EECS 281 - Project 3: 281Bank

Due Tuesday, November 12, 2024 at 11:59 PM

Overview

281Bank is looking for a developer to design their online banking infrastructure, and they require efficient code! The program will read in a set of customers, a set of instructions consisting of transactions between customers and various queries to analyze said transactions, and hopefully by the end of the day, 281Bank will be a profitable bank so that 281 might have enough funds to get tens of thousands of new autograder machines!

As financial institutions, banks have to keep detailed records of their transactions in the event of a dispute, to process legitimate transactions, and to flag and reverse fraudulent ones. As each transaction builds on the past ones, (each new transaction carries with it a certain change to the current state of all balances on the system) this presents a challenge, as we want our system to be just as responsive a year from now. Banking traditionally used to batch transactions and apply credits before applying the debits, and then any payments which bounced would be reversed. A relatively newer approach is to use something called a real-time gross settlement system, and that is how 281Bank will operate! We will be applying credits and debits in the order they are set to be executed, while still flagging fraudulent ones.

For this project, you will write a bank wire transfer simulator. Your executable will be called `bank`. You will gain experience writing code that makes use of multiple data structures, your job will be to find which data structures are best suited for which problem at hand.

Learning Goals

- Selecting appropriate data structures for a given problem. Now that you know how to use various abstract data types, it's important that you are able to evaluate which will be the optimal for a set of tasks.
- Evaluating the runtime and storage tradeoffs for storing and accessing data contained in multiple data structures
- Designing a range of algorithms to handle different situations
- Evaluating different representations of the same data

Command Line Options

Your program, `bank`, should accept the following command line options:

- `--help/-h`

This causes the program to print a helpful message and immediately `exit(0)`

- `--file/-f filename`

  This is followed by a `filename` for the registration file

- `--verbose/-v`

  This causes the program to print certain log messages, as defined in the spec

You do not need to error check command line inputs, but you might find it helpful during testing if you print an error message if the filename is not provided, or the file cannot be opened.

The option `--verbose/-v`, means that calling the program with `--verbose` does the same thing as calling the program with `-v`. See getopt for how to do this.

Legal Command Line Examples

`$ ./bank --file registration.txt < commands.txt > out.txt`
Run the program with registrations read from the `registration.txt` file, commands redirected to `cin` from `commands.txt`, and `cout` output redirected to `out.txt`.
`$ ./bank -vf spec-reg.txt < spec-commands.txt > output.txt`
Run the program with registrations read from the `spec-reg.txt` file, transactions redirected to `cin` from `spec-commands.txt`, and `cout` output redirected to `output.txt`; verbose output must be generated.
`$ ./bank --help`
Prints a helpful message and exits.

Illegal Command Line Examples

`$ ./bank --verbose < spec-commands.txt`
`registrations` filename not specified.

Input

Unless `--help/-h` is asked for, your program should receive the `--file/-f` option on the command line, followed by the name of the account registration file (described in more detail below). You could either type the commands by hand, copy/paste them from a file, or redirect input for the commands file. You can assume that there will always be at least one operation command, and the `$$$` line to indicate the break between operations and queries, though there are not required to be any queries. There is more information on each type of command later in this spec.

Account File

Your program will read in a list of user registrations to facilitate subsequent transactions. Each line of input will consist of four parts:

`REG_TIMESTAMP|USER_ID|PIN|STARTING_BALANCE`

Timestamp Format

Timestamps will be given in the format yy:mm:dd:hh:mm:ss, where the various components (year, month, day, hour, minute, second) between colons are given as a pair of digits. In the universe of 281bank, each one of these components can range from 00-99. Note that there will always be two digits because numbers such as 7 are represented as "07". You do not need to check that the digits in each field 'make sense' according to our time system. It is OK to have a timestamp with more than 24 hours or more than 12 months, for example. Further, 00:00:00:00:01:00 is later than 00:00:00:00:00:61 in the context of this project. Similarly, 00:00:00:01:30:00 is later than 00:00:00:00:91:00, etc. This is to make input parsing and error checking easier, though it may seem counterintuitive.

The field `REG_TIMESTAMP` represents the time at which the customer has registered as a member to the bank.

User ID Information

The `USER_ID` is guaranteed to be unique in the registration file. It is a combination of letters, numbers, and possibly other characters like `_` or `-`. It should never have a space in it.

Other Registration Information

The other two fields are a 6-digit Personal Identification Number or PIN (no need to securely store passwords in 281Bank, we do not deal with security here and leave it to the professionals in 388Bank), and finally their starting balance. A user's balance will always fit into a 32 bit unsigned integer, and due to potential rounding errors, all transactions will be completed with whole integer values.

User Commands

On startup, your program should prepare to accept commands from the user. These commands may or may not take the form of redirected input. There will be two parts to the input: operations and the query list. For both parts of the input, the first letter of each command is unique, and if you see that character at the start of a command, you can assume that you know the command. For example, if a command starts with `l`, you can assume that it is a login command.

Operations

There are five defined operations that can be performed: **#** (comment), **login**, (log)**out**, **balance** and **place** (transaction):

Operation: # (comment)

`Usage: #Any text that you want, with or without a space after the number sign`

Used to add comments to command files, produces no actions or output. Discard any lines beginning with #. This command does not produce any output nor should it generate any errors.

Operation: Login

`Usage: login <USER_ID> <PIN> <IP>`

The user attempts to login to their account using their `USER_ID` and 6 digit `PIN`. If both match, this user is allowed to start placing transaction requests, and their IP address is saved in a user-specific valid IP list for future processing. If the verbose flag is set, print `User <USER_ID> logged in.` to standard output. If for any reason the user is not able to log in and if the verbose flag is set, print `Login failed for <USER_ID>.`

Operation: Logout

`Usage: out <USER_ID> <IP>`

If the user has an active session and the IP is an IP the user logged in with before, this logs them out, and removes the IP from the valid IP list for that user. Logged out users cannot place transaction requests from the same IP without logging back in. If the verbose flag is set, print `User <USER_ID> logged out.` to standard output. If for any reason the user is not able to log out (e.g., the IP does not match any of the IPs that the user has previously logged in with) and the verbose flag is set, print `Logout failed for <USER_ID>.`

Operation: Balance

`Usage: balance <USER_ID> <IP>`

If the user has an active session and the IP is an IP the user logged in with before, this displays their balance as of the most recently known timestamp.

If the verbose flag is set:

- If the user ID is invalid, print `Account <USER_ID> does not exist.`
- If the account does not have an active user session, print `Account <USER_ID> is not logged in.`
- A transaction must also be checked for fraudulent activity. 281Bank defines a fraudulent transaction as a transaction where the IP address in the command is not one of the valid list of IP addresses for the sender. Check back on the `login` and `out` commands on how the valid IP list is maintained. If the transaction ***is*** fraudulent, you should output the following:

  `Fraudulent transaction detected, aborting request.`

If everything about the command is valid, regardless of the verbose setting, you should display:

`As of <TIMESTAMP>, <USER_ID> has a balance of $<BALANCE>.`

The `<TIMESTAMP>` that you would use is the most recent one specified in a `place` command (see below for details). It does not matter if the `place` command was valid, or whether the transaction has been processed yet, just use the timestamp specified in the command. If no `place` command has occurred yet, use the timestamp when the account was created, from the registration file.

Operation: Place (Transaction)

`Usage: place <TIMESTAMP> <IP> <SENDER> <RECIPIENT> <AMOUNT> <EXEC_DATE> <o/s>`

This command will have 7 arguments: the timestamp at which the order is placed, the IP of the sender, the sender ID, the recipient ID, the transaction amount, proposed execution date, and whether the transaction fee is covered by our account (o, meaning the sender) or shared equally between the sender and the recipient (s). This is used to place a transaction to be executed in the future. You should make sure that all place commands contain non-decreasing timestamps (see the Error-checking section). As per international guidelines, 281Bank is only allowed to place transactions that are processed up to three days ahead of the given timestamp, meaning `EXEC_DATE - TIMESTAMP <= 3 days`.

When producing the output of the `place` command, the output should always occur in non-decreasing order of timestamps, in terms of when they are placed and actually executed.

Transactions

A transaction at heart is a transfer of funds betweeen two parties. The bank must have many checks in place to ensure transaction integrity for that transfer, and 281Bank will create some of its own checks before a transaction can successfully go through. At transaction ***placement*** time, here are some basics that a transaction must have:

- The sender is different from the recipient
- An execution date that's three or less days from the timestamp of the transaction
- The sender exists (in the registration data)
- The recipient exists
- An execution date that is not earlier than the sender's and recipient's registration dates (both users must have accounts already created at the execution time of the transaction)
- An active user session with the IP address given in the command

The `place` command should be checked for validity in the order listed above. The first error encountered stops the processing of the command, with output as stated below.

If the verbose flag is set:

- If an incoming transaction has the same sender and recipient, print `Self transactions are not allowed.`
- If the `EXEC_DATE` is set more than three days in the future, print `Select a time up to three days in the future.`
- If sender ID is invalid, print `Sender <SENDER> does not exist.`
- If recipient ID is invalid, print `Recipient <RECIPIENT> does not exist.`
- If the execution date is earlier than either party's registration dates, print `At the time of execution, sender and/or recipient have not registered.` (same message no matter who is not registered at

that time)

- If sender does not have an active user session, print `Sender <SENDER> is not logged in.`
- If the transaction is fraudulent (see the `balance` command above for details), you should output:

  `Fraudulent transaction detected, aborting request.`

Failing this or any of the above checks should invalidate the transaction. If the transaction is valid and verbose mode is on, the following message should be printed:

`Transaction <ID> placed at <TIMESTAMP>: $<AMOUNT> from <SENDER> to <RECIPIENT> at <EXEC_DATE>.`

If all the placement time checks are complete, the transaction may have its transaction ID generated and be queued for execution. It is paramount that transaction order integrity is maintained, meaning if a transaction is scheduled two days from now but a subsequent transaction is placed only 30 minutes from now, the transaction that's placed after the initial transaction will be executed first. This ensures that all users have the proper balances before a transaction is executed (i.e. Alice may not have enough money to send to Bob, but if the transactions execute in the order they should, Mallory will have sent Alice enough money to pay Bob in time). **For transactions that have identical execution dates, process the transaction with the earlier transaction ID.** You should use a single `priority_queue<>` to schedule all transactions.

During execution, 281Bank should check whether the sender and recipient have enough money to cover the transaction + fees. In the case of shared fees, the recipient must have enough money in their account to pay the fee **before** they can receive the transfer. If the transaction cannot happen and verbose mode is on, print the following message:

`Insufficient funds to process transaction <transaction_id>.`

When this happens, you should discard the transaction: it will not appear in either user's completed transaction, and no balances would change. Insufficient funds should be checked when the transaction **completes**, not when the order is placed. For example, I should be able to schedule a transfer to pay a bill 3 days from now, knowing that my employer will deposit my check tomorrow.

When should a transaction be removed from the priority queue? The top of the PQ should be the transaction with the earliest execution date. If you've read a new `place` order with a **timestamp** that is greater than or equal to the top execution date, you should process commands until this is no longer the case. Whether anything needed to be removed or not, the newest transaction can now be added to the PQ (and printed, if verbose mode is on). When you reach the end of operations (`$$$` encountered), any remaining transactions should be processed.

Transaction Fees

Any transaction that is deemed valid at the time will get a new transaction id (strictly increasing counter) and will be added to a master transaction list to be further examined. The bank will collect the transaction fee at the execution date, and will be calculated using the following formula:

- Minimum fee: 10 USD

- 1% fee of the transaction amount (integer division)
- Maximum fee: 450 USD

To reward loyalty, any customers who wish to **send** money and have been with 281Bank for more than 5 years from the given execution date of the transaction will receive a 25% discount on the fees calculated (see below). This applies to both `o` and `s` type transactions; the 25% deduction will be a deduction on the overall transaction fee. It's important that only the sender is considered when calculating the discount, meaning a longstanding (>5 years) customer who places a transaction with a new member will still have the discount applied on that transaction, even if the fee is shared. The discount is also calculated after the maximum/minimum is imposed, so a longstanding customer would effectively have a maximum/minimum fee of $337/$7 respectively. When calculating shared fees, if the total fee is an odd number, the sender pays the larger share of the fee. For example, if there was a $7 fee, the sender would pay $4 and the recipient would pay $3. When a transaction is successfully executed and verbose mode is on, the following message should be printed:

```
Transaction <ID> executed at <EXEC_DATE>: $<AMOUNT> from <SENDER> to <RECIPIENT>.
```

When calculating the fee when a discount **does** occur, use this formula to make sure you get the correct value: `fee = (fee * 3) / 4;`

Query List

After executing every transaction, we may want to go back in time and examine certain parts of the transaction chain, or collect some aggregate data. The queries will be kept within the same standard input as the transaction requests. A single line separator of `$$$` will signify the switch over to query mode. Before you switch over to query mode, make sure to execute all transactions that were placed earlier. Once in query mode, there will be four commands to query all the completed transactions. Here, each query's output will always printed regardless of the verbose flag's state. For commands that require transactions to be printed, print them in chronological order.

List Transactions

```
Usage: l x y
```

`l x y` : lists transactions that were executed between execution times [x,y). The timestamps must include colons, and y must be >= x; for instance:

```
l 11:22:33:44:55:66 22:33:45:67:89:00
```

Refer to Timestamp Format for more information.

Output format

```
<Transaction ID>: <sender> sent <amount> dollars to <recipient> at <execution_date>.
...
...
...
```

```
<Transaction ID>: <sender> sent <amount> dollars to <recipient> at <execution_date>.
There were <total number> transactions that were placed between time x to y.
```

If there was only one transaction, you should instead output `There was 1 transaction that was placed between time x to y.`

If someone sent 1 dollar to someone else, you should output `1 dollar`, not `1 dollars`.

However, if the two execution times are the same, you should instead output `List Transactions requires a non-empty time interval.`

Bank Revenue

`Usage: r x y`

`r x y`: calculates the revenue the bank will make from timestamp [x,y). This revenue is calculated by the total amount of fees the bank will collect on all transactions that were executed between the timestamps x and y. Remember that the fee a bank makes from a transaction is determined when the transaction is placed, not when it's executed, but the bank will receive that revenue once the transaction is executed. The timestamps should meet the same criteria as the `l` command (above).

Output: Revenue

```
281Bank has collected <amount> dollars in fees over <time interval formatted(y-x)>.
281Bank has collected 420 dollars in fees over 2 months 3 days 1 hour 22 seconds.
```

As an example, if `x` is time `00:00:00:00:00:00` and `y` is time `00:00:11:22:33:44`, the time formatted version of this interval would be: `11 days 22 hours 33 minutes 44 seconds`. Because of the minimum transfer fee, you can assume that it will always be `dollars`.

However, if the two execution times are the same, you should instead output `Bank Revenue requires a non-empty time interval.`

Customer History

`Usage: h user_id`

`h user_id`: summarizes a customer's history, outgoing and incoming executed transactions. If there are more than 10 incoming or outgoing transactions, show the latest 10 for each category. Always display them in order from least recent to most recent (in terms of execution). If the user_id is not in our database, output `User USER_ID does not exist.`

Output: History

```
Customer paoletti account summary:
Balance: $18300
Total # of transactions: 3
Incoming 2:
```

```
2: hjgarcia sent 27 dollars to paoletti at 30001.
3: mmdarden sent 420 dollars to paoletti at 30004.
Outgoing 1:
17: paoletti sent 1100 dollars to darden at 51002.
```

If someone sent 1 dollar to someone else, you should output `1 dollar`, not `1 dollars`.

Summarize Day

`Usage: s timestamp`

`s timestamp`: summarize all transactions that were executed on the day of the timestamp (given in integer format), from timestamp to timestamp + 1 day, inclusive/exclusive. For example, if the command is: `s 00:01:12:37:89:62`, you should summarize the transactions that occur between `[00:01:12:00:00:00, 00:01:13:00:00:00)` (do not include colons in the output).

Output: Summarize

```
Summary of [112000000, 113000000):
2: hjgarcia sent 27 dollars to paoletti at 112030001.
3: mmdarden sent 420 dollars to paoletti at 112030004.
4: mertg sent 75 dollars to danlliu at 112030454.
17: paoletti sent 1100 dollars to darden at 112031002.
There were a total of 4 transactions, 281Bank has collected 41 dollars in fees.
```

For the final summary line: if there was only one transaction, you should print `There was a total of 1 transaction.` But as for the fee output, because of the minimum transfer fee, you can assume that it will always be `dollars`.

Libraries and Restrictions

The use of the C/C++ standard libraries is highly encouraged for this project, especially functions in the `<algorithm>` header and container data structures. The smart pointer facilities, and thread/atomics libraries are prohibited. As always, the use of libraries not included in the C/C++ standard libraries is forbidden.

Unless otherwise stated, you are allowed and **encouraged** to use all parts of the C++ STL and the other standard header files for this project, especially functions in the `<algorithm>` header and container data structures. You are not allowed to use other libraries (eg: boost, pthread, etc). You are not allowed to use the `shared_pointer` or `unique_pointer` constructs from the memory include file (we want you to learn proper use of pointers yourself). You are not allowed to use the C++11 regular expressions library (it is not fully implemented in gcc 6.x) or the thread/atomics libraries (it spoils runtime measurements).

Output

The program will write its output to standard output (`cout`).

Testing Your Solution

It is extremely frustrating to turn in code that you are "certain" is functional and then receive half credit. We will be grading for correctness primarily by running your program on a number of test cases. If you have a single silly bug that causes most of the test cases to fail, you will get a very low score on that part of the project *even though you completed 95% of the work*. Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To help you do this we will require that you write and submit a suite of test files that thoroughly test your project.

Your test files will be used to test a set of buggy solutions to the project. Part of your grade will be based on how many of the bugs are exposed by your test files. (We say a bug is exposed by a test file if the test file causes the buggy solution to produce different output from a correct solution.)

Each test file group should be a pair of input files that describes the bank in terms of users registered and queries made. The test files should be named `test-n-reg.txt` and `test-n-commands.txt` where $1 \leq n \leq 15$. Here, `reg` represents the registration file and `commands` represents the operations and queries file. Your tests will be run against the buggy solutions with the `--verbose` flag as similar to the following, though we might use long or short versions of `-f` and `-v` (you do not need to submit `test-n-output.txt`):

```
$ ./bank -v -f test-n-reg.txt < test-n-commands.txt > test-n-output.txt
...
```

Test files may have no more than 8 users and no more than 32 transactions + queries (your activity / query file must not have more than 32 lines excluding comments and the separator). You may submit up to 15 pairs of test files (though it is possible to get full credit with fewer test files). The test cases the autograder runs on your solution are NOT limited to 8 users and 32 transactions; your solution should not impose any size limits (as long as sufficient system memory is available).

Errors you must check for

A small portion of your grade will be based on error checking. You must check for the following errors:

- A `place` command with a timestamp earlier than the previous `place`.
- A `place` command which contains an execution date before its own timestamp.

In all of these cases, print an informative error message to standard error (`cerr`) and call `exit(1)`. In the starter files, look at the file named Error_messages.txt. If you output any one of those standard error messages when the input is actually valid, the autograder will repeat the error message back to you. This can be helpful when your program thinks that a valid input is invalid, because you can more easily figure out what went wrong.

If you want to, you can add a second line below the first, to tell you what was wrong (such as the two timestamps in question). The autograder will not look at or show you this line, but it could be useful for debugging and when running with your own test files.

You do not need to check for any other errors.

Safe Assumptions

- We will not put extra characters after the end of a line of the registration file or after a command.
- Every `USER_ID` is unique and will be created once in the registration file.
- The timestamps given will be formatted correctly according to the invariants described above in the spec.
- Every IP address will be formatted correctly.

Submission to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file: `// Project Identifier: 292F24D17A4455C1B5133EDD8C7CEAA0C9570A98`
- The Makefile must also have this identifier (in the first TODO block).
- You have deleted all .o files and your executable(s). Typing `make clean` should accomplish this.
- Your makefile is called `Makefile`. Typing `make -R -r` builds your code without errors and generates an executable file called bank. The command line options `-R` and `-r` disable automatic build rules, which will not work on the autograder.
- Your makefile specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can often speed up code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with the debug flag, `-g`, as this will slow your code down considerably. If your code "works" when you compile without `-O3` and breaks when `-O3` is added, it means you have a bug in your code!
- Your test files are named `test-n-reg.txt, test-n-commands.txt` and no other project file names begin with test. Up to 15 pairs of test files may be submitted.
- The total size of your program and test files does not exceed 1MB.
- You do not have any unnecessary files or other junk in your submit directory and your submit directory has no subdirectories.
- Your code compiles and runs correctly using the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC running on CAEN Linux. We want you to use version 11.3.0 (available on CAEN with a command and/or Makefile); this version is also installed on the autograder machines.

Turn in all of the following files

- All your .h, .hpp, and .cpp files for the project
- Your Makefile

- Your test files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Our Makefile provides the command `make fullsubmit`. Alternately you can go into this directory and run this command:

```
$ COPYFILE_DISABLE=true tar -cvzf ./submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
...
```

This will prepare a suitable file in your working directory.

Submit your project files directly to the 281 autograder at: https://eecs281ag.eecs.umich.edu/ . You may submit up to three times per calendar day (more during the Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade.** If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form. **If you use an online revision control system, make sure that your projects and files are PRIVATE; many sites make them public by default! If someone searches and finds your code and uses it, this could trigger Honor Code proceedings for you.**

Please make sure that you read all messages shown at the top section of your autograder results! These messages will help explain some of the issues you are having (such as losing points for having a bad Makefile or using disallowed libraries).

Grading

All grading will be done by the autograder, with points distributed as follows.

80 points – Your grade will be primarily based on the correctness of your algorithms. Your program must work correctly in both verbose and non-verbose mode. **Additionally:** Part of your grade will be derived from the runtime performance of your algorithms. Fast running algorithms will receive all possible performance points. Slower running algorithms may receive only a portion of the performance points.

10 points – No memory leaks. Solutions that are proven to run (passing certain autograder test cases) will also be run through valgrind, to check for memory leaks at exit. This is something you should do yourself, to check for memory leaks as well as a number of other issues that valgrind can enumerate. Valgrind is available at the CAEN command prompt, and can also be downloaded to run on Linux-based personal environments.

10 points – Testing and Bug Finding (effectiveness at exposing buggy solutions with user-created test files).

When you start submitting test files to the autograder, it will tell you (in the section called “Scoring student test files”) how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!

Coding Style

Although your project will not be graded on style, style is a very important part of programming and software development. Among other things, good coding style consists of the following:

- Clean organization and consistency throughout your overall program
- Proper partitioning of code into header and cpp files
- Descriptive variable names and proper use of C++ idioms
- Effective use of library (STL) code
- Omitting globals, unnecessary literals, or unused libraries
- Effective use of comments
- Reasonable formatting - e.g. an 80 column display
- Code reuse/no excessive copy-pasted code blocks
- Effective use of comments includes stating preconditions, invariants, and postconditions, explaining non-obvious code, and stating big-Oh complexity where appropriate

It is **extremely helpful** to compile your code with the gcc options: `-Wall -Wextra -pedantic -Wconversion`. This will help you catch bugs in your code early by having the compiler point out when you write code that is either of poor style or might result in behavior that you did not intend.

A great resource for C++ style is the Google C++ Style Guide.

Hints and Advice

- Design your data structures and work through algorithms on paper first. Draw pictures. Consider different possibilities before you start coding. If you're having problems at the design stage, come to office hours. After you have done some design and have a general understanding of the assignment, re-read this document. Consult it often during your assignment's development to ensure that all of your code is in compliance with the spec.
- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this project and in this course, and coding before thinking often results in inefficient algorithms.
    - If you are considering linked lists, be sure to review the lecture slides or measure their performance against vectors first (theoretical complexities and actual runtime can tell different stories).
- Only print the specified output to standard output.
- You may print whatever diagnostic information you wish to standard error (`cerr`). However, make sure it does not scale with the size of input, or your program may not complete within the time limit for large test cases.
- Make sure that `main()` does a `return 0`.
- This is not an easy project. **Start it immediately!**

Have fun coding!

CREDITS

Originally composed by: Mert Gerdan, David Paoletti, Marcus Darden, and Ryan Chua (Toafu)