Programming Session Assignment 04

By TA 林尚謙

1. Required files

You need to summit a .zip file named PSA04_b07901xxx.zip (your student ID) that contains the following files:

PSA04_p1.cpp

PSA04_p2.cpp

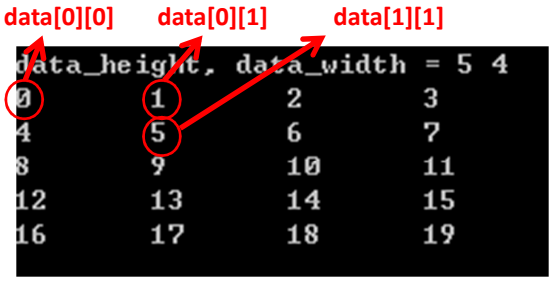
Please upload the .zip file to the CEIBA website by the deadline. Do not submit any executable files (.exe). Files with names in wrong format will not be graded.

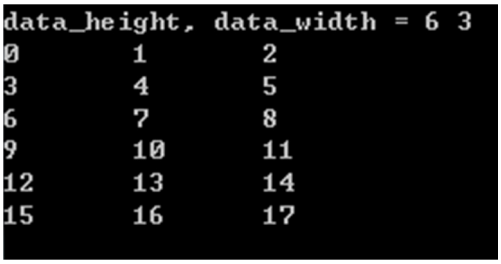
Due date: 10/26 03:00

2. Problem Description

(1) [Dynamic Array][Required File: PSA04_p1.cpp][50pts]

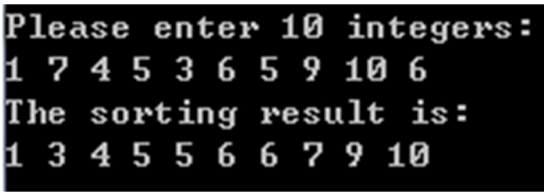
We can create a multi-dimensional dynamic array using pointer to pointers. Please implement the 2D dynamic array in PSA04_p1.cpp. Remember to delete the array to free the memory in the end. The value of each array elements should increase as the below example:





(2) [Sorting][Required File: PSA04_p2.cpp][50pts]

We can sort a large group of data to find the data we need easier. Please write a program that request 10 integers from user and print these integers from the smallest one to the largest one. The program should work like:



[Hint] There are many simple sorting algorithms, such as Bubble sort and Selection sort. We will take Bubble sort as an example in the next page. You can choose your sorting algorithm to sort this problem.

Bubble Sort

Bubble Sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass:

(**5 1** 4 2 8) ➡ (**1 5** 4 2 8)	Algorithm compares the first two elements, and swaps them since 5 > 1
(1 **5 4** 2 8) ➡ (1 **4 5** 2 8)	Swap since 5 > 4
(1 4 **5 2** 8) ➡ (1 4 **2 5** 8)	Swap since 5 > 2
(1 4 2 **5 8**) ➡ (1 4 2 **5 8**)	Since these elements are already in order (8 > 5), the algorithm doesn't swap them

Second Pass:

(**1 4** 2 5 8) ➡ (**1 4** 2 5 8)	No swaps
(1 **4 2** 5 8) ➡ (1 **2 4** 5 8)	Swap since 4 > 2
(1 2 **4 5** 8) ➡ (1 2 **4 5** 8)	No swaps
(1 2 4 **5 8**) ➡ (1 2 4 **5 8**)	No swaps

Now, the array is already sorted, but our algorithm doesn't know if it is completed or not. The algorithm needs one more pass without any swaps to know it is sorted.

Third Pass:

(**1 2** 4 5 8) ➡ (**1 2** 4 5 8)	No swaps
(1 **2 4** 5 8) ➡ (1 **2 4** 5 8)	No swaps
(1 2 **4 5** 8) ➡ (1 2 **4 5** 8)	No swaps
(1 2 4 **5 8**) ➡ (1 2 4 **5 8**)	No swaps. The sorting is finished.