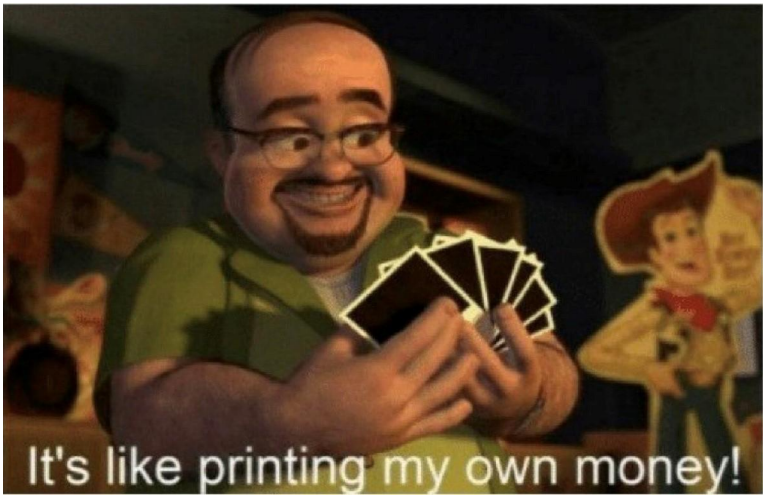
Algorithmic Trader

Throughout this course, we have studied a variety of data structures and associated algorithms. Optimising time (and space) complexity has been a real challenge and a seemingly fruitless one at that. Let us try and witness one real-life example where efficient algorithms are essential - An Algorithmic Trader!

Le sophies after this project -



Introduction

Excusing our poor sense of humour, let us start without further ado. So, what is an algorithmic trader? More precisely, what even is trading? A simple definition of trading is buying and selling. But what do we buy and sell? And how do we make money doing it? Good questions, padawans.
Well, algorithmic trading essentially uses a computer program that follows a defined set of instructions (ergo, algorithm) to place an order. The order, when executed, results in a trade that, in theory, can generate profits at a speed and frequency that is impossible for a human trader. Algorithmic trading, thus, is born from an amalgamation of computer programming and financial markets to execute trades at precise moments. What we focus on trading are financial instruments, particularly stocks and other structures born out of stocks. One last question: where does this trading occur? An exchange is a marketplace where securities, commodities, derivatives and other financial instruments are traded. Orders get sent to the exchange, which expresses the wish of an individual to buy or sell some entity. A trade occurs when a buy order and sell order agree on a price for a particular entity. The buyer buys from the seller.

Our end goal is to have a trader capable of executing trades. We can access data across networks, utilise it to make correct trading decisions and execute them by placing the appropriate orders. All these terms and ideas will become clearer through the problem statement. Let us start with the first phase, focussing on trading decisions (placing the correct orders) without any execution involved.

Phase One

The first step towards making an Autotrader is to process information from the exchange and place orders based on this information. Here, no execution of trades is involved. (How would this affect your approach to designing an algorithm?).

Part 1 - Buying Low, Selling High

Suppose we estimate the value of a stock as S. Then, we are more than happy to buy the stock at a price lower than S and sell a stock at a price higher than S. We use a very simple estimator for the price of a stock - the last price we **traded** on the stock. One small caveat remains - what should our first trade on a stock be? Since we do not have an estimate before the first trade, we will always trade with the first order we see on any stock.
Indeed, this seems like a decent strategy, considering it will make money as the price fluctuates. As the price goes down, we will buy and sell as the price goes up. However, is this strategy guaranteed to make us money? Think why or why not.

Input data orders are expressed as
<stock_name> <price> <s/b>#
(each line ends in a #), with each line being an available trade, where s stands for sell and b for buy.

Your output should contain **one line per line of input** indicating “No Trade” or the appropriate trade you wish to do (as <stock_name> <price> <sell/buy>), expressed as an order. Suppose you want to accept the order - ASML 100 b#. Your output should be ASML 100 s. **Each input line should correspond to a unique line in the output.**

Part 2 - Arbitrage

Consider the following example. Individual A is willing to buy the package deal of Stock of company X “minus” the Stock of company Y for 10 dollars. Individual B is willing to pay 20 dollars to buy Company Y's stock and sell Company Z's stock as a package deal. Individual C is willing to buy company Z’s stock and sell company X’s stock for 5 dollars. “This is the best trade deal in the history of trade deals, maybe ever.” Why? If you take all three deals, you have neither bought nor sold any stock but made off with 5 dollars. This is arbitrage.

Now, suppose our market also has such package deals expressed as linear combinations of stocks being bought and sold. Your algorithm should be able to detect all such opportunities where the trade deals executed collectively result in no stock being bought or sold, and yet, a net profit. In the case of many such possibilities, you should choose the one which maximises profit. Each order should be output similar to part 1 (but with stock_name replaced with the entire stock structure linear combination).

Input data stream will consist of lines in the following format.
<stock_name_1> <quantity_1> <stock_name_2> ….<quantity_n> <price> <s/b>#

with each line being an order, where there are n stocks in the linear combination of this stock structure (each quantity must be a non-zero integer)

Note that you cannot create your own linear combinations (that's a part of quoting, not trading). Also, **you may have to output multiple orders after a particular input trade**. (the combination of trades that maximises your projected profits) Output them in reverse order of what they were created in, with one trade per line (Think why!). Finally, output your (projected) profit on a new line. (Why is this projected and not actual profit? cue phase 2).

Part 3 - Order Book Processing

The above strategy is good because it guarantees us a non-negative profit. However, it is still incomplete. We should harness the full power of the market's information. In this part, we will allow each order to have more than one quantity of each deal available. Additionally, we will also store the entire order book. That is, an order will come in that can have multiple stocks or stock structures to buy/sell at a particular price. For instance, ASML 1 100 10 b# is a deal to buy 10 stocks of ASML for $100/stock. Also, a buy order of ASML 1 120 10 b# will NOT cancel this order. An order of ASML 1 100 20 s# will cancel the ASML 1 100 10 b# order and be reduced to ASML 1 100 10 s#, and will NOT affect the ASML 1 120 10 b# order (another way in which our algorithm can make money).

Data is expressed as
<stock_name_1> <quantity_1> <stock_name_2> ….<quantity_n> <price> <quantity> <s/b># with each line being an available trade, where there are n stocks in the linear combination of this stock structure (each quantity must be a non-zero integer) (for Part 3)

Your goal is to find optimal orders in this setting. **Output them similarly to part 2 above, except the quantity is also added.** For instance, you will enter ASML 100 20 s to sell 20 ASML stock at 100. Output your projected profit at the end again on a new line.

Information Line - How Data is Input to Our Autotrader

The information line in iml.cpp carries information from the exchange to the Autotrader. This information comprises the orders being made available in the market. It is passed through a socket as a stream of data. The socket is to be used by your trader to read. Each line has information about a new order. This has already been implemented. The Receiver rcv and function rcv.readIML() have been provided, which allows you to access the iml data periodically. A dollar indicates the end of the input stream
Some (obvious) cases to handle (across phase1): Two outstanding orders for the same stock, with one being 'buy' and the other being 'sell', cancel each other out if the sell price is **exactly equal** to the buy price. (strange restriction; will be relaxed in phase 2). For parts 1 and 2, we only maintain the best price for each "side". That is, if there is a buy order of 100 dollars and a buy order comes in for 120 dollars, the order of 100 dollars is deleted, even if it was not satisfied. (However, we consider it for part 3; obviously)

Putting it Together

Keep all input in “data.txt”. [1-3] is a command line argument indicating the part. Please modify only phase1/trader.cpp for implementing all three parts.

Please read the given Makefile. Use the following commands to run your program.

```
make runpart1
make runpart2
make runpart3
```

Phase Two

Are we done? Do we have a running algorithm guaranteed not to lose money? Not quite. Throughout the previous phase, we have mentioned that no execution is taking place. What does this mean? Now, we will implement a market and a trader in two parts.

Part 1 - Making the Market

Execution of orders, which results in trades, seeks to optimise price for all customers and seeks to maximise trading.
In this part, our goal is to model the market. So, how does the market work:-

- The market processes a trace file of orders input by various clients at different times.
- Each buy and sell order is stored until expiry. An order looks like this:

  BrokerName SELL STOCK $120 #32 1

  where BrokerName is the client's name, and BUY/SELL is the type of order requested. STOCK is the stock name. $120 is the price. #32 is the quantity, and 1 is the time to expiration. In place of stock name, we can also have linear combinations of stocks (as described in phase 1).
- Each order is also associated with a time, representing the timestamp when the order came in. (assumed to be an integer multiple of some arbitrary least count). After expiration, and only after expiration, does this order become invalid. An expiry of -1 implies the order will never expire, while an expiry of 0 means it is either filled immediately or cancelled. Suppose we had the following order:

  1 KarGoCorp BUY GE $200 #20 8

  This order came at timestep 1 and expiration 8. Hence, it will expire at t=9. Hence, any order placed at t=10 and after cannot trade with this order.
- Clients must NOT create opportunities to arbitrage on their own. This implies that no single client must have a sell order at a lower price than a preceding unexpired buy order and vice versa. Remember that this applies to us as well. 🙂
- The market performs a priority-based matching between buyers and sellers. Priority is given first based on price, then based on time, and finally based on alphabetical order.

An example trace is presented below (cropped from phase2/samples/output.txt provided to you)

```
0 KarGoCorp SELL AMZN $100 #30 10
0 KarGoExpress SELL AMD $120 #32 1
1 KarGoKrab BUY AMZN $130 #12 0
1 KarGoCorp BUY GE $200 #20 8
2 KarGoCorp BUY AMZN $50 #30 5
2 KarGoTravels BUY AMD $100 #50 10
2 KarGoCorp SELL AMZN $50 #5 0
3 KarGoKrab BUY AMD $130 #10 7
4 KarGoTravels BUY AMD $150 #25 0
4 KarGoExpress SELL GE $150 #50 6
5 KarGoExpress SELL AMD $80 #100 -1
5 KarGoExpress BUY NFLX $80 #15 6
5 KarGoCorp BUY AMD $120 #10 1
6 KarGoTravels SELL AMZN $50 #22 -1
6 KarGoKrab BUY GE $110 #10 3
7 KarGoCorp SELL GE $50 #15 -1
!@
```

```
Successfully Initiated!
KarGoKrab purchased 12 share of AMZN from KarGoCorp for $100/share
KarGoCorp purchased 5 share of AMZN from KarGoCorp for $50/share
KarGoCorp purchased 20 share of GE from KarGoExpress for $200/share
KarGoKrab purchased 10 share of AMD from KarGoExpress for $130/share
KarGoTravels purchased 50 share of AMD from KarGoExpress for $100/share
KarGoCorp purchased 10 share of AMD from KarGoExpress for $80/share
KarGoCorp purchased 22 share of AMZN from KarGoTravels for $50/share
KarGoKrab purchased 10 share of GE from KarGoCorp for $110/share

---End of Day---
Total Amount of Money Transferred: $14750
Number of Completed Trades: 8
Number of Shares Traded: 139
KarGoTravels bought 50 and sold 22 for a net transfer of $-3900
KarGoExpress bought 0 and sold 90 for a net transfer of $11100
KarGoCorp bought 57 and sold 27 for a net transfer of $-3600
KarGoKrab bought 32 and sold 0 for a net transfer of $-3600
```

NOTE: - On the left, we have a trace file, as seen by the market and on the right, we have the market results as printed. Note that there are a few caveats that we have not mentioned here, as they are for you to discover, and we expect you to cover them during the implementation of the market. You are encouraged to craft suitable test cases. The final Profit/Loss is calculated by equating the price of stock to the median price of all trades. Input to the market must be from “output.txt”. Refer to main.cpp to understand how the execution step is happening. Inappropriate input should also be handled appropriately. **You can see the samples folder for a few input-output pairs to get a better (albeit not necessarily) and complete understanding of all the cases that your market must handle.**

Run your code using the following

```
make market
```

Part 2 - Implementing a trader using “Market Making” strategy

In this part, we improve the trading strategies that we developed in the previous part to optimise execution and trading decisions jointly. Now, the market which you implemented above, functions on trace files. This trace file is produced by multiple agents placing orders in parallel. Much like in the actual market, our trader will be one such agent. Hence, we will also be supplying orders to output.txt. The salient points of our strategy will be

- Insert orders to exploit arbitrage opportunities as in phase 1, except that there will be no arbitrage in the form of a buy price being higher than a sell price for a given stock
- Each order must be inserted with the expiration equal to the minimum time to expiry left amongst all the orders we wish to accept. Additionally, ensure that you are not creating self-arbitrage opportunities, as described above.
- Additionally, maintain a moving median price (much like you have done for the market, right? Right?), sell above the median, and buy below the median.
- As an observation, compute the difference between your projected profit and the actual profit and try to explain this difference (this will not be graded, obviously).
- Use the client name corresponding to your team - rollno1_rollno2 (or rollno for solo)

The Dilemma

In this part, the core strategies remain the same as in the previous part. However, we must sacrifice some power of the previous strategies to better our execution. Hence, in this part, you must ONLY determine arbitrage, in time asymptotically polynomial in input magnitude. This could result in reduced profits, but that's the price we must pay for execution.
Note:- This stage will involve a fair amount of stochasticity; thus, do not expect the same output every time. This will be accounted for during grading. However, it does make testing your code a little bit more challenging, but we do love challenges, don't we? 😉

Much like the above part, you can run the code as follows

```
make trader
```

Phase Three

Phew! That has laid down the foundation for a large part of the Autotrader. We, now, have the capacity to make optimal trading decisions as well as execute them on the market by placing priority-sensitive orders (albeit with a tradeoff). But yet, one essential component is missing. And that missing component is….drumroll...statistical arbitrage. Oh! Arbitrage again? "Statistical arbitrage (or stat arb) refers to a group of trading strategies that utilise mean reversion analyses. Statistical arbitrage strategies are market-neutral because they simultaneously open long and short positions to take advantage of inefficient pricing in correlated securities. Known as a deeply quantitative, analytical approach to trading, stat arb aims to reduce exposure to beta as much as possible across two phases: "scoring" provides a ranking to each available stock according to investment desirability, and "risk reduction" combines desirable stocks into a specifically-designed portfolio aiming to lower risk." That was a lot to take in! Let us simplify.

Consider we have correlated stocks. For simplicity, we will consider a stock that has a unit correlation with itself. What does this mean? It means that if the price of a stock goes up in one market, it should go up in another. But does it? In this phase, we will be performing arbitrage across markets. There will be different markets, and we have different information lines providing data across different markets to us. Your goal is to process this information obtained from different markets and identify opportunities for arbitrage. You will output the corresponding order in all the markets. For instance,

Suppose we get an input of 3 ASML 100 10 b# from one market and 4 ASML 95 20 s# from another market. Then, we must output 4 ASML 100 10 s# in one trace file for one market and 4 ASML 95 10 b# in the other, assuming both orders are available in both markets at t=4. Print ONLY your projected profit to standard output. Trace file i is named "output<i>".txt

Input/Output Details

Input is taken from inputs/input<i>.txt, where i is the index of the market (not the financial index). Outputs should be to outputs/output<i>.txt for the trace of the i-th market. You can run phase3 as follows:

```
make
```

Approach

Finally, this concludes the problem statement. However, here are some directions regarding how you should go about coding these algorithms

- Sockets: sockets are used to connect between programs and communicate between programs. **You do not need to worry too much about how they work**. The relevant code to handle this communication has been provided to you already. You need to use it correctly.
- Threading: You should read up a little on threading to understand what's happening in your code. Much of the implementation has been done for you, but refer to online resources to understand what multi-threaded programs are, as it will be relevant.
- Most importantly, you should not concern yourself with improving the provided trading strategy. Any doubts regarding the strategy should be clarified. Your goal is to implement an efficient algorithm to execute these strategies. The space complexity should be "reasonable". The time efficiency of algorithms depends on the asymptotic complexity as well as the constants (do not ignore the constants, as they will cause your profit to drop)
- Clear up any doubts you may have as soon as possible.
- **Regarding STL, vector and string are allowed, no questions asked. Other STL imports are not allowed.**
- **You do not have to submit the files marked "DO NOT MODIFY". You can modify them to test your code ONLY. But we will be using our versions of those files for auto-grading.**

P.S.: Depending on your OS, you may have to change the template code slightly to run it on your machine

Submission Instructions

This project is to be done in groups of **at most two students**. Each team should submit one tarball named <rollno1>_<rollno2>.tar.gz on Moodle, where all letters should be in lowercase. On untarring, we should get a folder named <rollno1_rollno2>. Inside, there should be 3 sub-folders, labelled ‘phase1’,’phase2’ and ‘phase3’, each sub-folder containing only the files you have created + files you were supposed to modify within each phase. NO other files can be included. Do NOT include any Makefile in your submission. Refer to the following directory structure:

```
rollno1_rollno2
├── phase1
│       └── <all-modified-code-files>
├── phase2
│       └── <all-modified-code-files>
└── phase3
        └── <all-modified-code-files>
```

Grading

The grading will be automated against a deck of test cases; hence, it is paramount that you match the output specifications. Any doubts regarding this should be clarified on piazza as soon as possible.

Marking Scheme

Marks are allocated for correctness and efficiency. The marks allocation scheme will NOT be revealed beforehand. Please do not approach TAs for any kind of detail.

Honour Code

Plagiarism of ANY sort is not acceptable. All code submitted should be yours and yours alone. Discussion with peers and with TAs is permitted and encouraged. Please feel free to ask for help and use this project as an opportunity to learn. Do NOT share code with one another. However, please make it a point to ask for clarifications where necessary! Submitting the assignment, should you choose to, automatically implies you have read, understood and agreed to the honour code.

All the Best!