NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

Practical Examination 2 (PE2) for Semester 1, AY2016/7
CS1010 Programming Methodology

22 October 2016 Time Allowed: 2 hours

INSTRUCTION TO CANDIDATES

1. You are only allowed to read this cover page and the last page **(page 8)**. Do **not** read the questions until you are told to do so.
2. This paper consists of **2** exercises on **8** pages. Each exercise constitutes 50%. The exercises have different levels of difficulty. Therefore, beware of time spent on each exercise.
3. This is an open-book exam. You may bring in any printed material, but **not** electronic devices, including but not limited to laptop, thumb-drive, electronic dictionary and calculator. You are to switch off/silence your mobile phone and keep it out of view.
4. You may turn to the last page to read some advice.
5. You will be logged into a special Windows account at the beginning of the PE. Do not log off until the end of the PE. Do not use your own NUSNET account.
6. A plab account slip will be issued to you at the beginning of the PE. Bring your matriculation card for identification when you collect it. Please leave your matriculation card on the desk in front of you throughout the PE.
7. You are to write your program in the given **plab account**. The host name is **plab4** (not sunfire!). No activity should be done outside this plab account.
8. You do **not** need to submit your programs to CodeCrunch. We will retrieve your programs and submit them to CodeCrunch after the PE.
9. Skeleton programs **(.c files)** and some test data files **(.in and .out files)** are given in your plab account. **Write your solutions in these two programs**. Do **not** change the file names, or put your programs into a subdirectory; otherwise we will **not** be able to find them!
10. **Only your source code files (.c files)** will be collected after the PE. Hence, how you name your executable files is not important.
11. Read carefully and follow all instructions in the question. If in doubt, raise your hand and an invigilator will attend to you.
12. Any form of communication with other students or the use of unauthorised materials is considered cheating and you are liable to disciplinary action.
13. Save your programs regularly during the PE.
14. When you are told to stop, please do so **immediately**, or you will be penalised.
15. At the end of the PE, please **log out from your plab account**.
16. Please check and take your belongings (especially matriculation card) before you leave.
17. We will make arrangement for you to retrieve your programs after we have finished grading. Grading may take a week or more.

ALL THE BEST!

Exercise 1: Spy! [50 marks]

Problem Statement

Mr. Mean is a spy who has been secretly gathering confidential information in various countries and sending it back to his home country.

Since it is dangerous to send the information as a plain message, every day he uses a different keyword given by his country to encrypt the message to be sent.

The encryption process consists of three steps:

Step 1: *Keyword Simplification*. For a given keyword, only the first occurrence of a letter is kept. All subsequent occurrences of the same letter (if any) are removed.

For example, let say the keyword is *mississippi*. Only the first occurrence of 'm' (1st letter), 'i' (2nd letter), 's' (3rd letter) and 'p' (9th letter) are kept. The rest of the letters are removed and the simplified keyword is *misp*. Note that the order of the 4 remaining letters stays the same as in the original keyword.

Step 2: *Alphabet creation*. The letters in the original (English) alphabet are rearranged to form a new alphabet. This new alphabet starts with the letters in the keyword (in the same order as they are in the keyword), followed by the letters which are missing from the keyword (in the original alphabetical order).

For example, if the simplified keyword is *misp*, the original and new alphabets are as shown below:

Original (English) alphabet:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

New alphabet created based on the keyword *misp*:

m	**i**	**s**	**p**	a	b	c	d	e	f	g	h	j	k	l	n	o	q	r	t	u	v	w	x	y	z

As shown in the new alphabet, the keyword *misp* appears as the first 4 letters in the new alphabet. This is followed by the rest of the letters in the original alphabetical order ('a' to 'z' but without 'i', 'm', 'p' and 's' since they have appeared in the keyword).

Step 3: Encryption. The letters in the message are replaced based on the new alphabet: A letter which is the i-th letter in the original alphabet is replaced with the i-th letter in the new alphabet, while the case of the letter remains unchanged.

For example, if the message is *SuperSpy*, the uppercase letter 'S', which the 19th letter in the original alphabet, is replaced by the uppercase letter 'R', which is the 19th letter in the new alphabet. In contrast, the lowercase letter 'u' remains unchanged since it is the 21st letter in both alphabets. The resulting encrypted message is *RunaqRny*.

To help Mr. Mean encrypt any message based on any keyword quickly, you are to implement this encryption process as a program.

Write on the skeleton file **spy.c** given to you. You should include the following three functions in your program:

- **void simplify(char keyword[])**

  This function takes in a char array **keyword** which contains the given keyword as a string. It performs keyword simplification and returns the simplified keyword as a string via the same array.

- **void createAlphabet(char keyword[], char alphabet[])**

  This function takes in a string **keyword** which is the simplified keyword. It creates the new alphabet and returns it as a string via the char array **alphabet**.

- **void encrypt(char message[], char alphabet[], char result[])**

  This function takes in a string **message**, which is the message to be encrypted, and a string **alphabet**, which represents the new alphabet. It encrypts the message based on the new alphabet and returns encrypted message as a string via the char array **result**.

You are **not** allowed to change the **main()** function. You may assume that 1) both the keyword and the message are at most 80 characters long and 2) the keyword contains only lowercase letters, while the message contains uppercase and/or lowercase letters. Check sample runs for input and output format.

(Hint 1: As a start, you may want to skip Step 1 completely since not all keywords contain duplicate letters. You may come back to implement Step 1 after your program is able to complete Step 2 and Step 3 given a keyword without duplicate letters.)

(Hint 2: There is no need to explicitly store the original alphabet in Step 2 and Step 3.)

Sample Runs

Three sample runs are shown below with user input highlighted in **bold**.

```
Enter keyword: z
Enter message: hello
Encrypted message: gdkkn
```

```
Enter keyword: mean
Enter message: hello
Encrypted message: fbjjo
```

```
Enter keyword: mississippi
Enter message: SuperSpy
Encrypted message: RunaqRny
```

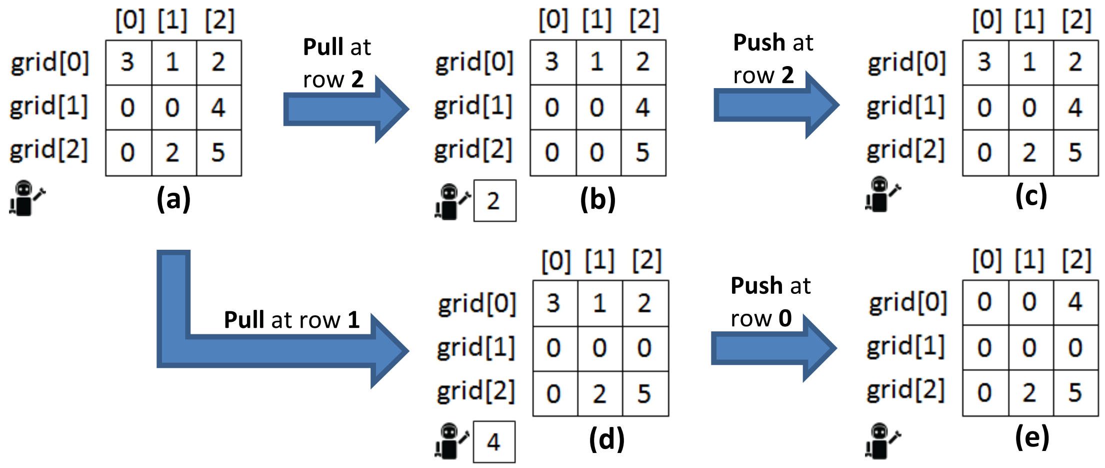
Exercise 2: Block Buster [50 marks]

Problem Statement

Block Buster is a game in which a robot retrieves blocks from a grid and uses them to destroy other blocks in the same grid.

A grid in this game consists of ***n*** x ***n*** cells (where ***n*** is a positive integer). Each cell may be empty (represented by ***0***), or contain a block of weight ***k*** (represented by a positive integer ***k***). No empty cell may appear on the right of a block at the same row.

For example, in grid **(a)**, there are 3 empty cells (*e.g.*, [1][0]) and 6 blocks of various weights (*e.g.*, a block of weight 3 at [0][0]), as shown below.



	[0]	[1]	[2]
grid[0]	3	1	2
grid[1]	0	0	4
grid[2]	0	2	5

(a)

	[0]	[1]	[2]
grid[0]	3	1	2
grid[1]	0	0	4
grid[2]	0	0	5

(b)

	[0]	[1]	[2]
grid[0]	3	1	2
grid[1]	0	0	4
grid[2]	0	2	5

(c)

	[0]	[1]	[2]
grid[0]	3	1	2
grid[1]	0	0	0
grid[2]	0	2	5

(d)

	[0]	[1]	[2]
grid[0]	0	0	4
grid[1]	0	0	0
grid[2]	0	2	5

(e)

Given a grid, the robot performs a sequence of actions of two types: **Pull** and **Push**. The actions in a sequence are in an alternating manner starting from a **Pull** (*i.e.*, Pull → Push → Pull → Push → …). A sequence may end with either action. For each action, a row index ***i*** is specified to indicate at which row this action is performed.

For a **Pull** at row ***i***: The robot pulls the leftmost block at row ***i*** and holds it.

For example, after a **Pull** at row **2** of grid **(a)**, the robot holds the block from [2][1] (*i.e.*, the leftmost block at row 2), as shown in grid **(b)**. Similarly, after a **Pull** at row **1** of grid **(a)**, the robot holds the block from [1][2], as shown in grid **(d)**.

For a **Push** at row ***i***: The robot throws the block it is holding (*i.e.*, the block retrieved from the previous **Pull**) at row ***i***. The block travels in row ***i*** from the left towards the last column, and destroys all the blocks whose weight is less than its weight. The block stops on the left of the first block it cannot destroy, or at the last column if it has destroyed all the blocks in that row. For each block destroyed in this action, its weight is added to the total score. The robot no longer holds a block after this action.

For example, in grid **(b)**, the robot holds a block of weight 2 from the previous **Pull**. After a **Push** at row **2**, the block stops at [2][1] as shown in grid **(c)** since it cannot destroy the block at [2][2]. Since no block has been destroyed, the score remains unchanged. In contrast, in grid **(d)**, the robot holds a block of weight 4 from the previous **Pull**. After a **Push** at row **0**, the block stops at [0][2] as shown in grid **(e)** since it has destroyed all the blocks at that row. The total weight of the blocks destroyed (*i.e.*, 3 + 1 + 2 = 6) is added to the total score.

Note that since grid **(c)** happens to be the same as grid **(a)**, the five grids given above also form an example of performing a sequence of 4 actions (*i.e.*, **Pull** at row **2** → **Push** at row **2** → **Pull** at row **1** → **Push** at row **0**) on grid **(a)** and the result is grid **(e)**.

In addition, a **Pull** fails if there is no block at row ***i***, while a **Push** fails if row ***i*** is full of blocks and it cannot even destroy the leftmost block at that row.

A game is said to be **completed** if all the actions in the given sequence can be performed successfully; otherwise, as soon as one of the actions fails, the remaining actions are ignored and the game is said to be **uncompleted**.

In this exercise, you are to write a program to 1) perform a sequence of actions on a given grid, and 2) compute the total score and number of the blocks left in the grid.

Your program should read in an integer ***n*** (1 <= ***n*** <= 10), which is the size of the grid, as well as ***n*** x ***n*** values in the grid, which are either ***0*** (empty cell) or a positive integer ***k*** (a block of weight ***k***).

In addition, it should also read in an integer ***m*** (1 <= ***m*** <= 20), which is the number of row indexes for the actions in the given sequence, as well as ***m*** row indexes for the actions. For example, the sequence of 4 indexes *2 2 1 0* corresponds the sequence of actions which transforms grid **(a)** to grid **(e)**.

Your program should print a message indicating whether the game is **completed**. If the game is **completed**, it should also print the total score and the number of blocks left in the grid. Lastly, it should always print the final state of the grid.

Write on the skeleton file **block.c** given to you. You must include the following functions in your program:

- **readInputs()**

  This function reads in and returns 1) the size of the grid, 2) the values in the grid, 3) the number of row indexes for the actions, and 4) the row indexes.

- **play()**

  This function takes in the inputs returned from the **readInputs()** function. It performs the actions accordingly, and then returns 1) whether the game is **completed**, 2) the total score (if applicable), 3) the number of blocks left in the grid (if applicable), and 4) the final state of the grid.

You are to determine the return type and parameters for these functions. Check sample runs (on the next 2 pages) for input and output format.

Sample Runs

Five sample runs are shown below with user input highlighted in **bold**.

```
Enter size of grid: 1
Enter grid:
0
Enter number of indexes: 1
Enter indexes: 1
Unable to complete game.
Final state of the grid:
0
```

```
Enter size of grid: 3
Enter grid:
0 0 0
0 0 1
0 0 0
Enter number of indexes: 1
Enter indexes: 1
Game completed.
Score: 0
Number of blocks left: 0
Final state of the grid:
0 0 0
0 0 0
0 0 0
```

```
Enter size of grid: 3
Enter grid:
0 0 2
0 0 1
0 0 0
Enter number of indexes: 2
Enter indexes: 1 0
Game completed.
Score: 0
Number of blocks left: 2
Final state of the grid:
0 1 2
0 0 0
0 0 0
```

```
Enter size of grid: 3
Enter grid:
0 0 0
0 0 4
1 2 3
Enter number of indexes: 2
Enter indexes: 1 2
Game completed.
Score: 6
Number of blocks left: 1
Final state of the grid:
0 0 0
0 0 0
0 0 4
```

```
Enter size of grid: 3
Enter grid:
3 1 2
0 0 4
0 2 5
Enter number of indexes: 4
Enter indexes: 2 2 1 0
Game completed.
Score: 6
Number of blocks left: 3
Final state of the grid:
0 0 4
0 0 0
0 2 5
```

CS1010 AY2016/7 Semester 1
Practical Exam 2 (PE2)

Advice – Please read!

- You are advised to spend the first 10 minutes for each exercise thinking and designing your algorithm, instead of writing the programs right away.
- If you write a function, you must have a function prototype, and you must put the function definition after the **main()** function.
- You may write additional function(s) not mentioned in the question, if you think it is necessary.
- Any variable you use must be declared in some function. You are **not** allowed to use global variables (variables that are declared outside all the functions).
- You may assume that all inputs are valid, that is, you do not need to perform input validity check.
- Manage your time well! Do not spend excessive time on any exercise.
- Be careful in naming your executable code. Do **not** overwrite your source code with your executable code, especially if you are using the -o option in gcc!
- The rough marking scheme for both exercises is given below.

Rough Marking Scheme For Each Exercise [50 marks]

1. Style: 10 marks
   - Are name, matriculation number, plab-id, DG and description filled at the top of the program?
   - Is there a description written at the top of every function (apart from the **main()** function)?
   - Are there proper indentation and naming of variables?
   - Are there appropriate comments wherever necessary?
2. Design: 10 marks
   - Are there correct definition and use of functions?
   - Are function prototypes present?
   - Is the right construct used?
   - Is algorithm not unnecessarily complicated?
3. Correctness: 30 marks
4. Deductions (not restricted to the following):
   - Program cannot be compiled: Deduct 5 marks
   - Compiler issues warning with –Wall: Deduct 5 marks.
   - Use of global variables: Deduct 10 marks

--- END OF PAPER ---