$x^2 \leq n < (x+1)^2$$

Pour résoudre ce problème on utilise la méthode de Héron (une variante de la méthode de Newton). On considère une première approximation x de la racine (cette approximation doit être supérieur à la racine et inférieur à n, on prendra donc n comme première valeur). On calcule la prochaine approximation comme étant la moyenne arithmétique entre x et n/x et on continue tant que x est plus grand que n/x (c'est à dire tant que x est plus grand que la racine cherchée).

Exemple 3.1:

```
shell> ./question03 0 5
Fixed tests:
        my_intsqrt(    0)       =      0        [OK]
        my_intsqrt(    1)       =      1        [OK]
        my_intsqrt(    4)       =      2        [OK]
        my_intsqrt(   16)       =      4        [OK]
        my_intsqrt(   64)       =      8        [OK]
        my_intsqrt(  256)       =     16        [OK]
        my_intsqrt( 1024)       =     32        [OK]
        my_intsqrt( 4096)       =     64        [OK]
        my_intsqrt(16384)       =    128        [OK]
```

```
        my_intsqrt(65536)       =    256        [OK]
Random tests:
        my_intsqrt(1804289383)  =  42476        [OK]
        my_intsqrt( 846930886)  =  29102        [OK]
        my_intsqrt(1681692777)  =  41008        [OK]
        my_intsqrt(1714636915)  =  41408        [OK]
        my_intsqrt(1957747793)  =  44246        [OK]
```

Question 4 (1)

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

```
void swap(int *a, int *b);
```

`swap(a,b)` échange les valeurs pointées par `a` et `b`.

Exemple 4.1:

```
shell> ./question04 0 5
Testing swap:
        Before swap: a = 1804289383 | b =  846930886
        After  swap: a =  846930886 | b = 1804289383    [OK]

        Before swap: a = 1681692777 | b = 1714636915
        After  swap: a = 1714636915 | b = 1681692777    [OK]

        Before swap: a = 1957747793 | b =  424238335
        After  swap: a =  424238335 | b = 1957747793    [OK]

        Before swap: a =  719885386 | b = 1649760492
        After  swap: a = 1649760492 | b =  719885386    [OK]

        Before swap: a =  596516649 | b = 1189641421
        After  swap: a = 1189641421 | b =  596516649    [OK]
```

Question 5 (2)

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

```
int array_sum(int *begin, int *end);
```

`array_sum(begin, end)` calcule la somme des cases du tableau entre `begin` (inclus) et `end` (exclus.) Si `end - begin = 0`, alors la fonction renverra 0.

Exemple 5.1:

```
shell> ./question05 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  array_sum = 15 [OK]
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  array_sum = 3754
```

Question 6 (2)

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

```
int array_min(int *begin, int *end);
```

`array_min(begin, end)` cherche la valeur minimale dans le tableau entre `begin` (inclus) et `end` (exclus.) La fonction n'est défénie que si `end - begin > 0`.

Exemple 6.1:

```
shell> ./question06 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  array_min = 1 [OK]
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  array_min = 383
```

Question 7 (3)

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

```
int* avg_partition(int *begin, int *end);
```

`avg_partition(begin, end)` regroupe les valeurs inférieures ou égale à la moyenne dans la première partie du tableau et les valeurs supérieures dans la seconde partie. La fonction renvoie l'adresse de la première case contenant une valeur supérieur à la moyenne.

Le partitionnement fonctionne de manière similaire à celui de quick-sort, sauf que le pivot est obtenu en calculant la moyenne des cases du talbeau. Voici le pseudo-code de l'algorithme de partitionnement :

```
avg_partition(tab, left, right):
  a = average of tab
  pivot = left
  for i = left to right - 1 do:
    if tab[i] <= a:
      swap tab[i] and tab[pivot]
      pivot = pivot + 1
    end if
  done
  return pivot
```

Exemple 7.1:

```
shell> ./question07 0 8
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |
  run avg_partition
  lower half:
    |   1 |   2 |   3 |   4 |
  upper half:
    |   5 |   6 |   7 |   8 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 | 386 | 492 |
  run avg_partition
  lower half:
    | 383 | 335 | 386 | 492 |
  upper half:
    | 793 | 886 | 777 | 915 |
```

Question 8 (3)

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

```
void array_merge(int *dst, int *a1, int *a2, int *e1, int *e2);
```

`array_merge(dst, a1, a2, e1, e2)` fusionne les deux tableaux triés `a1` et `a2` dans le tableau `dst`. `e1` (respectivement `e2`) est le pointeur de fin (exclus) du tableau `a1` (respectivement `a2`). La zone mémoire pointée par `dst` est réputée suffisament grande pour contenir le contenu des deux tableaux (la fin du tableau `dst` se trouve à l'adresse `dst + e1 - a1 + e2 - a2`.)

Il n'y a aucune contrainte de taille entre `a1` et `a2`.

Le résultat de la fusion (dans `dst`) doit bien évidement être trié.

Exemple 8.1:

```
shell> ./question08 0 7
Fixed tests:
  array1:
    |   1 |   2 |   3 |
  array2:
    |   4 |   5 |   6 |   7 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Fixed tests (2):
  array1:
    |   1 |   3 |   5 |   7 |
  array2:
    |   2 |   4 |   6 |
  array_merge(dst, array1, array2, ...)
  dst:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |

Random tests:
  array1:
    |  10 |  28 |  44 |
  array2:
    |  19 |  26 |  39 |  49 |
  array_merge(dst, array1, array2, ...)
  dst:
    |  10 |  19 |  26 |  28 |  39 |  44 |  49 |
```

Question 9 (4)

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

```
void array_rot(int array[], size_t len, size_t offset);
```

`array_rot(array, len, offset)` effectue une *rotation* de `offset` cases du contenu du tableau `array` (de longueur `len`).

La rotation du tableau consiste à considérer le tableau comme étant *circulaire* et à décaler les cases vers la droite. Comme le tableau est circulaire, pour une rotation de 1, la dernière case prend la place de la première (la première devient la deuxième ...)

Vous pouvez implémenter de manière naïve la rotation de `offset` cases, comme la répétition (`offset` fois) du rotation de 1 case. Mais la version la plus efficace consiste à copier les `offset` dernières cases du tableau dans un tableau temporaire,

puis à décaler le tableau de `offset` cases vers la droite et pour enfin copier les cases sauvées en début de tableau. Ce qui donne en pseudo code :

```
array_rot(array, len, offset):
  tmp = malloc(offset * sizeof (int))
  copy offset cells from array[len - offset] to tmp
  shift array by offset cells to the right
  copy offset cells from tmp[0] to array
  free(tmp)
```

Pour cette second méthode, vous devrez allouer (avec `malloc(3)`) un tableau temporaire (suffisament grand, donc de `offset` entiers). Bien évidement dans ce cas vous devrez penser à libérer le tableau temporaire que vous avez allouer (avec `free(3)`).

Exemple 9.1:

```
shell> ./question09 0 8
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |
  array_rot(array, 8, 1)
  array:
    |   8 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |
  array_rot(array, 8, 4)
  array:
    |   4 |   5 |   6 |   7 |   8 |   1 |   2 |   3 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 | 386 | 492 |
  array_rot(array, 8, 2)
  array:
    | 386 | 492 | 383 | 886 | 777 | 915 | 793 | 335 |
```

Question 10 (3)

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

```c
void insert_sort(int *begin, int *end);
```

`insert_sort(begin, end)` trie le tableau entre `begin` (inclus) et `end` (exclus.) On utilisera un tri par insertion, dont voici l'algorithme :

```
insert_sort(tab, left, right):
  for cur = left to right - 1 do:
    x = tab[cur]
    i = cur
    while i > left && x < tab[i - 1] do:
      tab[i] = tab[i - 1]
      i = i - 1
    done
    tab[i] = x
  done
```

Exemple 10.1:

```
shell> ./question10 0 5
./question10 0 5
Fixed tests:
  sorted array:
    |   1 |   2 |   3 |   4 |   5 |
  sorting ... [OK]
  after insert_sort:
    |   1 |   2 |   3 |   4 |   5 |

  reverse sorted array:
    |   5 |   4 |   3 |   2 |   1 |
  sorting ... [OK]
  after insert_sort:
    |   1 |   2 |   3 |   4 |   5 |

Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  sorting ... [OK]
  after insert_sort:
    | 383 | 777 | 793 | 886 | 915 |
```

Question 11 (1)

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

```
str_toupper(char *s);
```

`str_toupper(s)` transforme toutes les lettres minuscules de la chaîne `s` (terminée par le charactère `'\0'`) en majuscule et laisse inchangés les autres charactères.

On rappel que les charactères en C ne sont que des entiers et que si `c` est une variable de type `char` contenant un lettre minuscule, alors `c` vérifie : `'a' <= c && c <= 'z'`. De même, si `c` est une lettre majuscule, alors : `'A' <= c && c <= 'Z'`.

Exemple 11.1:

```
shell> ./question11 0 26
Fixed tests:
  s = "abcdefghijklmnopqrstuvwxyz"
  str_to_upper(s)
  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Random tests:
  s = "n{6\Pavw[:mO4=ZvMD^bvU;e'R"
  str_to_upper(s)
  s = "N{6\PAVW[:MO4=ZVMD^BVU;E'R"
```

Question 12 (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 12.1:

```
shell> ./question12 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 13 (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 13.1:

```
shell> ./question13 0 10
src = "n{6\Pavw[:"

test: mystrncpy(dst,src,11)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,5)
dst = "n{6\P"
-- check:
  first char:  [OK]
  last char:  [OK]
  overflow:  [OK]

test: mystrncpy(dst,src,20)
dst = "n{6\Pavw[:"
-- check:
  first char:  [OK]
  last char:  [OK]
  0 fill:  [OK]
  overflow:  [OK]

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

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 `submission` 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`.)