CS1010 Programming Methodology
Week 9: Searching and Sorting

> Every act of conscious learning requires the willingness to suffer an injury to one's self-esteem. That is why young children, before they are aware of their own self-importance, learn so easily. ~*Thomas Szasz*

To students:

- Some programs for this discussion are on the CS1010 website, under the "Discussion" page.
- Some of the exercises here will be mounted on CodeCrunch.

I. For attempting on your own

The questions in this section are meant for you to attempt on your own before your discussion session. **They will not likely be discussed in class**, as these are the basics which we expect you to know. If you have doubts, please post them on the IVLE forum.

1. **Exploration: Timing your program.**
   Now that you have learned arrays which allow you to hold large amount of data, and algorithms with different running time complexities, you may be interested in timing certain parts of your program. The following program **timing.c** illustrates how to time the 'for' loop, using the **clock()** function in **<time.h>**. The value returned by **clock()** is the number of clock ticks elapsed since the program starts. To get the number of seconds used by the CPU, you need to divide it by **CLOCKS_PER_SEC** (defined in **<time.h>**).

```
#include <stdio.h>
#include <time.h>

int main(void) {
   clock_t start, finish;
   long i; // long = long integer

   start = clock();
   for (i=0; i<100000000; i++)
      ;   // empty loop body
   finish = clock();

   printf("Difference = %.2f sec.\n",
          (double)(finish - start)/CLOCKS_PER_SEC);
   return 0;
}
```

2. Choose the Selection Sort or Bubble Sort program and run it with arrays of different sizes, such as 1000, 2000, 4000, 8000, 16000. Verify that as you double the size of the array, the time it takes to sort the array is roughly quadrupled, providing empirical evidence that the algorithm is quadratic in running time complexity.

II. Discussion Questions

3. We illustrated sorting algorithms using integer arrays in class. Determining whether one element, say a[i], is smaller than another, say a[j], is simply done by comparing a[i] with a[j] (e.g.: `if (a[i] < a[j])`).

   What if the array elements are more complex (for example, a structure comprising more than one component, to be covered later), or the comparison criterion is more complex?

   Suppose you want to sort an integer array of 6 elements in increasing order of the first 3 digits of each element, how would you modify the selection sort program **selection_sort.c** that was given in class?

   A sample run is shown below:

```
Enter size: 6
Enter 6 values:
12345
9870
32
5555555
801784
729
After sort:
32 12345 5555555 729 801784 9870
```

4. **Enhanced Bubble Sort**
   As mentioned in class, the Bubble Sort can be enhanced. If you detect no swap in a pass, it implies that the array is already sorted by then. Write an enhanced Bubble Sort program **bubble_sort_enhanced.c**.

   The running time of your enhanced Bubble Sort is sensitive to the initial order of the data in the array. When does the best case occur? When does the worst case occur?

   (There are other variants of Bubble Sort, such as Bidirectional Bubble Sort, also known as Cocktail Sort or Shaker Sort. Check out the Internet for details.)

5. **Insertion Sort**
   Insertion Sort is another basic exchange sort besides Selection Sort and Bubble Sort. Refer to the PowerPoint file in the CS1010 module → "CA" → "Discussion" for the Insertion Sort algorithm.

   Implement Insertion Sort on an integer array.

6. **Checking duplicates.**
   The following is question 6 from week 6 discussion. Now that you have learned sorting, solve this task by sorting the array first.

   Write a program **duplicates.c** to fill an integer array with *n* ($1 \le n \le 1000$) random integers whose values are in the range [*lower*, *upper*] where *n*, *lower* and *upper* are inputs from user. Moreover, $0 < lower < upper$. Your program then computes the total number of duplicates in the array.

   For example, if a 15-element array contains the values { 97, 12, 45, 97, 23, 12, 53, 97, 30, 30, 10, 53, 8, 1, 53 }, then there are altogether 8 duplicates (three 97s, two 12s, and three 53s).

7. **Search for pattern**
   In the minesweeper game, the character '*' represents a mine and the character '-' represents a safe cell on a minefield. Assuming that you have an 8×8 minefield, and a 2×3 pattern, write a program **search_pattern.c** to count the number of times the pattern appears in the minefield. A sample run is shown below.

   Note: if you use scanf on characters, you will encounter some errors, because the 'enter' is itself a character, which will be read in. One way to overcome this is to use a space in the format specifier so that whitespace/enter characters are ignored, e.g.

```
scanf (" %c", &charvariable);
```

```
Enter minefield:
---****-
-*-**-**
********
-*-*--*-
********
**-*-**-
--------
********
Enter search pattern:
***
*-*
Answer = 4
```