CS246—Assignment 2 (Spring 2018)

C. Kierstead A. Moss V. Sakhnini

Due Date 1: Monday, May 28, 11:55pm
Due Date 2: Monday, June 4, 11:55pm

Questions 1, 2a, 3a, 4a, Bonus(a) 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 test suites for C++ programs that you will later submit by Due Date 2. While part 1 focuses on testing, this doesn't mean that you can't start develop your code for Due Date 2. You should always start working on both parts as early as possible.
Test suites will be in a format compatible with A1Q5/6. 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.

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 hand marking 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. Further comments on coding guidelines can be found here: https://www.student.cs.uwaterloo.ca/~cs246/current/ AssignmentGuidelines.shtml

Note: You are not permitted to ask any public questions on Piazza about what the programs that make up the assignment are supposed to do. A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Instead, we will provide compiled executables, suitable for running on linux.student.cs, that you can use to check intended behaviour. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

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 ≤ 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 ≤ 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 A1Q5: 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 the repository (under 1185/assignments/a2 folder) you will find a program called `args.cc,` which demonstrates how to access command line arguments from a C++ program. Use that program as an example to help you solve this problem.

   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, as command-line parameters, 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, from highest to lowest denomination.

   For example, if a particular country has coins with values `1, 10,` and `25,` and you have `68` units of money, then the command-line would read as follows:

```
./change 1 10 25 68
```

The initial three values are the coin denominations, in any order. The last value is the total. For this input, the output should be:

```
2 x 25
1 x 10
8 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.
- The program needs at least 2 command-line parameters: a minimum of one denomination and one total. If the user doesn't provide at least the minimum number of command-line arguments, output the following line to standard output and exit.

  ```
  Usage: change [denominations] [amount]
  ```
- Valid command-line parameters are positive integers. When testing, you may assume that only valid input is passed on the command-line (i.e. no alphabetic or otherwise invalid characters are passed as arguments).
- Denominations may be listed in any order.
- You may assume that the number of denominations is at most 10. Do not allocate heap memory.
- 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`, `.out` and `.args` files, into the file `a2q2.zip` (note: you need to provide *.{args,in}, even if it's empty).

(b) **Due on Due Date 2:** Write the program in C++. Save your solution in a2q2.cc

3. In this problem, you will write a program called `wordWrap,` whose purpose is to confine text to a given width. `wordWrap` can take a single argument on the command line, a positive integer denoting the width of the line. If no argument is supplied on the command line, the width is 25. `wordWrap` takes a sequence of words on `stdin` and echoes them to `stdout`, such that the width of the output is no wider than the provided command-line argument. For example, if the width is 25 and the text is as seen below:

```
 Friends Romans countrymen lend me your ears I come to bury
Caesar not to praise him
```

then the output would be

```
Friends Romans countrymen
lend me your ears I come
to bury Caesar not to
praise him
```

If a word is too long to fit on what remains of the line, put it on the next line. Do not break a word unless it is longer than the entire allowed width. For example, the same text with a width of 8 becomes

```
Friends
Romans
countrym
en lend
me your
ears I
come to
bury
Caesar
not to
praise
him
```

When outputting words, separate them by a single whitespace character, either a single space or a single newline, regardless of how they are spaced in the input. For example, if the input contained words separated by two spaces, they would still be separated by one whitespace character in the output (In another words, all original whitespace between any two words is replaced by the one necessary whitespace character, either a space or a newline) .

(a) **Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteq3.txt`. Zip your suite file, together with the associated `.in, .out,` and `.args` files, into the file `a2q3.zip.`

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q3.cc.`

4. 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 reference 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 reference 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 begins with a capacity of `0.` The first time data is stored in an `IntArray` structure, it is given a capacity of `5` and space allocated accordingly. If at any point, this capacity proves to be not enough, you must double the 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 `a2q4.cc`, which you will find in your `cs246/1185/assignments/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 `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++. Call your solution `a2q4.cc.`

Bonus (4 points)

(This question is not mandatory; for each part you will either get full marks or nothing; no partial marks are awarded for the bonus parts.)

Enhance your solution to problem 3 by accepting a second optional command-line option, which will be the name of a file. The command-line syntax for your program will now be

```
./a2qbn width filename
```

Note that it is indeed possible to just specify a filename and not the width, in which case the default width is used.

The file whose name is specified on the command line will contain hyphenation information, formatted as in the following example:

```
hotdog 3
backgammon 4
starting 5
```

The number indicates the number of characters after which the associated word may be hyphenated. For example, “hotdog” may be hyphenated as “hot-dog”. You may assume that the provided numbers will always be at least 2 and at most (length of the word – 2).

The behaviour of your program should be the same as in problem `3,` except that if a word does not fit on the given line, you should check to see whether the word would fit if it were hyphenated (remember that the hyphen itself occupies a character). If it would, print the first part of the word (and a hyphen) on the current line, and continue the word on the following line. Otherwise, print the entire word on the following line, as before. If a word does not occur in the hyphenation file, it cannot be hyphenated.

For example, the example given in problem 3, with a width of `24,` would (with the appropriate hyphenation information) be printed as

```
Friends Romans country-
men lend me your ears I
come to bury Caesar not
to praise him
```

If you need to look up hyphenation information, you **must** search the file. Do **not** store the contents of the hyphenation file in memory. You must open the file and search it each time you need hyphenation information.

(a) **[2 points] Due on Due Date 1**: Design a test suite for this program. Call your suite file `suiteqbn.txt`. Zip your suite file, together with the associated `.in`, `.out`, and `.args` files, into the file a2qbn.zip.

(b) **[2 points] Due on Due Date 2**: Write the program in C++. Save your solution in `a2qbn.cc.`