EPITA S3 Promi 2020
Practical Programming - Computer Test

Marwan Burelle[*]

October 28, 2016

Instructions:

You must read the whole subject and all these instructions. Every explicit instructions in the subject are mandatory. Points lost for ignoring subject rules are not open to arguments, including compiling issues or usage of directory hierarchy.

Your home directory during the test is temporary, in this directory you'll find a directory `subject` *and a directory* `submission`*. In the* `submission` *directory, you'll find all the needed files for the test (questions template.)*

You must make regular uploads to be sure not to lose your work. In order to upload your work, simply call the command `submission`.

In `submission` *directory, you'll find: a* `Makefile` *offering targets to compile your code, some annex files, a file for each questions named* `questionXX.c`*. Only question files can be modified, all other files wil be replaced by the original one during the automatic correction.*

The `Makefile` *offers a target building a test program for each question. This test program will perform all the interraction part (input and output) and call your (or yours) function(s) with the correct expected parameters. In order to target the build of these test programs, you need to issue the command (for question number XX):* `make questionXX`.

During automatic correction, this `Makefile` *will be used and thus the question XX will be evaluated only if* `make questionXX` *succeed. Of course,* ***the grade will depends on the correctness of your answer****.*

Grades information are only indicative and may be changed later.

At the end of the document, you'l find the extra sections providing advices about test programs.

There are 24 points and 17 questions in this test.

[*]marwan.burelle@lse.epita.fr

Question 1 (1)

Write the following function(s):

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

`fact(n)` compute factorial of n.

Factorial sequence is defined by:

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

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)

Write the following function(s):

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

`fibo(n)` compute the rank n of the Fibonacci sequence.

The Fibonacci sequence is defined by:

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

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)

Write the following function(s):

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

`my_intsqrt(n)` compute the integer part of the square root of n.

The integer part of the square root is the (integer) solution x of the inequation:

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

In order to solve this problem, we'll use the Heron method (variation of the Newton method.) Let x be an upper approximation of the square root (thus bigger than the root but smaller than n, we can take n as a initial approximate value.) We compute the next approximation as the arithmetic mean between x and n/x et we continue while x is bigger than n/x (*i.e.* x is bigger than the expected root.)

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)

Write the following function(s):

```c
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)

Write the following function(s):

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

array_min(begin, end) finds the minimum value of the cells in the array betwee begin (included) and end (excluded.) The function is only defined if 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)

Write the following function(s):

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

array_count_occurences(begin, end, x) count the number of time where the value x appears in the array between begin (included) and end (excluded.)

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)

Write the following function(s):

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

`array_sum(begin, end)` computes the sum of the cells in the array betwee `begin` (included) and `end` (excluded.) If `end - begin = 0`, the function will return 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)

Write the following function(s):

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

`array_reverse(begin, end)` reverse the content of the cells in the array between `begin` (included) and `end` (excluded.)

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)

Write the following function(s):

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

`next_permutation(array, len)` rearrange the array `array` of length `len`, in place. The permutation is the smallest order superior to the original array using the lexicographic order. The function returns false (0), if the array is already the greatest permutation (array sorted in reverse order) the function rearrange the array with the lowest permutation (array sorted in increasing order.) Otherwise the function always returns true.

Here is the full sequence of permutations (in lexicographic order) for an array of size 3:

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

The algorithm:

```
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
```

`reverse_range(v, begin, end)` reverses in place the content of the array v on the range `begin` (included) and `end` (excluded).

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)

Write the following function(s):

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

`array_merge(dst, a1, a2, e1, e2)` merges the two sorted arrays `a1` and `a2` in the array `dst`. `e1` (respectively `e2`) is the end pointer (excluded) of the array `a1` (respectively `a2`). The area pointed to by `dst` is supposed to be sufficient in order to contain the content of both arrays (the end of `dst` is at address `dst + e1 - a1 + e2 - a2`.)

There's no constraint of size betwee `a1` and `a2`.

Of course, the result of merging (in `dst`) must be sorted.

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)

Write the following function(s):

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

`prime_sieve(n, primes)` computes the prime numbers smaller or equal to *n*, stores them in the array `primes` and returns the number of used cells. The array `primes` is already allocated with enough rooms.

Prime numbers discovery will be implemented using the sieve of Eratosthene. The idea is pretty simple, for each discovered prime number, you mark its multiple as non prime in the list of candidates.

Here is a pseudo-code version of the algorithm:

```
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)
```

There're several possible optimisations in this algorithm, especially in the loop marking multiples.

Warning: in you C implementation, you'll need to allocate and initialize an array of integer (corresponding to the array `numbers` in the algo), you need to carefuly allocate this array using `malloc` and releaser it with `free`. Here is a possible *solution*:

```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)

Write the following function(s):

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

`lex_compare(b1, e1, b2, e2)` compare the two arrays (betweem `b1` and `e1` for the first one and between `b2` and `e2` for the seond) and returns a negative value if the first array is smaller, 0 if they're equal and a negative value otherwise. The expected order is the lexicagraphic order (the same as the order of dictionnary.)

The lexicographic order is recursively defined as follow:

- one of the array is empty, it's the smaller one, if both are empty, they are equal.
- the first elements of the two arrays are equal, the result is the result of the comparison for the two arrays without their first cell.
- the smallest array is the one with the smallest first element.

Example: the work *ab* is bigger than the word *aaa* since *b* is smaller than *a*.

Exemple 12.1:

```
shell> ./question12 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 13 (1)

Write the following function(s):

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

`dot_product(a, b, len)` returns the product of the vectors `a` et `b` (of the same length `len`).

Reminder: the dot product of two vectors is defined by:

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

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)

Write the following function(s):

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

`matrix_transpose(A)` compute the transpose of the matrix `A`.

Matrices are reprensented using the following type:

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

Obviously, `lines` represents the number of lines and `cols` the number of columns. The field `data` is a pointer to a memory area of `lines * cols` integers, storing the matrix cells (line first.)

In order to access the cell (i, j) in matrix `A`, we'll use the classic shift: `A->data[i * A->cols + j]`.

The tranpose of matrix A of dimensions $n \times m$ is the matrix A^t of dimensions $m \times n$, its cells are defined by:

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

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)

Write the following function(s):

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

`matrix_mul(A, B)` computes the product of the matrices `A` and `B`. The result is a newly created and allocated matrix.

Matrices are reprensented using the following type:

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

Obviously, `lines` represents the number of lines and `cols` the number of columns. The field `data` is a pointer to a memory area of `lines * cols` integers, storing the matrix cells (line first.)

In order to access the cell (i, j) in matrix `A`, we'll use the classic shift: `A->data[i * A->cols + j]`.

Remember, the product of a matrix A of dimensions $n \times m$ by a matrix B of dimensions $m \times p$ is the matrix AB of dimensions $(n \times p)$, cells are computed by:

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

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)

Write the following function(s):

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

`mystrlen(s)` function calculates the length of the string s, excluding the terminating null byte (`'\0'`). The input pointer is not `NULL`. You must respect the expected behavior of the function `strlen(3)`.

Exemple 16.1:
```
shell> ./question16 0 5
s = "n{6\P"
mystrlen(s) = 5 -- check: [OK]
```

Question 17 (1)

Write the following function(s):

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

`mystrncpy(dst,src,len)` : function copies at most `len` bytes of the string pointed to by `src` to the buffer pointed to by `dst`. The strings may not overlap, and the destination string `dst` must be large enough to receive the copy. If there is no null byte among the first `len` bytes of `src`, the string placed in `dst` will not be null-terminated. If the length of `src` is less than `len`, `mystrncpy()` writes additional null bytes to `dst` to ensure that a total of `len` bytes are written.

Note: `mystrncpy()` **always** writes exactly `len` bytes, whatever is the length of `src`.

You should read carefully the manual page of `strncpy(3)` which provides a complete description of the expected function.

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]
```

About The Test Session

Once the test is over, you must leave your session by closing the clock (that's the only way.) Note that when the test is over, your session will close directly.

When the session closed, you'll be prompted for your password (the one used to login.) This will end the test (your *submission* directory will be archived and sent to the collecting server.) **You must not shutdown the computer before the completion of this final step, otherwise your work will be lost.**

You can send intermediary versions of your test by using the shell command `submission`. It is strongly advised that you do so to prevent data lost before the end of the test.

Even after the end of the test (in the few minutes following the test, of course), you can restart your computer to eventually re-send your work (this may be required sometimes if something goes wrong during the final step.)

About Questions Skel

For every question, a skeleton of code is provided. This code is the minimal requirement for the compilation of the file *w.r.t.* the test program. The content of the skeleton will also induce a failure at execution time and thus you must remove the the body of the function(s). In C, be sure to remove (or comment) the `REMOVE_ME()` line of code: if it's still in the file, it will probably be executed anyway.

Exemple 1:
For example, if you're asked for the following function:

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

`identity` *x returns x.*
You'll find the following skeleton:

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

Your answer will look like:

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

About test programs

When invoked with `make questionXX`, `make` will build a binary program named `questionXX`. This program can be used to test your answer to the question X in this subject. All binary wait for parameters et display a small help when run with `-help`.

Exemple 2:
For example, the program for question 1 (this is an example and may not corresponds to the actual question 1) will display:

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

The two parameters are thus: `graine` (seed) and `taille` (size). These parameters are present in most question: the seed is used to initilize the random number generator (for a given seed, the generator will produce the same sequence of number) and the size can be either the size of generated data (for list or strings . . .) or the number of tested . . .

If you need more detail, read the `test_qXX.c` files.