CS246—Assignment 2 (Fall 2013)

R. Ahmed B. Lushman M. Prosser

Due Date 1: Friday, October 4, 5pm
Due Date 2: Friday, October 11, 5pm

Questions 1, 2a, 3, 4a, 5a are due on Due Date 1; the remainder of the assignment is due on Due Date 2.

Note: On this and subsequent assignments, you will be required to take responsibility for your own testing. As part of that requirement, this assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code due on Due Date 1. Instead, you will be asked to submit a few short answer questions and test suites for C++ programs that you will later submit by Due Date 2.

Test suites will be in a format compatible with A1Q4. So if you did a good job writing your `runSuite` script, it will serve you well on this assignment.

Be sure to do a good job on your test suites, as they will be your primary tool for verifying the correctness of your submission. For this reason, **C++ code due on Due Date 2 will only get one release token per day**. We want you to rely on your own pre-written test suite, not Marmoset, to verify correctness.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.

Note: Further to the previous note, your solutions may only `#include` the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, and `<string>`. No other standard headers are allowed. Marmoset will check for this.

Note: There will be a handmarking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. Please code to a standard that you would expect from someone else if you had to maintain their code.

1. **Note: there is no coding associated with this problem**. You are given a non-empty array $a[0..n-1]$, containing n integers. The program `maxSum` determines the indices i and j, $i \leq j$, for which $\sum_{k=i}^{j} a[k]$ is maximized and reports the maximum value of $\sum_{k=i}^{j} a[k]$. Note that since $i \leq j$, the sum always contains at least one array element. For example, if the input is

```
-3 4 5 -1 3 -9
```

then `maxSum` prints

```
11
```

(Output is printed on a single complete line with no padding.) Your task is not to write this program, but to design a test suite for this program. Your test suite must be such that a correct implementation of this program passes all of your tests, but a buggy implementation will fail at least one of your tests. Marmoset will use a correct implementation and several buggy implementations to evaluate your test suite in the manner just described.

Your test suite should take the form described in A1P4: each test should provide its input in the file `testname.in`, and its expected output in the file `testname.out`. The collection of all `testname`s should be contained in the file `suiteq1.txt`.

Zip up all of the files that make up your test suite into the file `a2q1.zip`, and submit to Marmoset.

2. In this problem, you will write a program called `change` that makes change for any country's monetary system (real or fictional). This program accepts, from standard input, the coin denominations that make up the monetary system, and the total value. It then prints a report of the combination of coins needed to make up the total. For example if a particular country has coins with values 1, 3, and 8, and you have 13 units of money, then the input would be as follows:

```
3
8
1
3
13
```

The initial 3 means that there are 3 coin types. The next three values are the coin denominations, in some order. The last value is the total. For this input, the output should be

```
1 x 8
1 x 3
2 x 1
```

Notes:

- Most coin systems have the property that you can make change by starting at the highest coin value, taking as many of those as possible, and then moving on to the next coin value, and so on. Although not all combinations of coin denominations have this property, you may assume that the input for `change` will always have this property.
- The Canadian government has recently abolished the penny. Consequently, once the remaining pennies work their way out of circulation, it will be impossible to construct coin totals not divisible by 5. Similarly, in whatever system of denominations you are given, it may not be possible to construct the given total. If that happens, output `Impossible` (and nothing else) to standard output.
- You may assume that the number of denominations is at most 10.
- You may assume that no denomination will be listed twice.
- If a given coin is used 0 times for the given total, do not print it out; your output should contain only those denominations that were actually used, in decreasing order of size.

(a) **Due on Due Date 1:** Design a test suite for this program. Call your suite file `suiteq2.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q2.zip`.

(b) **Due on Due Date 2:** Write the program in C++. Call your program `change.cc`.

3. Consider the following C++ program (`float.cc`, available in the SVN repository):

```
#include <iostream>
using namespace std;

int main () {
  float x = 0.0001;
  float y = 0;
  for (int i=0; i < 10000; i++) {
    y += x;
  }
  cout << y << endl;
  return 0;
}
```

Compile and run this program, and then answer the following questions:

(a) How does the actual behaviour of this program differ from its expected behaviour?

(b) How do you explain this difference?

Put your answers to both questions into the file `a2q3.txt`.

4. In an election, there are n candidates, numbered 1 through n, where n is non-negative. Each voter is allowed one vote for the candidate of his/her choice. The vote is recorded by number (from 1 to n). The number of voters is unknown beforehand, but is, of course, non-negative. Votes are terminated by end-of-file. Any vote which is not a number from 1 to n is considered an invalid (*spoilt*) vote. The input consists of the names of the candidates (one full name per line), followed by the votes. The first name is considered as candidate 1, and second as candidate 2, and so on. A candidate's name will never contain a numeral.

   Write a program to read the data and display the results of the election. For example, given the following data:

```
Victor Taylor
Denise Duncan
Kamal Ramdhan
Michael Ali
Anisa Sawh
Carol Khan
Gary Owen
3 1 2 5 4 3 5 3 5 3 2 8 1 6 7
7 3 5 6 9 3 4 7 1 2 4 5 5 1 4
```

   your program should produce the following output:

```
Number of voters: 30
Number of valid votes: 28
Number of spoilt votes: 2

Candidate      Score

Victor Taylor    4
Denise Duncan    3
Kamal Ramdhan    6
Michael Ali      4
```

```
Anisa Sawh       6
Carol Khan       2
Gary Owen        3
```

A starter file `a2q4.cc` has been provided in your `cs246/1139/a2` directory.

- Write the function, `readVotes`, which reads the data from standard input, processes the votes, and records the required information.
- Write the function, `printResults`, which prints the results of the election, using the information obtained by the call to `readVotes` (which you may assume has happened). The data must be presented **exactly** as in the example above. In particular, the Candidate column should be left-aligned and should have a width of 15 characters, and the Score column should be right-aligned with a width of 3 characters (except for the header, which is left-aligned).

Assume there are at most 10 candidates, and each name consists of at least 1 and at most 15 characters (including any spaces).

(a) **Due on Due Date 1**: Design a test suite for this program, using the main function provided in the test harness. Call your suite file `suiteq4.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q4.zip`.

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q4.cc`.

5. We typically use arrays to store collections of items (say, integers). We can allow for limited growth of a collection by allocating more space than typically needed, and then keeping track of how much space was actually used. We can allow for unlimited growth of the array by allocating the array on the heap and resizing as necessary. The following structure encapsulates a partially-filled array:

```
struct IntArray {
  int size;  // number of elements the array currently holds
  int capacity;  // number of elements the array could hold, given current
                 // memory allocation to contents
  int *contents;
};
```

- Write the function `readIntArray` which returns an `IntArray` structure, and whose signature is as follows:

  ```
  IntArray readIntArray();
  ```

  The function `readIntArray` consumes as many integers from `cin` as are available, populates an `IntArray` structure in order with these, and then returns the structure. If a token that cannot be parsed as an integer is encountered before the structure is full, then `readIntArray` fills as much of the array as needed, leaving the rest unfilled. If a non-integer is encountered, the first offending character should be removed from the input stream (i.e., call `cin.ignore` once with no arguments). In all circumstances, the field `size` should accurately represent the number of elements actually stored in the array and `capacity` should represent the amount of storage currently allocated to the array.

- Write the function `addToIntArray`, which takes a pointer to an `intArray` structure and adds as many integers to the structure as are available on `cin`. The behaviour is identical to `readIntArray`, except that integers are being added to the end of an existing `intArray`. The signature is as follows:

```
void addToIntArray(IntArray&);
```

- Write the function `printIntArray`, which takes a pointer to an `IntArray` structure, and whose signature is as follows:

  ```
  void printIntArray(const IntArray&);
  ```

  The function `printIntArray(a)` prints the contents of `a` (as many elements as are actually present) to `cout`, on the same line, separated by spaces, and followed by a newline. There should be a space after each element in the array (including the last element), and not before the first element.

  It is not valid to print or add to an array that has not previously been read, because its fields may not be properly set. You should not test this.

For memory allocation, you **must** follow this allocation scheme: every `IntArray` structure, when first used, is initially allocated with a capacity of 5. If at any point, this capacity proves to be not enough, you must double its capacity (so capacities will go from 5 to 10 to 20 to 40 ...). Note that there is no `realloc` in C++, so doubling the size of an array necessitates allocating a new array and copying items over. Your program must not leak memory.

A test harness is available in the starter file `a2q5.cc`, which you will find in your `cs246/1135/a2` directory. **Make sure you read and understand this test harness, as you will need to know what it does in order to structure your test suite.** Note that we may use a different test harness to evaluate the code you submit on Due Date 2 (if your functions work properly, it should not matter what test harness we use).

(a) **Due on Due Date 1:** Design a test suite for this program, using the main function provided in the test harness. Call your suite file `suiteq5.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q5.zip`.

(b) **Due on Due Date 2:** Write the program in C++. Call your solution `a2q5.cc`.