(*Adapted to Source of AY2019/2020 Semester 1*)

CS1101S — Programming Methodology

Semester 1, 2018/2019

Practical Assessment

Date: 15 November 2018 **Time Allowed:** 1 Hour 45 Minutes

Instructions (please read carefully):

1. This question booklet comprises **9 printed pages** and has **3 questions** with a total of **50 marks**. Answer all questions.
2. This is an **open book** assessment. Any written or printed material, or programs stored on the Source Academy may be used as reference material. You are also allowed access to the Lecture slides, Reflection sheets, Studio sheets and Source documentation stored at [deleted].
3. The **internet** must not be used, except the **Source Academy** site.
4. Apart from the lab computer assigned to you, you are not allowed to use any other **electronic devices**.
5. All programs should be written in **Source §4**, unless otherwise stated.
6. The questions in this paper should be answered and submitted on the Source Academy. Please go to **Missions → Practical Assessment (Session *X*)** to attempt the questions.
7. Remember to **save frequently**, especially before running programs.
8. You may **finalize submission** when you have completed the practical assessment. Note that this is irreversible. If the submission is not finalized by the deadline, the **last saved** version will automatically be submitted.
9. The `assert` function calls in the solution templates provide sample **test cases** to check the correctness of your solutions. You may view the test results in the REPL. **Note that the given test cases are not exhaustive; passing them does not mean your solution is correct**. You are strongly encouraged to add your own test cases.
10. The solutions of some tasks require correct solutions of previous tasks. In the environment that we set up for you, **you can program and test your solutions without dependencies**. This is achieved by the `assert` function. Each time `assert` is called, our correct implementation of the solutions to all relevant previous tasks is installed in the global environment. We therefore strongly encourage you to **test your programs only using `assert`**.
11. If you are writing your own **helper functions**, they must be declared **in the body** of the function that you are writing for the task.
12. Do not leave the room until you are told to do so, even after you have finalized the submission.

GOOD LUCK!

Question 1: Big-Integers [25 marks]

In this question, we consider the use of a ***Big-Integer*** data structure to represent **non-negative integers**. The *big-integer* representation of a non-negative integer *n* is a list of the decimal digits of *n* (each digit is a Source number ranging from 0 to 9), where the least significant digit (LSD) is the first element of the list, and the most significant digit (MSD) is the last element of the list. For example, the integer 903 is represented as the big-integer `list(3,0,9)`.

The integer 0 (zero) is represented as `list(0)`, but **leading zeros are not allowed** in all other big-integers. For example, the integer 903 cannot be represented as `list(3,0,9,0)`.

Note that big-integers can represent integers with hundreds and thousands of digits, which is beyond what Source numbers can represent.

Task 1A. `make_big_int_from_number` [4 marks]

Write a function `make_big_int_from_number(num)` that takes a non-negative integer `num` (a Source number), and returns a big-integer representation of `num`. You can assume that `num` is within the range that can be precisely represented in Source.

Examples:

```
make_big_int_from_number(0);
// returns list(0)

make_big_int_from_number(1234);
// returns list(4, 3, 2, 1)
```

Task 1B. `big_int_to_string` [2 marks]

Write a function `big_int_to_string(bint)` that takes a big-integer `bint` and returns a string that shows the represented integer. The input big-integer `bint` must not be modified.

Examples:

```
big_int_to_string(list(0));
// returns "0"

big_int_to_string(list(0, 0, 3, 2, 1, 8, 8, 8));
// returns "88812300"
```

Task 1C. `big_int_add` [5 marks]

Write a function `big_int_add(bintX, bintY)` that takes big-integers `bintX` and `bintY`, and returns the big-integer that represents the **sum** of the integers represented by `bintX` and `bintY`. The input big-integers `bintX` and `bintY` must not be modified.

Examples:

```
big_int_add(list(0), list(3, 2, 1));
// returns list(3, 2, 1)

big_int_add(list(7, 8, 9), list(5, 6));
// returns list(2, 5, 0, 1)      (because 987 + 65 = 1052)

big_int_add(list(9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9), list(5));
// returns  list(4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1)
```

Task 1D. `big_int_mult_by_digit` [5 marks]

Write a function `big_int_mult_by_digit(bint, digit)` that takes a big-integer `bint` and a integer `digit` (a Source number) ranging from 0 to 9, and returns the big-integer that represents the **product** of `digit` and the integer represented by `bint`. The input big-integer `bint` must not be modified.

Examples:

```
big_int_mult_by_digit(list(0), 5);
// returns list(0)

big_int_mult_by_digit(list(7, 4, 3), 0);
// returns list(0)

big_int_mult_by_digit(list(7, 4, 3), 5);
// returns list(5, 3, 7, 1)      (because 347 * 5 = 1735)

big_int_mult_by_digit(list(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,9), 3);
// returns            list(3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,7,2)
```

Task 1E. `big_int_mult_by_10_pow_n` [3 marks]

Write a function `big_int_mult_by_10_pow_n(bint, n)` that takes a big-integer `bint` and a non-negative integer `n` (a Source number), and returns the big-integer that represents the **product** of `10`^{n} and the integer represented by `bint`. The input big-integer `bint` must not be modified.

Examples:

```
big_int_mult_by_10_pow_n(list(0), 5);
// returns list(0)

big_int_mult_by_10_pow_n(list(7, 4, 3), 0);
// returns list(7, 4, 3)      (because 347 * 10⁰ = 347 * 1 = 347)

big_int_mult_by_10_pow_n(list(7, 4, 3), 3);
// returns list(0, 0, 0, 7, 4, 3)      (because 347 * 10³ = 347000)

big_int_mult_by_10_pow_n(list(7, 4, 3), 20);
// returns list(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,4,3)
```

Task 1F. `big_int_mult` [6 marks]

Write a function `big_int_mult(bintX, bintY)` that takes big-integers `bintX` and `bintY`, and returns the big-integer that represents the **product** of the integers represented by `bintX` and `bintY`. The input big-integers `bintX` and `bintY` must not be modified. Your function may call functions for the preceding tasks.

The product of two integers can be computed by using the method of *long multiplication*. For example, using long multiplication, the product of 1234 and 567 is computed as

$$1234 \times 5 \times 10^2 \ + \ 1234 \times 6 \times 10^1 \ + \ 1234 \times 7 \times 10^0.$$

Examples:

```
big_int_mult(list(0), list(3, 2, 1));
// returns list(0)     (because 0 * 123 = 0)

big_int_mult(list(9), list(6));
// returns list(4, 5)      (because 9 * 6 = 54)

big_int_mult(list(7, 8, 9), list(5, 6));
// returns list(5, 5, 1, 4, 6)      (because 987 * 65 = 64155)

big_int_mult(list(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1), list(7,8,9));
// returns   list(7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,8,9)
```

Question 2: Arranging the Digits [13 marks]

For the following tasks, your function may call the following functions provided in the solution template:

- `swap(A, i, j)`
- `copy_array(A)`
- `reverse_array(A)`
- `array_to_list(A)`
- `list_to_array(L)`
- `sort_ascending(A)`
- `digits_to_string(digits)`

Task 2A. `build_largest_int` [3 marks]

Write a function `build_largest_int(digits)` that takes as argument `digits`, which is an **array** of **non-zero** decimal digits (each digit is a Source number ranging from **1** to **9**), and returns a **string** that shows the **largest possible integer** that can be formed by rearranging the given digits. Each given digit must be used exactly once. The input array `digits` has at least one element, and it must not be modified by your function.

Examples:

```
build_largest_int([4, 1, 9, 4, 1]);
// returns "94411"

build_largest_int([5, 5, 5]);
// returns "555"
```

Task 2B. `build_2nd_largest_int` [5 marks]

Write a function `build_2nd_largest_int(digits)` that takes as argument `digits`, which is an **array** of **non-zero** decimal digits (each digit is a Source number ranging from **1** to **9**), and returns a **string** that shows the **second largest integer** that can be formed by rearranging the given digits. Each given digit must be used exactly once. If the second largest integer does not exist, your function should just return a string for the smallest possible integer. The input array `digits` has at least one element, and it must not be modified by your function.

Examples:

```
build_2nd_largest_int([4, 1, 9, 4, 1]);
// returns "94141"

build_2nd_largest_int([5, 5, 5]);
// returns "555"
```

Task 2C. `build_nth_largest_int` [5 marks]

Write a function `build_nth_largest_int(digits, n)` that takes as arguments `digits`, which is an **array** of **non-duplicate** and **non-zero** decimal digits (each digit is a Source number ranging from **1** to **9**), and a positive integer number `n`, and returns a **string** that shows the **`n`-th largest integer** that can be formed by rearranging the given digits. Each given digit must be used exactly once. If the `n`-th largest integer does not exist, your function should return a string for the smallest possible integer. The input array `digits` has at least one element, and it must not be modified by your function.

Examples:

```
build_nth_largest_int([3, 1, 4, 2], 1);
// returns "4321"

build_nth_largest_int([3, 1, 4, 2], 2);
// returns "4312"

build_nth_largest_int([3, 1, 4, 2], 10);
// returns "3214"

build_nth_largest_int([3, 1, 4, 2], 24);
// returns "1234"

build_nth_largest_int([3, 1, 4, 2], 28);
// returns "1234"
```

Question 3: Terrain Elevation Maps [12 marks]

A rectangular *terrain elevation map* is drawn on a *R*×*C* grid, where *R* is the number of rows and *C* the number of columns. Each grid cell has an elevation value, which is the height of the terrain at that grid cell.

We represent an elevation map as a "**2D array**" of numbers (each such 2D array is actually an array of arrays of numbers in Source). For example, the elevation map

1	0	2	4	3
2	0	0	2	2
2	1	0	0	1

is represented as the following 2D array:

```
[ [1, 0, 2, 4, 3],
  [2, 0, 0, 2, 2],
  [2, 1, 0, 0, 1] ]
```

Task 3A. [7 marks]

In an elevation map, a cell *C* is a ***peak*** if *all* its **8 surrounding/neighboring cells** have **strictly lower** elevation than that of *C*. Note that a cell on the edge of an elevation map cannot be a peak, since it has fewer than 8 neighboring cells.

Task 3A(I). `count_lower_neighbors` [4 marks]

Write a function `count_lower_neighbors(emap, r, c)` that takes an `emap`, and integers `r` and `c`, and returns the **number of neighbors** of `emap[r][c]` that **have strictly lower** elevation than `emap[r][c]`. If (`r`, `c`) is a cell location on the edge or outside `emap`, then the function returns 0. The input `emap` is at least 1×1 in size. The input array must not be modified by your function.

Examples:

```
const emap = [[3, 1, 1, 1, 1, 1, 1],
              [1, 1, 1, 1, 2, 3, 1],
              [1, 0, 3, 2, 1, 1, 0],
              [1, 1, 1, 1, 3, 1, 1],
              [1, 2, 1, 1, 3, 1, 3],
              [1, 1, 1, 1, 4, 1, 1]];

count_lower_neighbors(emap, 0, 0);  // returns 0
count_lower_neighbors(emap, 5, 4);  // returns 0
count_lower_neighbors(emap, 1, 1);  // returns 1
count_lower_neighbors(emap, 2, 2);  // returns 8
count_lower_neighbors(emap, 2, 3);  // returns 5
```

Task 3A(II). `count_peaks` [3 marks]

Write a function `count_peaks(emap)` that returns the **number of peaks** in `emap`. The input `emap` is at least 1×1 in size. The input array must not be modified by your function. Your function may call the function for the preceding task.

Example:

```
const emap = [[3, 1, 1, 1, 1, 1, 1],
              [1, 1, 1, 1, 2, 3, 1],
              [1, 0, 3, 2, 1, 1, 0],
              [1, 1, 1, 1, 3, 1, 1],
              [1, 2, 1, 1, 3, 1, 3],
              [1, 1, 1, 1, 4, 1, 1]];

count_peaks(emap);
// returns 3
```

Task 3B. `count_islands` [5 marks]

Consider an elevation map `emap`, in which the elevation values are **non-negative**. An elevation value of **0** indicates **water** (e.g. sea) and **non-zero** elevation values indicate **land**. We want to find the number of **groups of connected non-zero elements** in `emap`. Each of such groups is called an ***island***.

For example, given the following 6×7 `emap`:

```
[[2, 1, 0, 2, 1, 1, 3],
 [0, 1, 0, 1, 0, 0, 2],
 [0, 0, 0, 2, 3, 1, 1],
 [1, 0, 2, 0, 0, 0, 0],
 [0, 0, 1, 2, 0, 0, 0],
 [1, 0, 3, 0, 1, 1, 2]]
```

There are 6 islands, as shown by the 6 shaded regions:

```
[[2, 1, 0, 2, 1, 1, 3],
 [0, 1, 0, 1, 0, 0, 2],
 [0, 0, 0, 2, 3, 1, 1],
 [1, 0, 2, 0, 0, 0, 0],
 [0, 0, 1, 2, 0, 0, 0],
 [1, 0, 3, 0, 1, 1, 2]]
```

Note that two non-zero elements are **connected** to each other only if one is immediately on the **left, right, below, or above** the other (i.e. they do not connect to each other "diagonally").

Write a function `count_islands(emap)` that returns the **number of islands** found in `emap`. The input `emap` is at least 1×1 in size. The input array must not be modified by your function.

Example:

```
const emap = [ [1, 2, 0, 0, 1, 0, 0, 1],
               [1, 2, 2, 3, 1, 0, 2, 1],
               [0, 1, 1, 0, 1, 0, 0, 1],
               [0, 0, 0, 0, 0, 3, 3, 0],
               [1, 1, 2, 0, 0, 0, 0, 0],
               [1, 0, 1, 0, 0, 1, 2, 3],
               [1, 3, 2, 1, 1, 0, 1, 1] ];

count_islands(emap);
// returns 5
```

——— END OF QUESTIONS ———