**Lab #4: Arrays   
CS1010 AY2017/8 Semester 1   
Date of release: 22 September 2017, Friday, 8am.   
Submission deadline: 11 October 2017, Wednesday, 5pm.   
School of Computing, National University of Singapore**

**0 Introduction**

**Important:** Please read [Lab Guidelines](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/labguide.html) before you continue.

This lab consists of 3 exercises. The best 2 out of 3 exercises will be used to determine your attempt mark.

The main objective of this lab is on the use of arrays to solve problems.

The maximum number of submissions for each exercise is **12**.

If you have any questions on the task statements, you may post your queries **on the relevant IVLE discussion forum**. However, do **not** post your programs (partial or complete) on the forum before the deadline!

Important notes applicable to all exercises here:

* You should take the "Estimated Development Time" seriously and aim to complete your programming within that time. Use it to gauge whether your are within our expectation, so that you don't get surprised in your PE. We advise you to do the exercises here in a simulated test environment by timing yourself.
* Please do **not** use variable-length arrays. An example of a variable-length array is as follows:   
    int i;   
    int array[i];   
  This is not allowed in ANSI C, as explained in Unit #8 Arrays. Declare an array with a known maximum size. We will tell you the maximum number of elements in an array.
* Note that you are **NOT allowed to use recursion** for the exercises here. Hold on your "recursive streak" till a later lab. Using recursion here would amount to violating the objective of this lab assignment.
* You are **NOT allowed to use global variables**. (A global variable is one that is not declared in any function.)
* You are free to introduce additional functions if you deem it necessary. This must be supported by well-thought-out reasons, not a haphazard decision. By now, you should know that you **cannot write a program haphazardly**.
* In writing functions, we would like you to include function prototypes before the main function, and the function definitions after the main function.
* As mentioned in Unit #11 UNIX I/O Redirection (to be covered in week 7 but the slides are avaiable on the IVLE workbin and CS1010 website now), you may consider entering the input data in a file and then use UNIX input redirection to feed the data into your programs to test your programs.

**1 Exercise 1: Nanotable1**

**1.1 Learning objectives**

* Enhancing Nanotable0.
* Problem solving on two-dimensional array.

**1.2 Task statement**

This is an extension of an earlier lab Nanotable 0. Now with arrays, our Nanotable program can be enhanced. Tables of data can be stored as arrays.

Nanotable1 has the following features (new features highlighted in **!!**):

* The command parsing works as in Nanotable0, which is in a REPL style. Actually this part is given in the skeleton program already.
* When the input is "help", the help information is printed. Then the system waits for input again.
* When the input is "exit", the loop breaks and the system exits.
* **!!** When the input is "insert", the system asks for the input of "students ID" and "score". Then the (ID, score) entry will be stored in the system.
* **!!** When the input is "init", the whole table is emptied.
* **!!** When the input is "sum", if the table is empty, the system will say that the table is empty; otherwise it will print the sum of all the scores inserted so far. Then the system waits for input again.
* **!!** When the input is "ave", it is similar to the case of "sum" but the printed result will be the average of all the input scores.
* **!!** When the input is "rank", if the table is empty, the system will say that the table is empty; otherwise it will ask for input of an integer "rank", which should be from 1 to size (the number of existing entries in the table). If "rank" is out of this range, then print error information and ask for input of rank again. If "rank" is within this range, then print the ID and score of the entry with the rank-th entry in the table. (All the entries in the table are ordered in this way: entries with lower score have lower rank; for entries with same scores, those with smaller ID number have lower rank. It is assumed that all the inserted data has distince ID numbers.)

**1.3 Sample runs**

Sample runs using interactive input (user's input shown in blue). Note that the first two lines (in greenbelow) are commands issued to compile and run your program on UNIX.

You would need to use the **-lm** option in **gcc** if you use some math function in your program.

Sample run #1: (insert, sum, ave)

$ gcc -Wall -lm nanotable1.c -o nanotable1

$ nanotable1

Welcome to Nanotable!

Type "help" for more information...

Waiting for command...

insert

Please input the student's ID...

1

Please input the student's score...

90

Waiting for command...

insert

Please input the student's ID...

2

Please input the student's score...

80

Waiting for command...

insert

Please input the student's ID...

5

Please input the student's score...

100

Waiting for command...

sum

The sum of score is 270

Waiting for command...

ave

The average of score is 90

exit

See you!

Sample run #2: (insert, rank)

Welcome to Nanotable!

Type "help" for more information...

Waiting for command...

insert

Please input the student's ID...

1

Please input the student's score...

90

Waiting for command...

insert

Please input the student's ID...

2

Please input the student's score...

80

Waiting for command...

insert

Please input the student's ID...

3

Please input the student's score...

80

Waiting for command...

rank

Please input the rank (1 - 3)...

5

Invalid rank: 5

Please input the rank (1 - 3)...

1

ID: 2, Score: 80

Waiting for command...

rank

Please input the rank (1 - 3)...

2

ID: 3, Score: 80

Waiting for command...

rank

Please input the rank (1 - 3)...

3

ID: 1, Score: 90

Waiting for command...

exit

See you!

Sample run #3: (insert, init, sum)

Welcome to Nanotable!

Type "help" for more information...

Waiting for command...

insert

Please input the student's ID...

1

Please input the student's score...

100

Waiting for command...

sum

The sum of score is 100

Waiting for command...

init

Initializing table...

Waiting for command...

sum

The table is empty

Waiting for command...

exit

See you!

**1.4 Skeleton program and Test data**

* The skeleton program is provided here: [nanotable1.c](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex1/skeleton/nanotable1.c)
* You may develop your system based on that or you may ignore the skeleton code and write everything from scratch yourself.
* In the skeleton program, we have implemented this utility for you:
  + print\_help() - print the help information.
* Test data: [Input files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex1/testdata_for_students/input) | [Output files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex1/testdata_for_students/output)

**1.5 Important notes**

* You should implement a way to store a table. Each entry in this table contains two field: ID and score. The number of entries in the table will not exceed 100. (Text in red is added on 25 September.)
* You should at least implement functions insert\_table(), sum(), ave and rank(), whose parameters and return types are for you to decide.
* **Do not use any sorting functions in the library**. Any implementation that globally sorts an array is **not** allowed.

**1.6 Estimated development time**

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

* Devising and writing the algorithm (pseudo-code): 30 minutes
* Translating pseudo-code into code: 10 minutes
* Typing in the code: 10 minutes
* Testing and debugging: 10 minutes
* **Total: 1 hour**

**2 Exercise 2: Subsequence**

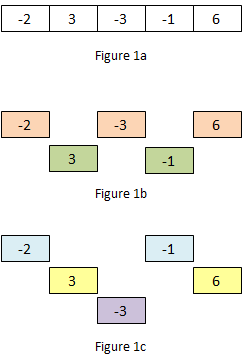
**2.1 Learning objectives**

* Problem solving on one-dimensional array.

**2.2 Task**

Given a list, a ***k*-interval subsequence** is a sublist where each element in the subsequence is *k*positions away from the next element in the subsequence.

For example, Figure 1a shows a list with 5 elements {-2, 3, -3, -1, 6}. Figure 1b shows the two 2-interval subsequences {-2, -3, 6} and {3, -1}. Figure 1c shows the three 3-interval subsequences {-2, -1}, {3, 6}, and {-3}. In general, there are *k* *k*-interval subsequences derivable from a list. You should have noticed that the 1-interval subsequence is the list itself.



Your task is to find the maximum sum of a *k*-interval subsequence among all *k*-interval subsequences. You are to report the solution subsequence: its sum, its interval *k*, and its starting position.

For the above example, all the *k*-interval subsequences and their respective sums are listed below:

* 1-interval subsequence
  + {-2, 3, -3, -1, 6}; sum = 3
* 2-interval subsequences
  + {-2, -3, 6}; sum = 1
  + {3, -1}; sum = 2
* 3-interval subsequences
  + {-2, -1}; sum = -3
  + **{3, 6}; sum = 9** // Solution
  + {-3}; sum = -3
* 4-interval subsequences
  + {-2, 6}; sum = 4
  + {3}; sum = 3
  + {-3}; sum = -3
  + {-1}; sum = -1
* 5-interval subsequences
  + {-2}; sum = -2
  + {3}; sum = 3
  + {-3}; sum = -3
  + {-1}; sum = -1
  + {6}; sum = 6

Hence, the solution is the 3-interval subsequence starting at position 1 with a sum of 9.

There might be more than one subsequence with the maximum sum. For example, given the list {1, 5, 5 -6, -2, -2}, there are two subsequences with the maximum sum of 5:

* A 4-interval subsequence starting at position 2: {5}
* A 5-interval subsequence starting at position 1: {5}

You should report the former as its value of *k* (the interval) is smaller.

A skeleton program is given. You are to complete the given function

void sum\_subsequence(int arr[], int size, int ans[])

You are **NOT** to change the function header given above, or marks will be deducted.

As the task is to find 3 values: the maximum sum, the interval, and the starting position, the ideal approach is to use address parameters or structure. However, we have not covered either of these topics yet. Hence, for the moment, we will use a 3-element integer array ans to hold the values, which are incidentally all integers (hence we can use an array):

* ans[0] = maximum sum of solution subsequence
* ans[1] = interval *k* of the solution subsequence
* ans[2] = start position of the solution subsequence

You may assume that there is **at least one element and at most 10 elements** in the list.

**2.3 Sample runs**

Sample runs using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

$ gcc -Wall subsequence.c -o subsequence

$ subsequence

Enter number of elements: 1

Enter 1 element: 123

**Max sum 123 of 1-interval subsequence starting at position 0.**

Second sample run:

$ subsequence

Enter number of elements: 5

Enter 5 elements: -2 3 -3 -1 6

**Max sum 9 of 3-interval subsequence starting at position 1.**

Third sample run:

$ subsequence

Enter number of elements: 6

Enter 6 elements: 1 5 5 -6 -2 -2

**Max sum 5 of 4-interval subsequence starting at position 2.**

**2.4 Skeleton program and Test data**

* The skeleton program is provided here: [subsequence.c](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex2/skeleton/subsequence.c)
* Test data: [Input files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex2/testdata_for_students/input) | [Output files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex2/testdata_for_students/output)

**2.5 Important notes**

* The list contains at least 1 element and at most 10 elements.
* Test your program thoroughly with different inputs.
* Ensure that the output of your program comforms to the output format. Note that if the number of elements is 1, the prompt to read the element reads "Enter 1 element: " and not "Enter 1 elements: ".

**2.6 Estimated development time**

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

This exercise is tedious to code. Please brace yourself for long hours of preparation, coding and testing!

* Devising and writing the algorithm (pseudo-code): 30 minutes
* Translating pseudo-code into code: 15 minutes
* Typing in the code: 15 minutes
* Testing and debugging: 30 minutes
* **Total: 1 hour 30 minutes**

**3 Exercise 3: Clash of the Frog Clans**

**3.1 Learning objectives**

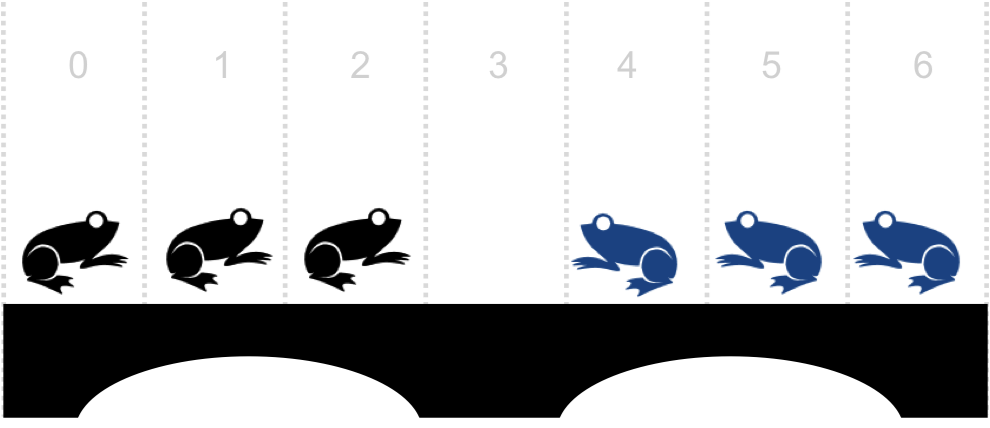
* Problem solving on one-dimensional array
* Breaking down the problem into subproblems
* Enumerate and program for all possible cases

**3.2 Task statement**

In the far far land of Maruku, there lives a population of optimistic frogs. Overly optimistic actually. Their life philosophy revolves around the idea of being forward looking. They believe that there is only one way in life, which is to move forward. Frogs who turn back are condemned in the society.

One fine day, two clans of frogs meet at the bridge of Marukunan River. The problem is, they are on different ends trying to cross the bridge. A nearby tourist commented, "Oh no, there's no way they can cross the bridge!" while another shouted, "Use your forward thinking to leap across all obstacles!".

The frogs stood there for a long time, thinking of a way to cross the bridge without dishonoring their family until a CS1010 student came by and said "Oh, I think they can cross it alright. It's just a matter of how many jumps will they need to take."

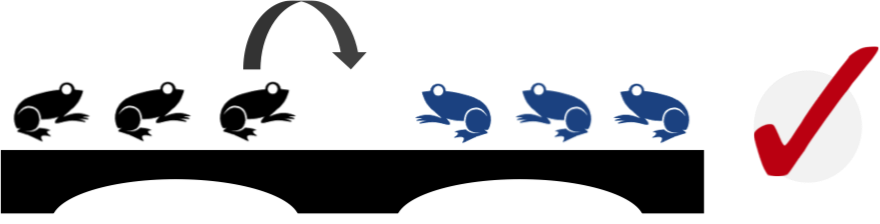
  
Figure 2. Frogs on the bridge

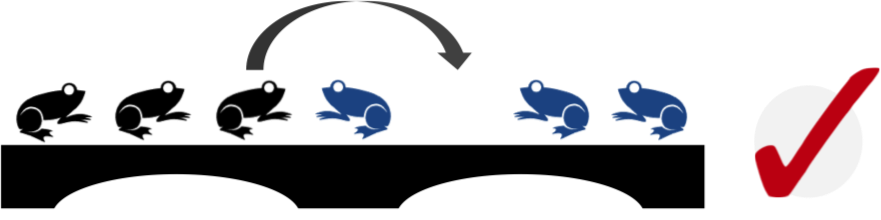
Write a program **frogs.c** to simulate a way for the frogs to cross the bridge. To simulate this, you are to write an integer array to represent the bridge and each element will represent the one frog or an empty space.

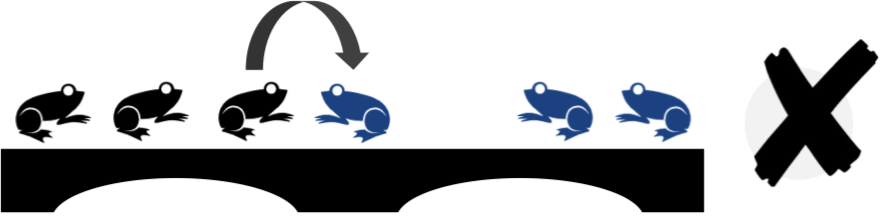
* Frogs cross towards the right side should be represented as **1**
* Frogs going to the left should be represented as **-1**
* An empty space should be represented as **0**.

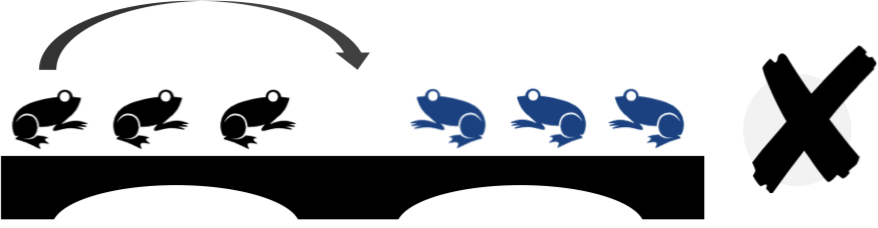
Each frog can only jump into the space in front of it, or jump over one frog into the space after that frog.

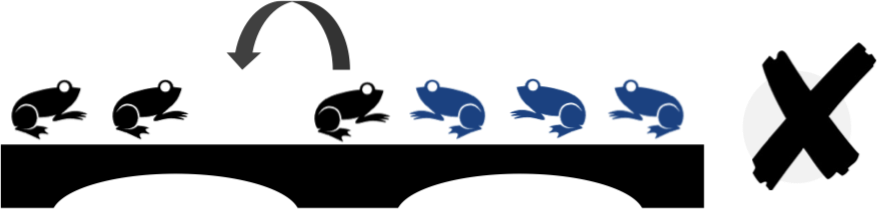
Your program should take in the length of a bridge (array), and the frogs and space (1, 0 or -1) at each position of the bridge. The user will then be able to move a frog to jump to the empty space directly in front of it OR to jump over a frog to the empty space behind the latter.

  
Figure 3-1. A frog can jump into empty space in front of it

  
Figure 3-2. A frog can jump over the frog in front of it to the empty space behind the latter

  
Figure 3-3. A frog can only jump into an empty space

  
Figure 3-4. A frog cannot jump so far!

  
Figure 3-5. A frog cannot jump backwards

Finally, your program should output when the game has ended. A game has ended if there is no possible jumps or all the frogs have successfully crossed the bridge. If they are successful, your program should output the number of jumps the frogs took to cross.

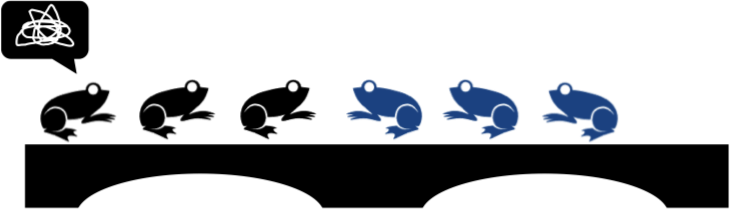
  
Figure 4-1. The frogs are stuck! Game over.

  
Figure 4-2. All the frogs have crossed the river!

You can be assured that the length of the bridge will not exceed 1000 and there is only one empty space on the bridge.

**3.3 Sample runs**

Sample run using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

Initialising the array:

$ gcc -Wall frogs.c -o frogs

$ frogs

Please enter the length of the bridge: 7

Please insert frog at position 0: 1

Please insert frog at position 1: 1

Please insert frog at position 2: 1

Please insert frog at position 3: 0

Please insert frog at position 4: -1

Please insert frog at position 5: -1

Please insert frog at position 6: -1

The above inputs represents *Figure 2*.

Write a function to print out the current state of bridge and its positions:

**Position: 0 1 2 3 4 5 6**

**Bridge: 1 1 1 0 -1 -1 -1**

Sample run (move frog)#1:

Move the frog at position: 2

**Position: 0 1 2 3 4 5 6**

**Bridge: 1 1 0 1 -1 -1 -1**

Frog at position 2 can jump into the empty space at position 3.

Sample run (move frog)#2:

Move the frog at position: 4

**Position: 0 1 2 3 4 5 6**

**Bridge: 1 1 -1 1 0 -1 -1**

Frog at position 4 can jump over frog at position 3 to the empty space in position 2.

Sample run (move frog)#3:

Move the frog at position: 1

**Invalid move!**

Frog at position 1 can only jump to the right, but there is no empty space.

Sample run (move frog)#4:

Move the frog at position: 4

**Invalid move!**

Position 4 is empty, therefore it's an invalid move.

The program comes to an end when all frogs have crossed the bridge or when there is no more possible moves. Sample outcome #1:

**Position: 0 1 2 3 4 5 6**

**Bridge: -1 -1 -1 0 1 1 1**

**Congratulations! The frogs took 15 jumps to cross the bridge!**

Sample outcome #2:

**Position: 0 1 2 3 4 5 6**

**Bridge: 1 1 -1 1 -1 -1 0**

**Oh no! It seems like the two clans of frogs are stuck!**

**3.4 Skeleton program and Test data**

* The skeleton program is provided here: [frogs.c](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex3/skeleton/frogs.c)
* Test data: [Input files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex3/testdata_for_students/input) | [Output files](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/ex3/testdata_for_students/output)

**3.5 Important notes**

* The length of the bridge is at most 1000 and there is only one empty space on the bridge.
* The function **printBridge()** is given on the skeleton program.
* We suggest that you write a function **jump()** to determine which position a given frog can jump to, and a function **checkGameState()** to check the state of the game (eg: whether it should end). You should determine the parameters of these functions.
* To promote Incremental Coding, you should type in these functions and test them out one at a time.
* In writing functions, we would like you to include function prototypes before the main() function, and the function definitions after the main() function.
* This is a problem-solving task where we look for **neat logic** in your program. Using descriptive variable names, and adding appropriate comments will help the readers (and yourself) to understand the logic better.

**3.6 Estimated development time**

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

* Devising and writing the algorithm move a frog (pseudo-code): 15 minutes
* Checking/tracing the algorithm: 5 minutes
* Translating pseudo-code into code: 5 minutes
* Typing in the code: 10 minutes
* Testing and debugging: 15 minutes
* Devising and writing the algorithm to check game state (pseudo-code): 15 minutes
* Checking/tracing the algorithm: 5 minutes
* Translating pseudo-code into code: 5 minutes
* Typing in the code: 10 minutes
* Testing and debugging: 15 minutes
* Devising and writing the algorithm for the main() function (pseudo-code): 15 minutes
* Checking/tracing the algorithm: 5 minutes
* Translating pseudo-code into code: 5 minutes
* Typing in the code: 10 minutes
* Testing and debugging: 15 minutes
* **Total: 2 hours 30 minutes**

**4 Deadline**

The deadline for submitting all programs is **11 October 2017, Wednesday, 5pm**. Late submission will NOT be accepted.

* [0 Introduction](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section0)
* [1 Exercise 1: Nanotable1](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1)
  + [1.1 Learning objectives](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_1)
  + [1.2 Task statement](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_2)
  + [1.3 Sample runs](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_3)
  + [1.4 Skeleton program and Test data](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_4)
  + [1.5 Important notes](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_5)
  + [1.6 Estimated development time](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section1_6)
* [2 Exercise 2: Subsequence](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2)
  + [2.1 Learning objectives](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_1)
  + [2.2 Task statement](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_2)
  + [2.3 Sample runs](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_3)
  + [2.4 Skeleton program and Test data](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_4)
  + [2.5 Important notes](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_5)
  + [2.6 Estimated development time](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section2_6)
* [3 Exercise 3: Clash of the Frog Clans](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3)
  + [3.1 Learning objectives](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_1)
  + [3.2 Task statement](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_2)
  + [3.3 Sample runs](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_3)
  + [3.4 Skeleton program and Test data](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_4)
  + [3.5 Important notes](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_5)
  + [3.6 Estimated development time](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section3_6)
* [4 Deadline](http://www.comp.nus.edu.sg/~cs1010/labs/2017s1/lab4/arrays.html#section4)

*Last updated: 1 October 2017*