EPITA S3 promo 2018
Practical Programming - Machine Test

Marwan Burelle*

Tuesday, December 23, 2014

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* `rendu`*. In the* `subject` *directory, you'll find a sub-directory called* `Skel`*, you have to copy the* ***content*** *of this directory in your* `rendu` *directory.*

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

Here is given as example, commands to perfom the needed copy from `subject/Skel` *directory to* `rendu` *directory:*

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

In this 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****.*

Scale information are only indicative and may be changed later.

At the end of the document, you'll find the extra sections providing:

- *Advices about test programs;*
- *List of all files that* **must** *be in your* `rendu` *directory.*

There are 24 points and 13 questions in this test.

*marwan.burelle@lse.epita.fr

Question 1 (1)

Write the following function(s):

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

`msb(n)` return the rank (starting from 1) of the most significant bit (corresponding also to the minimal number of bits required to represent the value.) In order to compute the `msb`, the simplest method is to divide the number by 2 until 0 and to count the number of pass.

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)

Write the following function(s):

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

`bin_string(n, output)` fill the string `output` with the value of each bits in the integer `n`. The most significant bits (even if equal to 0) will be at position 0. The array `output` has the necessary size in ordre to store all the bits of n. The size can be compute with the expression `(8 * sizeof (unsigned long))`.

To obtain the value of the least significant bit of a number, you just need to use the modulus with 2 (`n % 2`) and to obtain a numeric character, you just need to add the digit to the character `'0'` (for example the character `'1'` is obtained with `1 + '0'`.)

Exemple 2.1:

```
shell> ./question02 0 5
Fixed test:
  bin_string( 1) = "00000000000000000000000000000000000000000000000000000000000000001" (should be 1)
  bin_string( 2) = "00000000000000000000000000000000000000000000000000000000000000010" (should be 10)
  bin_string( 3) = "00000000000000000000000000000000000000000000000000000000000000011" (should be 11)
  bin_string( 4) = "00000000000000000000000000000000000000000000000000000000000000100" (should be 100)
  bin_string(42) = "00000000000000000000000000000000000000000000000000000000000101010" (should be 101010)
Random test:
  bin_string(15498727785856967622) = "1101011100010110100010101100111000110010011110110010001111000110"
```

```
bin_string(14445630959983478899) = "1100100000111100100110000110100100110011000110011010010000011100011"
bin_string(16816925489926593791) = "1110100101100000110111000010100010000110010100100101011100111111111"
bin_string( 6183768381126433004) = "0101010111010001001010001001010001100010010101010101100011101100"
bin_string( 5124038999138663629) = "0100011100011100001111100101001001000110111010000111110011001101"
```

Question 3 (1)

Write the following function(s):

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

`read_bin(input)` read an integer in binary from the string `input`. The string contains all the bits of an integer of type `unsigned long`, so its length is `8 * sizeof (unsigned long)` (like the previous exercise.) Like in the previous exercise, the most significant bit will be at position 0.

Exemple 3.1:

```
shell> ./question03 0 5
Fixed test:
  read_bin(00000000000000000000000000000000000000000000000000000000000000001) = 1 (should be 1)
  read_bin(00000000000000000000000000000000000000000000000000000000000000010) = 2 (should be 2)
  read_bin(00000000000000000000000000000000000000000000000000000000000000011) = 3 (should be 3)
  read_bin(00000000000000000000000000000000000000000000000000000000000000100) = 4 (should be 4)
  read_bin(00000000000000000000000000000000000000000000000000000000000101010) = 42 (should be 42)
Random test:
  read_bin(10111100110101100000101100011110001110101111010010101001000011101) = 13607075548612569373
  read_bin(01011101010100101000001101000010000110001110100010011101011111111) = 6724581513526549887
  read_bin(11101101011001011100010101010111100010100000101111100110110101111) = 17106295739405559511
  read_bin(11001101001111000000011100101010100110010001110010011000010111001) = 14787585135548641465
  read_bin(00000100000100000011001111010011110111001000000000110101000110001) = 292847945840093745
```

Question 4 (2)

Write the following function(s):

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

`binsearch(begin, end, x)` search the value `x` in the array starting at `begin` and ending at `end` (excluded.) The array is sorted in increasing order et you must use the binary search method to look for the value (warning: if you use a slower method, you won't have all the points due to execution time-outs.) The function must return the pointer to the array cell containing the value, or `NULL` if the value is not found.

Here is a quick overview of the binary search method:

```
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)

Write the following function(s):

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

`quick_sort(left, right)` sort the array of integer starting at `left` and ending at `right` (excluded.) Like the previous exercise, you must use the method described in the subject to avoid slower algorithms.

```
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)

Write the following function(s):

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

`insert(l, x)` insert in the sorted linked list `l` the value `x` at its place only if the value is not yet present, otherwise (when `x` is present) the function does nothing.

The list type is the following:

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

The lists given to the function will always start with a sentinel (thus, the pointer will never be `NULL` even for the empty list.) This sentinel must be unchanged and won't interfer with the oder of the list (its value is unspecified.)

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)

Write the following function(s):

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

`circular_list_mem(l, x)` returns true (1) if `x` belongs to the circular list `l` and false (0) otherwise.

The used lists have the same type as the one of the previous exercise. But, beware: theses lists don't have a sentinel and are **circular** ! You must take care when iterating on it to

avoid searching for the non-existant end of the list and rather stop when you encounter the head for the second time. The empty list is represented with the `NULL` pointer, don't forget to test that case.

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)

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 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)

Write the following function(s):

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

`my_strcmp(s1, s2)` compare the two null-terminated (terminated with `'\0'`) string `s1` and `s2`. The function will return 0 if both strings are equals, a negative value if the first one is *smaller* and a positive one in the other case.

Comparing 2 strings follow the rule of a lexicographic order: we compare the 2 strings character by character, if the characters are equals we move to the next one, if they differ the smallest character indicates the smallest string, and if one of the strings finished before the other it will be the smallest one.

You can return the difference of the two first characters that are not equals in the two string (for example, for the strings `"aa"` and `"ad"` you can return -3.)

You can read the manual page (`strcmp(3)`) for more 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)

Write the following function(s):

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

`height(t)` returns the height of the binary tree `t`. The height is defined recursively has: -1 for the empty tree and 1 plus the maximal height of the two children.

Trees are represented by the following type, empty tree are `NULL` pointer:

```
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)

Write the following function(s):

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

`prefix_list(t, keys)` fill the array `keys` with the keys of the binary tree `t` in the prefix order of encounter of a depth first traversal (left first.) The function will also returns the number of node encountered during the traversal.

You should have notice that there is no counter for position in the array `keys`: you will need, for recursive calls in the DFS, to pass the array shifted by the number of element from the current node (1) with also the number of element encounter during the traversal of the left child when doing the call on the right child.

We use the same type as the previous exercise.

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 |
```

In order to simplify the test during the exam, the nodes are numbered so that they are in the prefix order, but for the real evaluation the correcting program will use random value.

Question 12 (3)

Write the following function(s):

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

`is_white(img)` return true (1) if the image is totally white (see the description) and false (0) otherwise.

The images have the following type:

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

These images are represented using *quad-tree*: an image is either black (the pointer is `NULL`), or either white (the pointer is not `NULL` and the field `white` is true) or is composed of four images (the four pointers in the 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)

Write the following function(s):

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

`decipher(msg, len, key, keylen)` decipher the message in `msg` of length `len` with the key `key` of length `keylen`. The deciphered text will take the place of the original text.

The cipher algorithm used is *Végenaire*, it's a simple algorithm, the deciphering is done this way: for each character of the message, we get the deciphered character with:

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

Beware: in C, arithmetic computation are done on `int`, you'll need to enforce a `char` type for the result (use *casts*.)

Exemple 13.1:
Christmas exercise: you'll see the result by your self.

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 *rendu* 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 `rendu`. 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.

Lists of Files

Immutable Files

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

Answer Files

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