Programming Design, Spring 2013
Suggested Solution to Homework 07

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

Problem 1

(a) It will print out the execution time of those statements contained in the commented block.

(b) Why do we need to cast `endTime - stTime`?

Both `endTime` and `stTime` are (long) integers. As `CLK_TCK` is also an integer, if none of these three are cast into a fractional number, the division operator will return an integer, which may not be accurate enough.

Problem 2

(a) The output will be 8 and 3, where 8 is the largest number in the array and 3 is the index of 8 (with the first number indexed as 0).

(b) The function use a loop to check the value of every element once, from the first one to the last one. Whenever a value is larger than its next value, it has the potential of becoming the maximum number. The function then compare its with the currently maximum number to see if a replacement should be performed. After the whole array is processed, the maximum value is found. More importantly, instead of returning the value, the function returns a pointer pointing to this value. This allows the calling function (the main function in this example) to output not only the maximum value but also its index in the array.

(c) The statement prints out the index of the maximum value in the array. `ptrGMax` stores the address of the maximum value while `value` stores the address of the first element (indexed as 0). As the difference is implemented to return the difference of indices, this statement prints out the corresponding index of the address stored in `ptrGMax`.

(d) The implementation is not efficient enough because it makes several unnecessary assignments `ptrLMax = &value[i]` and comparisons `if (*ptrLMax > *ptrGMax)`. In the current implementation, once a value is larger than its next value, it will be treated as a candidate of the maximum number. However, it is possible that it is smaller than the previous value. Therefore, we may modify the outer comparison `if (value[i] > value[i + 1])` to, e.g.,

```
if (value[i] > value[i + 1] && value[i] > value[i - 1])
```

for $i \geq 1$.

(e) A new function is implemented below. Note that it is not implemented in the most efficient way. Instead, it duplicates the idea of the original implementation.

```
void maxNoPtr (double value[], int arrayLen, double& gMax, int& maxIndex)
{
    double lMax = value[0];
    gMax = value[0];
    maxIndex = 0;

    for (int i = 0; i < arrayLen - 1; i++)
    {
        if (value[i] > value[i + 1])
```

```
        {
            lMax = value[i];

            if (lMax > gMax)
            {
                gMax = lMax;
                maxIndex = i;
            }
        }
    }

    if (value[arrayLen - 1] > gMax)
    {
        gMax = value[arrayLen - 1];
        maxIndex = arrayLen - 1;
    }
}
```

Problem 3

(50 points) Consider the program "PDSp13_hw07_rand.cpp". In this program, we implement two algorithms for generating `ARRAY_LEN` nonrepeating random numbers from 0 to `ARRAY_LEN - 1`.

(a) Omitted.

(b) When the number is small, the execution times of the two algorithms are similar. When the number becomes larger, however, the execution time of the `bruteForce()` is much larger than `shuffle()`.

(c) For `bruteForce()`, we try to generate random numbers one by one. When we want to generate the ith number, we first generate a random number and then compare it to all numbers we have generated. If there is an overlapping, we regenerate a random number. This is why when there are many existing numbers, generating a new nonrepeating random number is time consuming: It is too easy to generate a number that has been generated.

For `shuffle()`, we use a different way. We first generate a sorted list containing values from 1 to `ARRAY_LEN`, and then we "shuffle" the list. For iteration i, we swap value i with value j, where j is a randomly generated number. This in effect makes value i something randomly drawn from the whole pool of candidate values.

(d) As we explained in the previous part, `shuffle()` is more efficient.

(e) The new function is implemented below:

```
void bruteForceWhile (int array[])
{
    // in each iteration, set one value
    int i = 0;
    while (i < ARRAY_LEN)
    {
        // try to generate a randome number
        int theRand = rand() % ARRAY_LEN;

        // check whether this number has been generated
        bool hasRepeated = false;
        for (int j = 0; j < i; j++)
        {
            if (theRand == array[j])
```

```
            {
            hasRepeated = true;
                break;
            }
        }

        // if not, set it; otherwise, try again
        if (hasRepeated == false)
        {
            array[i] = theRand;
            i++;
        }
    }
}
```

(f) When we set `ARRAY_LEN` to 32500, the program terminates successfully. However, when we set it to 35000, it does not terminates. This is because the function `rand()` is implemented to return an integer between 0 and 32767, and 35000 is larger than 32767. Because in the function `shuffle()`, it is impossible to use `rand()` to generate a number that is unequal to all the first 32767 numbers, the program will not terminate.