EECS 281 Winter 2014

Project 3: Market Maker

Due Tuesday, April 1st at 11:55pm (yes, it's REALLY due on April Fool's Day)

Overview

In this project, you will write an electronic stock exchange market. The market offers a variety of equities. Any market client can place a *buy* or *sell* order on an equity to request that a transaction be executed when matching sellers or buyers become available. Your program should take in *buy* and *sell* orders for a variety of equities as they arrive and match buyers with sellers to execute trades as quickly as possible. In addition, you will have the ability to create an insider trader allowing for instantaneous trades, letting your stock exchange make even more money on the side.

Project Goals

- Use core data structures: priority queues (`std::priority_queue`), hash tables (`std::unordered_map`, `std::unordered_set`, `std::unordered_multimap`, `std::unordered_multiset`), and binary search trees (`std::map`, `std::set`, `std::multimap`, `std::multiset`)
- Become more proficient with making design decisions
- Become more familiar with the STL and practice the ability to use it effectively

Input

The input will arrive from standard input (cin), **not** from an ifstream. There will be two input formats, *trade list* (`TL`) and *pseudorandom* (`PR`). The format is indicated by the first line of input --- which will be either `TL` or `PR` (never both) --- and applies to the entirety of the input.

For Trade List (TL) Input:

The input will contain a series of *orders* that will be presented such that orders with lower timestamps always appear first. Orders will be formatted as follows:

```
TIMESTAMP CLIENT_NAME BUY_OR_SELL EQUITY_SYMBOL $PRICE #QUANTITY
```

On a single line, with all fields separated by one or more spaces/tabs. For example:

```
TL
0 Jack  BUY  GOOG $100 #50
```

To avoid unexpected losses of large amounts of money, you must check for invalid input. Specifically, you must check that:

- Non-negative integer values are given for `TIMESTAMP`.
- Positive integer values are given for `PRICE`.
- All orders arrive in timestamp order (where 0 is the lowest possible timestamp value).
- Both the `$` and `#` signs appear where they are expected.
- The equity name must contain between 1-5 characters.
- The client name must not be empty and only contain alphanumeric characters and underscores.
- The `BUY_OR_SELL` field must only contain either the string “BUY” or “SELL”.
- Note that once a timestamp is read on a new line it can be assumed that all the following fields will be present in the line. You will not be given input files with partial orders.

If you detect invalid input at any time during the program, `exit(1)`. You do not need to check for other kinds of input errors - we will not be testing for errors not specified here.

Field Information:

- `TIMESTAMP` - Will be a non-negative integer value corresponding to the UNIX epoch time, where adding ‘1’ to the value means adding ‘1 second’. You should not try to interpret this timestamp into its corresponding date/time, as you will only need it for comparison, which can be conveniently done with the normal operators.
- `CLIENT_NAME` - The buyer or seller’s name. This will be a string that contains only alphanumeric characters and ‘_’ (see `std::isalnum` for a definition of alphanumeric).
- `BUY_OR_SELL` - The string `"BUY"` or the string `"SELL"` corresponding to the type of order.
- `EQUITY_SYMBOL` - The shorthand name of the equity. This will be a string that contains 1-5 characters that are alphanumeric or ‘_’ or ‘`.`’ (examples: `C`, `F`, `GM`, `KO`, `TGT`, `WMT`, `AAPL`, `PZZA`, `BRK.A`, `BRK.B`)
- `PRICE` - This is a positive integer (not zero). If it’s a buy order, this is the highest price a buyer is willing to pay for the equity. If it’s a sell order, this is the lowest price the seller is willing to sell the equity for. Buyers may pay less than their limit price, sellers might get more money than the minimum they ask for (see below). The $ sign will appear in the input before this value.
- `QUANTITY` - A positive integer number of stocks the client wants to buy/sell. The # sign will appear in the input before this value.

All valid input will be arranged by timestamp (lowest timestamp first). As you read in orders, you should assign all orders a unique ID number, such that the first order you read gets an ID of 0, the second an ID of 1, and so on. These ID numbers ensure that there is only one possible matching of buyers and sellers based on arrival timestamps. They are also useful for tracking and debugging.

For Pseudorandom (PR) Input:

To determine the timestamp of randomly generated orders, your program will make use of a `GENERATOR_TIMESTAMP`, whose initial value is always 0.

You will be given the following integer values, with each item on its own line:

- `RANDOM_SEED` - The value to initialize the random seed to.
- `NUMBER_OF_ORDERS` - The number of orders to generate. You may assume that this value will be less than or equal to $2^{32} - 1$.
- `LAST_CLIENT` - The name of the last client that will be trading. This value will be a character between ‘`a`’ and ‘`z`’.
- `LAST_EQUITY` - The of name of the last equity that will be traded. This value will be a character between ‘`A`’ and ‘`Z`’
- `ARRIVAL_RATE` - A double corresponding to the arrival rate, or approximately one over the average time between order arrivals. We are going to use the arrival rate to approximate a Poisson arrival process. A Poisson process assumes that the time of the next arrival is independent of the amount of time that has passed since the previous arrival. We can approximate a Poisson process by observing that the time between arrivals is exponentially distributed.

This information will be provided via standard input (cin) in exactly this format/arrangement, which you may assume **will always be correctly formatted**:

```
PR
RANDOM_SEED: 3453243
NUMBER_OF_ORDERS: 20
LAST_CLIENT: b
LAST_EQUITY: C
ARRIVAL_RATE: 1.8
```

For the above example file, 20 total orders will be generated, with clients "`C_a`" and "`C_b`" trading the "`E_A`", "`E_B`", and "`E_C`" equites (see below).

The procedure for generating your input in pseudorandom mode is as follows:

1. Include `<random>` in your header
2. Seed the the random number generator with `std::mt19937 gen(RANDOM_SEED);`
3. Initialize several distributions for the various random elements:

```
std::uniform_int_distribution<char> clients('a', LAST_CLIENT);
std::uniform_int_distribution<char> equities('A', LAST_EQUITY);
std::exponential_distribution<> arrivals(ARRIVAL_RATE);
std::bernoulli_distribution buy_or_sell;
std::uniform_int_distribution<> price(2, 11);
```

```
std::uniform_int_distribution<> quantity(1, 30);
```

4. For each randomly generated trade use the following pseudo code to generate all aspects of the order:

```
TIMESTAMP := GENERATOR_TIMESTAMP
GENERATOR_TIMESTAMP := GENERATOR_TIMESTAMP + floor(arrivals(gen) + .5)
CLIENT_NAME := string("C_") + clients(gen)
//Range of possible client names is thus 'C_a' to 'C_z'.
BUY_OR_SELL := (buy_or_sell(gen) ? BUY : SELL)
EQUITY_SYMBOL := string("E_") + equities(gen)
//Range of possible equity names is thus "E_A" to "E_Z".
PRICE := 5 * price(gen) // $10 - $55
QUANTITY := quantity(gen) // 1 - 30
```

Note: These fields should be computed in exactly this order to be consistent with the autograder.
All values have the same meaning as in Trade List (`TL`) inputs, and should be used in exactly the same way. ID numbers should be given in the same manner, except now they correspond to order of generation. If a client in the given range is not generated, then it should not produce any output.

By following the given procedure with the input above, you should generate the same order list as the following `TL` input:

```
TL
0 C_b SELL E_B $25 #26
1 C_b SELL E_C $25 #10
1 C_a SELL E_B $15 #22
1 C_a BUY  E_B $55 #7
2 C_a BUY  E_B $35 #22
2 C_b BUY  E_C $25 #26
2 C_b BUY  E_A $25 #23
3 C_a BUY  E_C $35 #21
3 C_b SELL E_A $55 #4
3 C_b BUY  E_B $30 #29
3 C_a BUY  E_B $20 #22
4 C_a BUY  E_B $10 #21
4 C_b SELL E_C $40 #2
4 C_b BUY  E_A $50 #5
4 C_b SELL E_B $45 #19
4 C_a SELL E_C $50 #26
5 C_a SELL E_C $40 #8
6 C_b BUY  E_C $40 #3
6 C_b SELL E_B $20 #30
7 C_a BUY  E_C $30 #11
```

Market Logic

`CURRENT_TIMESTAMP` should always start at 0, it is then maintained throughout the run of the program.

Your program must perform the following tasks:

1. Add any insiders into your clients data structures so that they are printed out using the --transfers flag at the *end of day* even if no trade is performed.
2. Read the next order from input
3. If the order's `TIMESTAMP != CURRENT_TIMESTAMP` then:
    a. If the `--median` option is specified, print the median price of all equities that have been traded on at least once by this point in lexicographical order by `EQUITY_SYMBOL` (see below).
    b. Set `CURRENT_TIMESTAMP` equal to the order's `TIMESTAMP` .
4. Add the order from step 2 to your data structures.
5. Process all possible trades with the information in your data structures (match buyers to sellers). If any orders cannot be filled, you should leave them in your data structures to match with future orders.
6. If there is an insider for the equity traded then perform the following steps:
    a. Determine if a buy should be performed given the current market price of the equity.
        i. If the buy should be performed, add the insider buyer to your data structures.
        ii. After adding the buyer, process all possible trades, as in step 5.
    b. Determine if a sell should be performed given the current market price of the equity.
        i. If the sell should be performed, add the insider seller to your data structures.
        ii. After adding the seller, process all possible trades, as in step 5.
7. Repeat previous steps 2-6 until the *end of the day*, defined as when there is no more input (and thus no more trades to be performed).
8. Treat the end of day like the timestamp has moved again, and output `median` information as necessary.
9. Print all *end of day* output.

Orders and Trades

All orders on the market are considered *limit* orders. A *buy limit* order expresses the intent to buy at most *N* shares of a given stock at no more than *D* dollars per share, where *D* is called the *limit*. A *sell limit* order expresses the intent to sell at most *N* shares of a given stock at no less than *D* dollars per share. To facilitate trade execution, an order can be split and matched with several other orders at different execution prices.

Order books and trade execution

For each equity, you should keep track of all current buy and sell orders in a dedicated container data structure called an *order book.*[1] A trade can be executed for a given equity when the *buy order* with the highest limit

[1] There is a considerable number of possible different data structures for the order book, including reusable STL data structures and hybrids. We recommend carefully analyzing several alternatives before committing to one. You may want to

price (and in the event of ties, lowest ID number) has a *buy limit price* greater or equal to the *sell limit price* of the *sell order* with the lowest limit price (and in the event of ties, lowest ID number). For a given order book, the order of trade execution is fully determined. You must support this order correctly and ensure high speed of trade execution by optimizing data structures.

For example, given the following orders as input:

```
0 SELLER_1 SELL GOOG $125 #10
0 SELLER_2 SELL GOOG $100 #30
0 SELLER_3 SELL GOOG $100 #15
0 BUYER_1  BUY  GOOG $200 #4
0 BUYER_2  BUY  GOOG $250 #50
0 SELLER_4 SELL GOOG $60  #20
```

The *first* trade to be executed would be `BUYER_1` buying 4 of `SELLER_2`'s shares. This is because at this point, the program has not read `BUYER_2`'s order yet (see market logic section) and `SELLER_2` has the lowest sell price. While `SELLER_2` and `SELLER_3` are both selling at the same price, `SELLER_2`'s order arrived first, and will therefore have a lower ID number.

When that first trade is executed, `SELLER_2`'s order must be revised to offer only 26 shares (because 4 had already been traded). The revised order keeps the id of the original order.

Whenever a trade is executed, the *match price* of the trade is the limit price of the order (buy or sell) with the lower ID number. In this case, `BUYER_1` offered to buy for $200 and `SELLER_2` offered to sell for $100; because `SELLER_2` has a lower ID number, the trade will be executed at a match price of $100 per share.

Commission fees and trade reporting

For providing this matching service, the market (and thus your program) also takes a commission fee of 1% from every completed trade from **both** the buyer and the seller. The commission in our example is:
$(100 \cdot 4)/100 = \$4$ from both the buyer and seller.

And so `BUYER_1` will pay $(100 \cdot 4) + 4 = \$404$, `SELLER_2` will receive $(100 \cdot 4) - 4 = \$396$, and The Market will earn a commission of $(2 \cdot 4) = \$8$. For these calculations, all values should be integers and therefore all decimals will be truncated. Commissions must be computed exactly as above when executing the trade; to be clear, do **not** calculate the combined commissions of the buyer and seller in a single arithmetic expression (such as $total\_commission = match\_price \cdot num\_shares/100 \cdot 2$), as this may yield different results when truncating. First calculate the commission as shown, then multiply the result by 2 instead.

experiment with several data structures, so keep the interface sufficiently independent of the implementation. Sometimes it may be useful to switch from one data structure to another dynamically, but this is not a requirement.

Command-Line Input

Your program `market` should take the following case-sensitive command-line options:

- `-s, --summary`
  An optional flag that indicates the program should print a variety of summary information describing the day's trades (See *Output* section below).
- `-v, --verbose`
  An optional flag that indicates the program should print additional output information while trades are being executed (See *Output* section below).
- `-m, --median`
  An optional flag that indicates the program should print the current median match price for each equity at the times specified in the Market Logic' section above, i.e. during execution.
- `-t, --transfers`
  An optional flag that indicates the program should print additional output information at the end of the day to show the net amount of funds transferred by all clients (see *Output* section below).
- `-i, --insider EQUITY_SYMBOL`
  An optional flag that may appear more than once with different equity symbols as arguments. The insider option allows for the program itself to have an advantage over other buyers and sellers by being able to instantaneously insert its own buys and sells into the input queue. More information in the *Insiders* section.
- `-g, --ttt EQUITY_SYMBOL`
  An optional flag that may appear more than once with different equity symbols as arguments. The Time-Travel Trading option requests that, at the end of the day the program determines what was the best time to buy (once) and then subsequently sell (once) a particular equity during the day to maximize profit. More information in the output section.
- `-h, --help`
  An optional flag that indicates you should print a help message and then exit, regardless of any other command line options.

If multiple options are specified that produce output at the end of the program, the output should be printed in the order that they are listed here in the spec (e.g., --summary before --transfers before --ttt).

Examples of legal command lines:

- `./market < infile.txt > outfile.txt`
- `./market --verbose --transfers --summary > outfile.txt`
- `./market --verbose --median > outfile.txt`
- `./market --ttt GOOG -i IBM`
- `./market --transfers -s --verbose --ttt GOOG --ttt IBM --insider GOOG`
- `./market -v -m -t -i GOOG -g IBM -s`
- `./market --help -t` (Note that -t will not run, but still valid input)

We will not be specifically error-checking your command-line handling, however we expect that your program conforms with the default behavior of getopt_long. Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

Output

Default

At the **end of the day**, after all input has been read and all possible trades completed, the following output should *always* be printed before any optional end of day output:

```
---End of Day---
```

Summary

If and only if the `--summary` **option** is specified on the command line, you should print the following information giving a summary of the day's trades:

```
Commission Earnings: $COMMISION_EARNINGS
Total Amount of Money Transferred: $MONEY_TRANSFERRED
Number of Completed Trades: NUMBER_OF_COMPLETED_TRADES
Number of Shares Traded: NUMBER_OF_SHARES_TRADED
```

Verbose Option

If and only if the `--verbose` **option** is specified on the command line (see above), whenever a trade is completed you should print:

```
BUYER_NAME purchased NUMBER_OF_SHARES shares of EQUITY_SYMBOL from SELLER_NAME
for $PRICE/share
```

on a single line. In the following example:

```
0 SELLER_1 SELL GOOG $125 #10
0 SELLER_2 SELL GOOG $100 #10
0 SELLER_3 SELL GOOG $100 #10
0 SELLER_3 SELL GOOG $80  #10
0 BUYER_1  BUY  GOOG $200 #4
```

you should print:

```
BUYER_1 purchased 4 shares of GOOG from SELLER_3 for $80/share
```

No trades can be executed on an equity for a given `CURRENT_TIMESTAMP` if there is no buyer willing to pay the lowest asking price of any seller. An example of such a scenario is:

```
0 BUYER_1  BUY  GOOG $100 #50
0 SELLER_1 SELL GOOG $200 #10
0 BUYER_2  BUY  GOOG $150 #3
0 SELLER_2 SELL GOOG $175 #30
```

Median Option

If and only if the **`--median` option** is specified on the command line, at the times described in the Market Logic section (above), your program should print the current median match price of all completed trades for each equity that were executed in the time interval [`0`, `CURRENT_TIMESTAMP`]. To be clear, this is the median of the match prices of the trades themselves. This does not consider the quantity traded in each trade. Equities with lexicographically smaller `EQUITY_SYMBOL`s must be printed first. If no matches have been made on a particular equity, do not print a median for it. If there are an even number of trades, take the average of the middle-most two to compute the median. The output format is:

```
Median match price of EQUITY_SYMBOL at time CURRENT_TIMESTAMP is $MEDIAN_PRICE
```

Transfers Option

If and only if the **`--transfers` option** is specified on the command line, you should print the following information for each client who placed an order during the run of the program:

```
CLIENT_NAME bought NUMBER_OF_STOCKS_BOUGHT and sold NUMBER_OF_STOCKS_SOLD for a
net transfer of $NET_VALUE_TRADED
```

This should be printed such that clients with lexicographically smaller client names are printed first (see `operator<` for `std::string`'s). The `NET_VALUE_TRADED` does not include commissions.

Time-Travel Trading Option

If and only if the **`--ttt` option** (Time-Travel Trading) is specified on the command line, you should do the following:

If the `--ttt` option is specified more than once, you should print the results for each `EQUITY_SYMBOL` that is given *in the same order that they were given in the command line*. In other words, if the command line had both `--ttt MSFT` and then `--ttt IBM`, you would print `MSFT`'s information before `IBM`'s. We will not specify the same equity twice.

In time travel trading, you are a time traveler that wants to find the ideal times that you "could have" bought an equity and then later "could have" sold that equity to maximize profit[2] (or if it is not possible to profit, to do this while minimizing losses). Your program will print a `TIMESTAMP1` and `TIMESTAMP2` corresponding to the times you "could have" placed orders to do this.

What this means is that `TIMESTAMP1` will be the same as some *actual* sell order that came in during the day, and that `TIMESTAMP2` will be the same as some *actual* buy order that came in *after the sell order* (with a higher ID number). The assumption is that the time traveler "would have" placed those orders immediately after the *actual* orders.

When calculating the results for time travel trading, the only factors are the time and price of orders that happened throughout the day. Quantity is not considered. One way to think about this is to imagine (only for the purpose of time travel trading) that all orders are for unlimited quantity.

If there would be more than one answer that yields the optimal result, you should prefer answers with lower timestamps. If during the day there is not at least one *actual* sell order followed by at least one *actual* buy order, then `TIMESTAMP1` and `TIMESTAMP2` should both be printed as -1. Make sure to also consider insider trades if an insider is used.

The output format is as follows:

```
Time travelers would buy EQUITY_SYMBOL at time: TIMESTAMP1 and sell it at time:
TIMESTAMP2
```

Help Option

If and only if the `--help` **option** is specified on the command line, you should print a helpful message about the usage to `std::cout` and then exit(0) the program without executing any trades.

[2] The 'Best Time to Buy and Sell Stock' problem at http://leetcode.com/onlinejudge corresponds to what we are asking for. The site includes a free autograder, so you can develop and test this algorithm separately from the rest of your project.

Insiders

Creation

If and only if the **`--insider` option** is specified on the command line, you should do the following:

The `--insider` option can be specified more than once in order to use an inside trader on more than one equity. However, we will never specify the same equity twice.

Even though you will be making a fair amount of money through the commissions collected in your program, you would like to use your control of the market to your advantage. Because you are at the control of this equity trading program, you have the ability to insert new orders immediately into the queue after reading each new order being placed. This will give you an (unfair…) advantage to be able to react faster than everyone else to the market. Note however, that you will be unable to change what orders have already have been placed in the past and cannot supercede them.

An insider behaves just like any other client and should be added into your client data structure at the beginning of program execution, as they do not appear in the input file stream. The name of the insider client will be the following format:

```
"INSIDER_" + EQUITY_SYMBOL
```

For example, if the flag `--insider AMD` was given on the command line then an insider client named `INSIDER_AMD` would be created. **Note that this means that client names that begin with `INSIDER_` are reserved.**

Logic

After an order from the input stream has been read in and processed (see Market Logic) the program will check to see if an insider exists for the recently traded equity. Next, it checks to make sure that a valid median trade value exists, as calculated in the *Median* section. Otherwise, it does not attempt a buy or sell.

Note that in this system, profit is defined as the median value of the equity versus the actual price paid to you or the seller. For example, if the median of an equity is $90 and you pay $70 for its purchase, your profit is $20. Even though you lost money, you potentially gained more value in the equity. **Note that profits are calculated on a per-share basis**.

The insider will then attempt to add a single buy immediately to the input queue at the current timestamp. Check the current market buy price per share (the minimum sell price) and also determine the number of shares still available for this order. If the profit of this transaction would be greater than 10% of the median, add the insider client as a buyer of all remaining shares and immediately match clients to one another. **Note that the insider**

will behave just like any other client (e.g. update median calculation).

The insider will also then attempt to add a single sell immediately to the input queue at the current timestamp. Check the current market sell price per share (the maximum buy price) and also determine the number of shares still available for this order. If the profit of this transaction would be greater than 10% of the median, add the insider client as a seller of all remaining shares and immediately match clients to one another.

Note that the client does not have a budget either in terms of money or shares of its equity and can sell and buy as it desires.

Output

Note that because an insider behaves the same as any other client, the same output will be expected for the `--transfers` and `--verbose` options. In addition, note that all trades should be performed exactly the same with a portion of the profits being paid to commission. You can attribute this to the need to not raise any suspicions by having a certain client receive special privileges (besides the early access bit).

Insider Output Example

Input:

```
TL
0 PlanetExpress SELL AMD  $120 #32
1 BadWolfCorp   BUY  AMD  $150 #20
2 BluthCorp     SELL AMD  $100 #50
3 KrustyKrab    BUY  AMD  $200 #10
4 PlanetExpress BUY  AMD  $210 #50
5 BadWolfCorp   BUY  AMD  $205 #70
6 BluthCorp     BUY  AMD  $210 #30
7 KrustyKrab    BUY  AMD  $205 #40
8 PlanetExpress SELL AMD  $155 #25
```

Output when run with `--verbose`, `--median`, `-i AMD`:

```
BadWolfCorp purchased 20 shares of AMD from PlanetExpress for $120/share
Median match price of AMD at time 1 is $120
INSIDER_AMD purchased 50 shares of AMD from BluthCorp for $100/share
Median match price of AMD at time 2 is $110
KrustyKrab purchased 10 shares of AMD from PlanetExpress for $120/share
Median match price of AMD at time 3 is $120
PlanetExpress purchased 2 shares of AMD from PlanetExpress for $120/share
PlanetExpress purchased 48 shares of AMD from INSIDER_AMD for $210/share
Median match price of AMD at time 4 is $120
BadWolfCorp purchased 70 shares of AMD from INSIDER_AMD for $205/share
Median match price of AMD at time 5 is $120
```

```
BluthCorp purchased 30 shares of AMD from INSIDER_AMD for $210/share
Median match price of AMD at time 6 is $120
KrustyKrab purchased 40 shares of AMD from INSIDER_AMD for $205/share
Median match price of AMD at time 7 is $162
Median match price of AMD at time 8 is $162
---End of Day---
```

Again, note that the median is based on the per-share price of each enacted trade, not the number of orders placed.

Full Output Example

Input:

```
TL
0 PlanetExpress SELL AMD  $120 #32
0 BadWolfCorp   BUY  GE   $200 #20
0 BluthCorp     BUY  AMD  $100 #50
1 KrustyKrab    BUY  AMD  $130 #10
1 PlanetExpress SELL GE   $150 #50
1 PlanetExpress BUY  NFLX $80  #15
3 BluthCorp     SELL AMZN $50  #22
4 BadWolfCorp   SELL GE   $50  #15
4 BadWolfCorp   SELL AMZN $100 #30
4 KrustyKrab    BUY  AMZN $130 #12
4 BadWolfCorp   BUY  AMZN $50  #30
5 BadWolfCorp   SELL AMZN $50  #5
5 BluthCorp     BUY  AMD  $150 #25
6 PlanetExpress SELL AMD  $80  #100
6 BadWolfCorp   BUY  AMD  $120 #10
6 KrustyKrab    BUY  GE   $110 #10
6 BluthCorp     BUY  GE   $200 #25
```

Output when run with `--verbose`, `-s`, `--median`, `--transfers`, `-i AMD`, `-i GE`, and `--ttt AMZN`:

```
KrustyKrab purchased 10 shares of AMD from PlanetExpress for $120/share
BadWolfCorp purchased 20 shares of GE from PlanetExpress for $200/share
INSIDER_GE purchased 30 shares of GE from PlanetExpress for $150/share
Median match price of AMD at time 1 is $120
Median match price of GE at time 1 is $175
Median match price of AMD at time 3 is $120
Median match price of GE at time 3 is $175
INSIDER_GE purchased 15 shares of GE from BadWolfCorp for $50/share
KrustyKrab purchased 12 shares of AMZN from BluthCorp for $50/share
```

```
BadWolfCorp purchased 10 shares of AMZN from BluthCorp for $50/share
Median match price of AMD at time 4 is $120
Median match price of AMZN at time 4 is $50
Median match price of GE at time 4 is $150
BadWolfCorp purchased 5 shares of AMZN from BadWolfCorp for $50/share
BluthCorp purchased 22 shares of AMD from PlanetExpress for $120/share
BluthCorp purchased 3 shares of AMD from INSIDER_AMD for $150/share
Median match price of AMD at time 5 is $120
Median match price of AMZN at time 5 is $50
Median match price of GE at time 5 is $150
BluthCorp purchased 50 shares of AMD from PlanetExpress for $100/share
INSIDER_AMD purchased 50 shares of AMD from PlanetExpress for $80/share
BluthCorp purchased 25 shares of GE from INSIDER_GE for $200/share
Median match price of AMD at time 6 is $120
Median match price of AMZN at time 6 is $50
Median match price of GE at time 6 is $175
---End of Day---
Commission Earnings: $574
Total Amount of Money Transferred: $28890
Number of Completed Trades: 12
Number of Shares Traded: 252
BadWolfCorp bought 35 and sold 20 for a net transfer of $-3750
BluthCorp bought 100 and sold 22 for a net transfer of $-11990
INSIDER_AMD bought 50 and sold 3 for a net transfer of $-3550
INSIDER_GE bought 45 and sold 25 for a net transfer of $-250
KrustyKrab bought 22 and sold 0 for a net transfer of $-1800
PlanetExpress bought 0 and sold 182 for a net transfer of $21340
Time travelers would buy AMZN at time: 3 and sell it at time: 4
```

Libraries and Restrictions

We highly encourage the use of the STL for this project, with the exception of two prohibited features: The C++11 regular expressions library (whose implementation in gcc 4.7 is unreliable) and the thread/atomics libraries (which spoil runtime measurements). **Do not** use other libraries (e.g., boost, pthreads, etc).

Testing and Debugging

Part of this project is to prepare several test cases that will expose defects in buggy solutions - your own or someone else's. As this should be useful for testing and debugging your programs, **we strongly recommend** that you **first** try to catch a few of our intentionally-buggy solutions with your test cases, before completing your solution. The autograder will also tell you if one of your own test cases exposes bugs in your solution.

Each test case should consist of an input file. When we run your test cases on one of intentionally-buggy project solutions, we compare the output to that of a correct project solution. If the outputs differ, the test case is said to expose that bug.

Test cases should be named either:

`test-n-FLAG.txt`

or

`test-n-FLAG-EQUITY_SYMBOL.txt`

where 0 < `n` <= 15, `FLAG` is one (and only one) of `v`, `m`, `t`, `i`, `s`, or `g`, and `EQUITY_SYMBOL` is a string that would be allowed (as stated in the input section) as a possible equity symbol (only for `i` or `g` flags).
For example: `test-1-w.txt` and `test-2-g-AMZN.txt` are both valid test case names. **Note that whenever the `-i` is selected, the `-v` will also be automatically be selected.** This is because `-i` by itself will not generate any output. Note that when running the program normally, `-i` by itself (plus its equity name) is still considered a valid flag set.

Your test cases **must be Trade List input files**, and may have no more t Than 30 lines in any one file. You may submit up to 15 test cases (though it is possible to get full credit with fewer test cases). Note that the tests on which the autograder runs your solution are **NOT** limited to 30 lines in a file; your solution should not impose any size limits (as long as sufficient system memory is available).

When debugging, we highly recommend setting up your own system for quick, automated, regression testing. In other words, check your solution against the old output from your solution to see if it has changed. This will save you from wasting submits. You may find the Linux utility ‘diff’ useful as part of this.

Submitting to the Autograder

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

- You have deleted all .o files and your executable(s). Typing ‘`make clean`’ shall accomplish this.
- Your makefile is called Makefile. Typing ‘`make -R -r`’ builds your code without errors and generates an executable file called "`market`". (Note that 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 speed up code by an order of magnitude.
- Your test case files are named as described above and no other project file names begin with test. Up to 15 test cases may be submitted.
- The total size of your program and test cases does not exceed 2MB.
- You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (i.e. the .git folder used by git source code management).
- Your code compiles and runs correctly using version 4.7.0 of 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 4.7.0 running on Linux (students using other compilers and OS did observe incompatibilities). Note****: To compile with g++ version 4.7.0 on CAEN you **must** put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-4.7.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-4.7.0/lib64
LD_RUN_PATH := /usr/um/gcc-4.7.0/lib64
```

Turn in all of the following files:

- All your .h and or .cpp files for the project
- Your Makefile
- Your test case 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. In this directory, run

```
dos2unix -U *; tar -czf ./submit.tar.gz *.cpp *.h Makefile test-*.txt
```

This will prepare a suitable file in your worm rking directory.

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. **Note that when the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). We will count only your last submission for your grade. Part of the programming skill is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. We realize that it is possible for you to score higher with earlier submissions to the autograder; however this will have no bearing on your grade. We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you 'commit' your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and CTools regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to note if the autograder shows that one of your own test cases exposes a bug in your solution (at the bottom).

Grading

90 points -- Your grade will be derived from correctness and performance (runtime). Details will be determined by the autograder.
10 points -- Test case coverage (effectiveness at exposing buggy solutions).

We also reserve the right to deduct up to 5 points for bad programming style, code that is unnecessarily duplicated, etc.

Refer to the Project 1 spec for details about what constitutes good/bad style.

Hints and Advice

The project is specified so that the various pieces of output are all independent. We strongly recommend working on them separately, implementing one command-line option at a time. The autograder has some test cases which are named so that you can get a sense of where your bugs might be; there's a MED case, a TTT case, a Transfers case, a Verbose case, etc.

We will be placing a stronger emphasis on time budgets in this project. This means that you may find that you need to rewrite sections of code that are performing too slowly or change data structures. Pay attention to the big-Oh complexities of your implementation and examine the tradeoffs of using different possible solutions.

Running your code locally in valgrind can help you find and remove undefined (buggy) behavior and memory leaks from your code. This can save you from losing points in the final run when you mistakenly believe your code to be correct.

It is extremely helpful to compile your code with the following gcc options: -Wall -Wextra -Wconversion -pedantic. This way the compiler can warn you about poor style and parts of your code that may result in unintended/undefined behavior.

Make sure that you are using getopt_long for handling command-line options.

Appendix A: Integer Calculations

Obey the following rules when making calculations in this program:

1. The term integer in this document speaks to 'integer representation'. For the purpose of this project, you should always use the C++ type **long long** to represent integers and avoid overflow issues. **Never use floats or doubles in this project**, and in particular do not use library functions for rounding to integers.
2. When order of operations can be changed, always perform multiplications before divisions, unless explicitly requested otherwise.
3. Use the closest representation of the formulas we provide in the spec in your code. Do not attempt to combine calculations or take shortcuts. We expect that the main speed-ups in this project are going to be in data structures rather than in arithmetic operations (at least when the number of trades is large).

Appendix B: Patch Notes

v1.11 (3/26/14):
Corrected typo in test case section: `-s` is also a valid flag to be passed to test cases.

v1.10 (3/19/14):
Added note about how `-v` is automatically included in test cases with the `-i` flag. Added clarifications and example to the insider section.

v1.00 (3/11/14):
Released.