EPITA S3 API Promo 2020
Programmation - Épreuve machine

Marwan Burelle *

28 octobre 2016

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* `submission` *vous trouverez tous les fichiers pour faire votre exam (squelette des questions.)*

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`.

Dans le répertoire `submission` *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 17 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):

```
size_t digit_count(size_t n);
```

`digit_count(n)` calcule le nombre de chiffre (en décimal) de `n`.

Exemple 4.1:

```
shell> ./question04 0 5
Fixed tests:
digit_count(1) =      1
digit_count(10) =     2
digit_count(100) =    3
digit_count(1000) =   4
digit_count(10000) =  5
digit_count(100000) = 6
Random tests:
digit_count(55062) =    5
digit_count(13138224) = 8
digit_count(3823726) =  7
digit_count(351506) =   6
digit_count(9320572) =  7
```

Question 5 (1)

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 5.1:

```
shell> ./question05 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 6 (1)

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

```
size_t array_count_occurences(int *begin, int *end, int x);
```

`array_count_occurences(begin, end, x)` compte le nombre de fois où la valeur `x` apparaît dans le tableau entre `begin` (inclus) et `end` (exclus.)

Exemple 6.1:

```
shell> ./question06 1 10
Fixed tests:
  array:
    |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 |
  array_count_occurences(begin, end, 5): 1
Random tests:
  array:
    |  0 |  9 |  8 |  5 |  1 |  8 |  4 |  7 |  5 |  7 |
  array_count_occurences(begin, end, 8): 2
  array_count_occurences(begin, end, 9): 1
  array_count_occurences(begin, end, 42): 0
```

Question 7 (1)

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 7.1:

```
shell> ./question07 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 8 (1)

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

```
void array_reverse(int *begin, int *end);
```

`array_reverse(begin, end)` inverse le contenu des cases du tableau entre `begin` (inclus) et `end` (exclus.)

Exemple 8.1:

```
shell> ./question08 0 5
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |
  after reverse:
    |   5 |   4 |   3 |   2 |   1 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 |
  after reverse:
    | 793 | 915 | 777 | 886 | 383 |
shell> ./question08 0 6
Fixed tests:
  array:
    |   1 |   2 |   3 |   4 |   5 |   6 |
  after reverse:
    |   6 |   5 |   4 |   3 |   2 |   1 |
Random tests:
  array:
    | 383 | 886 | 777 | 915 | 793 | 335 |
  after reverse:
    | 335 | 793 | 915 | 777 | 886 | 383 |
```

Question 9 (3)

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

```
int next_permutation(int array[], size_t len);
```

`next_permutation(array, len)` permute le tableau `array` de longueur `len` en place. La permutation obtenue est la permutation suivante dans l'ordre lexicographique. La fonction renvoie faux (0) si le tableau correspond à la plus grande permutation dans l'ordre lexicographique (tableau trié dans l'ordre décroissant), dans ce cas le tableau après permutation correspondra à la plus petite permutation (tableau trié dans l'ordre croissant.) Dans tous les autres cas la fonction renvoie vrai.

Voici la série des permutations dans l'ordre lexicographique pour un tableau de taille 3 :

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

L'algorithme en pseudo code :

```
def next_permutation(v):
    nxt = len(v) - 1
    while nxt > 0 and v[nxt] < v[nxt - 1]:
        nxt -= 1
    if nxt <= 0:
        v.reverse()
        return False
    nxt -= 1
    repl = nxt + 1
    while repl < len(v) - 1 and v[repl + 1] > v[nxt]:
        repl += 1
    v[nxt], v[repl] = v[repl], v[nxt]
    reverse_range(v, nxt + 1, len(v))
    return True
```

La fonction `reverse_range(v, begin, end)` inverse en place le contenu du tableau `v`, sur l'interval entre `begin` (inclus) et `end` (exclu).

Exemple 9.1:

```
shell> ./question09 0 3

Fixed tests:
  array:
|   1 |   2 |   3 |
```

```
|   1 |   3 |   2 |
|   2 |   1 |   3 |
|   2 |   3 |   1 |
|   3 |   1 |   2 |
|   3 |   2 |   1 |
Random tests:
  array:
| 383 | 886 | 777 |
  next_permutation: 1
| 777 | 383 | 886 |
```

Question 10 (2)

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 suffisamment 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 10.1:

```
shell> ./question10 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 11 (3)

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

```
size_t prime_sieve(unsigned int n, unsigned int primes[]);
```

`prime_sieve(n, primes)` calcule les nombres premiers inférieurs ou égals à n, les stocke dans le tableau `primes` et renvoie le nombre de cases utilisées dans le tableau. Le tableau `primes` est alloué et suffisament grand.

Le calcul des nombres premiers se fera en utilisant l'algorithme du crible d'Eratosthene. Le principe est simple, il s'agit pour chaque nombre premier trouvé d'éliminer de la liste tous ses multiples.

Voici une version pseudo code :

```
prime_sieve(n, primes):
    numbers = [ True for i in range(n + 1) ]
    primes.append(1)
    i = 2
    while i <= n:
        primes.append(i)
        for j in range(i + 1, n + 1):
            if j % i == 0:
                numbers[j] = False
        i += 1
        while i <= n and not numbers[i]:
            i += 1
    return len(primes)
```

Il y a un certain nombre d'optimisations possibles sur cet algorithme, notamment dans le marquage des multiples.

Attention : dans votre implémentation en C vous aurez besoin d'allouer et d'initialiser un tableau d'entier (correspondant au tableau `numbers` de l'algo), il faut allouer ce tableau correctement avec `malloc` et le libérer avec `free`. Voici une solution possible :

```c
int *numbers;
// get an array of (n + 1) integers
// numbers is allocated but not initialized
numbers = malloc((n + 1) * sizeof (int));

// Do your work here

// release the array
free(numbers);
```

Exemple 11.1:

```
shell> ./question11 20
Tests:
  Found 9 prime numbers smaller or equal to 20
    1
    2
    3
    5
    7
    11
    13
    17
    19
```

Question 12 (2)

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

```c
int lex_compare(int *b1, int *e1, int *b2, int *e2);
```

`lex_compare(b1, e1, b2, e2)` compare les deux tableaux (compris entre `b1` et `e1` pour l'un et `b2` et `e2` pour l'autre) et renvoie une valeur négative si le premier est plus petit, 0 s'ils sont égaux et une valeur positive si le premier est plus grand. L'ordre utilisé est l'ordre lexicographique (l'ordre du dictionnaire pour faire simple.)

On rappelle que l'ordre lexicographique se définit récursivement de la manière suivante :

— l'un des tableaux est vide, c'est le plus petit, si les deux sont vides ils sont égaux ;
— les premiers éléments des deux tableaux sont égaux, on compare les deux tableaux en se décalant d'une case (dans chaque) ;
— le tableau le plus petit est celui dont le premier élément est le plus petit.

Par exemple, le mot *ab* est plus grand que le mot *aaa* car *b* est plus grand que *a*.

Exemple 12.1:

```
shell> ./question12 0 5
Random tests:
  a:
    |   3 |   6 |  17 |  15 |  13 |
  b:
    |  15 |   6 |  12 |   9 |   1 |
lex_compare: a is smaller than b
shell> ./question12 1 5
Random tests:
  a:
    |  10 |  19 |   8 |  15 |   1 |
  b:
    |  18 |   4 |  17 |   5 |   7 |
lex_compare: a is smaller than b
shell> ./question12 2 5
Random tests:
  a:
    |   6 |   5 |   8 |   0 |   5 |
  b:
    |   0 |  12 |  16 |   1 |   4 |
lex_compare: a is bigger than b
```

Question 13 (1)

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

```
long dot_product(long a[], long b[], size_t len);
```

`dot_product(a, b, len)` renvoie le produit des vecteurs `a` et `b` (de même longueur `len`).

On rappelle que le produit (*dot product*) de deux vecteurs est défini par :

$$a.b = \Sigma_i a[i] \times b[i]$$

Exemple 13.1:

```
shell> ./question13 0 2
Fixed tests:
  a:
    |   3 |   6 |
  b:
    |   1 |   1 |
```

```
dot_product(a,b) = 9

  b:
    |   1 |   0 |
dot_product(a,b) = 3

  b:
    |   0 |   1 |
dot_product(a,b) = 6

Random tests:
  a:
    |  17 |  15 |
  b:
    |  13 |  15 |
dot_product(a,b) = 446
```

Question 14 (1)

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

```
struct matrix* matrix_transpose(struct matrix *A);
```

`matrix_transpose(A)` calcule la transposée de la matrice `A`.

Les matrices sont représentée par le type suivant :

```
struct matrix {
  size_t lines, cols;
  int   *data;
};
```

De manière évidente, `lines` représente le nombre de lignes de la matrice et `cols` son nombre de colonnes. Le champ `data` est un pointeur sur une zone mémoire contenant `lines * cols` entiers correspondant aux cases de la matrice stockées en mode *ligne d'abord*.

Pour accéder à la case (i, j) de la matrice `A`, on utilisera le décallage classique : `A->data[i * A->cols + j]`.

On rappelle que la transposée de la matrice A de dimensions $n \times m$ est la matrice A^t de dimensions $m \times n$ dont les cases sont définies par :

$$A^t_{i,j} = A_{j,i}$$

Exemple 14.1:

```
shell> ./question14 0 2 5
Random tests:
A =
|  83 |  86 |  77 |  15 |  93 |
|  35 |  86 |  92 |  49 |  21 |
B = matrix_transpose(A)
|  83 |  35 |
|  86 |  86 |
|  77 |  92 |
|  15 |  49 |
|  93 |  21 |
C = matrix_transpose(B)
|  83 |  86 |  77 |  15 |  93 |
|  35 |  86 |  92 |  49 |  21 |
```

Question 15 (2)

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

```
struct matrix *matrix_mul(struct matrix *A, struct matrix *B);
```

`matrix_mul(A, B)` calcule le produit des matrices `A` et `B`. Le résultat est une nouvelle matrice qui devra être correctement créée et allouée.

Les matrices sont représentée par le type suivant :

```
struct matrix {
  size_t lines, cols;
  int   *data;
};
```

De manière évidente, `lines` représente le nombre de lignes de la matrice et `cols` son nombre de colonnes. Le champ `data` est un pointeur sur une zone mémoire contenant `lines * cols` entiers correspondant aux cases de la matrice stockées en mode *ligne d'abord*.

Pour accéder à la case (i, j) de la matrice `A`, on utilisera le décallage classique : `A->data[i * A->cols + j]`.

On rappelle que le produit d'une matrice A de dimensions $n \times m$ par une matrice B de dimensions $m \times p$ est la matrice AB de dimension $(n \times p)$, dont les cases sont données par la formule suivante :

$$AB_{i,j} = \Sigma_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

Exemple 15.1:

```
shell> ./question15 0 3 5
Test with id matrix:
A =
|  83 |  86 |  77 |
|  15 |  93 |  35 |
|  86 |  92 |  49 |
id =
|   1 |   0 |   0 |
|   0 |   1 |   0 |
|   0 |   0 |   1 |
C = A * id
|  83 |  86 |  77 |
|  15 |  93 |  35 |
|  86 |  92 |  49 |
Random tests:
A =
|  21 |  62 |  27 |  90 |  59 |
|  63 |  26 |  40 |  26 |  72 |
|  36 |  11 |  68 |  67 |  29 |
B =
|  82 |  30 |  62 |
|  23 |  67 |  35 |
|  29 |   2 |  22 |
|  58 |  69 |  67 |
|  93 |  56 |  11 |
C = A * B
| 14638 | 14352 | 10745 |
| 15128 | 9538 | 8230 |
| 11760 | 8200 | 8921 |
```

Question 16 (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 16.1:

```
shell> ./question16 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 17 (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 17.1:

```
shell> ./question17 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`.)