CS246—Assignment 2 (Fall 2022)

R. Evans E. Lee B. Lushman

Due Date 1: Friday, October 7, 5pm
Due Date 2: Friday, October 21, 5pm

Questions 1, 2a, 3a, and 4a are due on Due Date 1;

The remainder of the assignment is due on Due Date 2.

Note: Due Date 1 consists of test suites that are compatible with A1's `runSuite`. Test suite submission zip files are restricted to contain a maximum of 40 tests and the size of each test file is restricted to 500 kilobytes.

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. We will not answer Piazza questions about coding style; use your best judgment. Further comments on coding guidelines can be found here: `https://student.cs.uwaterloo.ca/~cs246/F22/codingguidelines.shtml`

Before asking on Piazza, see if your question can be answered by the sample executables we provide. 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. 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, $i \leq j$, 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 provided via stdin 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 A1: each test should provide its input in the file `testname.in`. The collection of all `testname`s should be contained in the file `suiteq1.txt`. The program `maxSum` does not use command line arguments, so no `.args` files are necessary.

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

2. Implement a simplified version of the Unix command `wc`. If no arguments are provided, your implementation should read input from stdin. If arguments are provided, your implementation should display the line, word, and character counts from each file given as an argument. Your implementation should display a total sum of counts if more than one file is given.

   Your implementation should also support passing the argument `-` (a hyphen). If `wc` receives a hyphen, it does not try to look for a file called `-`, it instead reads from standard input. This can be tested with `wc`, or the sample executable.

   You do not have to support any flags, and you may assume all input is valid.

   Your output may differ from that of the "real" `wc` with respect to whitespace usage. Following the sample executable is easiest.

   (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`, `.args` and any other text files, into the file `a2q2.zip`.

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

3. Implement the game Tic Tac Toe.

   If no arguments are provided, your implementation should read input from stdin. If arguments are provided, your implementation should treat each file passed as an independent game of Tic Tac Toe. In either case, you must print a tally of `x` wins, `o` wins, ties, and aborted games. Your impementation **does not** have to support the argument `-` (a hyphen).

Input Format

Tic Tac Toe is played by taking turns on a 3 by 3 grid. Each player's objective is to mark a line of 3 squares on the grid as their own. This line can be horizontal, vertical, or diagonal.

Each line of input consists of a player (`x` or `o`), as well as the row and column they wish to mark as their own. `x` always gets to go first.

An example game might look like:

```
x 0 0
o 0 1
x 1 1
o 0 2
x 1 2
o 2 2
x 1 0
```

Abritrary amounts of whitespace is allowed between each word on a line.

Output Format

After each line of input, a board should be printed to stdout. The board at the end of the previous game would look like:

```
|x|o|o|
|x|x|x|
| | |o|
```

When a player wins, your implementation must print either `x won!` or `o won!` to stdout. If a tie occurs (all squares are filled without any lines of three), print `Tie` to stdout.

Before the program terminates (i.e, after all the games have been completed), you must print a summary of the games like so:

```
x Wins:         4
o Wins:         5
Ties:           2
Aborted Games:  1
```

The text appears left aligned in a column of width 16. The scores appear left aligned immediately after the column.

Required Error Checking:

You must report the following errors to stderr.

- `Bad file: filename` if the argument `filename` cannot be read.

- `Invalid move: move` if a move is invalid. The second occurrence of the word `move` should be replaced by the line of the file currently being read. E.g, `x 100 200` would print `Invalid move: x 100 200` to stderr. Possible invalid moves include:
  - There is more or less input on a line than necessary
  - The coordinates are out of range, of an incorrect type, or for a square that has already been used
  - It was not the player's turn to move, or an invalid character (not x/o) was given for the player.
- `Unfinished game` if you reach EOF without a clear winner.

If any error occurs, that game is considered aborted, and your program should move on to the next game. If any game is aborted, your program should return an exit code of 1 on completion.

Although we are prescribing behaviour for these listed error cases, there are other possible invalid inputs not listed here. Such cases are considered *undefined behaviour*. If we do not explicitly tell you how to handle a particular type of invalid input, then you do not need to worry about that case. Such cases (invalid inputs that fall under the umbrella of undefined behaviour) should also not be submitted as part of a test suite. This applies to all assignments and questions in the course.

(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`, `.args` and any other text files, into the file `a2q3.zip`.

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

4. In this question, you will implement a structure that represents a mathematical set for integers. The structure definition and its associated function signatures have already been given to you in the file `a2q4.cc` in your repository, and are reproduced below:

```
struct IntSet {
  int *data;
  int size;
  int capacity;
};

void init( IntSet & i );    // Initialize to be "empty" with following field values: {nullptr,0,0}
void destroy( IntSet & i ); // Clean up the IntSet and free dynamically allocated memory.

IntSet operator| ( const IntSet & i1, const IntSet & i2 ); // Set union.
IntSet operator& ( const IntSet & i1, const IntSet & i2 ); // Set intersection.
bool   operator==( const IntSet & i1, const IntSet & i2 ); // Set equality.

bool isSubset( const IntSet & i1, const IntSet & i2 );   // True if i2 is a subset of i1.
bool contains( const IntSet & i, int e );                // True if set i contains element e.
void add( IntSet & i, int e );                           // Add element e to the set i.
void remove( IntSet & i, int e );                        // Remove element e from the set i.

// Output operator for IntSet.
std::ostream& operator<<( std::ostream & out, const IntSet & is );

// Input operator for IntSet. Continuously read int values from in and add to the passed IntSet.
// Function stops when input contains a non-int value. Discards the first non-int character.
std::istream& operator>>( std::istream & in, IntSet & is );
```

You have also been given a test harness in `a2q4.cc` as the `main` function. **Do not change it.** Make sure you read and understand this code, as you will need to know what it does in order to structure your test suite. It isn't very robust and does no error-checking. You are allowed to assume that the commands it receives are valid, so *do not write test cases for it*!

`q`	terminates the program. Also invoked by Ctrl-d.
`n` *idx*	initializes the set at index *idx*. Uses the input operator to reads in integers and store them in the set until a non-integer is detected, which is then thrown away.
`p` *idx*	prints the set at index *idx* using the output operator.
`&` *idx1 idx2*	prints the intersection of sets *idx1* and *idx2*.
`|` *idx1 idx2*	prints the union of sets *idx1* and *idx2*.
`=` *idx1 idx2*	prints true if set *idx1* is equal to set *idx2*; otherwise, prints false.
`s` *idx1 idx2*	prints true if set *idx2* is a subset of *idx1*; otherwise, prints false.
`c` *idx elem*	prints true if set *idx* contains element *elem*; otherwise, prints false.
`a` *idx elem*	adds element *elem* to set *idx* .
`r` *idx elem*	removes element *elem* from set *idx* .

We have provided a sample solution to this problem in your `a2/q4` directory in the form of an executable binary named `IntSet`. Note that it is compiled to run on the `linux.student.cs.uwaterloo.ca` environment. We have also provided a sample test input file `q4/sample.in` Since the program doesn't take command-line arguments, there is no `sample.args` file.

Implement the described `IntSet` functions. Your implementation must satisfy the following requirements:

- A set cannot contain duplicate items. Attempting to add an item already in the set does not change the set nor does it produce an error (see `add` and `operator&`).

- `init` initializes both the capacity and size of the set to 0. Its array pointer is set to `nullptr`.
- `destroy` frees all dynamically allocated memory. We will be using valgrind in marmoset to check for memory leaks.
- After `add( IntSet & i, int e )` is called, `e` must be in the set `i`. When the first integer is added, the capacity is increased to 5. If at any point the capacity is not enough, it should be doubled. The size of the set goes up by one for every value added .
- Set intersection (`operator&`) returns a new set containing only the elements that are in both sets.
- Set union (`operator|`) returns a new set containing all of the elements from both sets, without any duplicates.
- Two sets A and B are equal (`operator==`) if and only if no element of one is not an element of the other. More precisely: $(\nexists x s.t. x \in A \wedge x \notin B) \wedge (\nexists x s.t. x \in B \wedge x \notin A)$.
- `isSubset( const IntSet & i1, const IntSet & i2 )` returns true if and only if every element in `i2` is an element of the set `i1`: otherwise it returns false.
- `contains( const IntSet & i, int e )` returns true if `e` is an element of the set `i`: otherwise it returns false.
- After `remove( IntSet & i, int e )` is called, `e` must not be in the set `i`. You are not required to reduce the array size or capacity in this implementation.
- The input operator (`operator>>`) initializes the set parameter, and then adds the read-in integers to the set until a non-integer value is read. The first non-integer character is thrown away.
- The output operator (`operator<<`) first prints a left parenthesis '(', then each integer in *ascending order*, delimited by a comma unless it is the last integer, and ends with a right parenthesis ')'. For example the set containing 3, 5, and 2 is printed as `(2, 3, 5)`. An empty set is printed as `()`.

(a) **Due on Due Date 1:** Design a test suite for this program. Submit a file called `a2q4.zip` that contains the test suite you designed, called `suiteq4.txt`, and all of the `.in` files.

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

Submission:
The following files are due at Due Date 1: `a2q1.zip`, `a2q2.zip`, `a2q3.zip`, `a2q4.zip`,

The following files are due at Due Date 2: `a2q2.cc`, `a2q3.cc`, `a2q4.cc`.