ALGO2 (Market Making) Python Algorithm Tutorial

Table of Contents

Introduction

This tutorial is designed for students who are planning to use Python to build a market-making algorithm for ALGO2 using RIT REST API. Students are expected to have followed the RIT REST API User Guide[1] to complete a python algorithm for ALGO1, and also have read the case brief[2] for ALGO2 so that students have a firm understanding of the case. This tutorial is not *required* to complete the algorithm for the case as students can build it without this tutorial, but some students may find it very useful when developing an algorithm for this case that is more 'intelligent' and 'adaptable'.

Basic Setup

Similar to previous tutorials, we will first import the 'requests' package as well as the 'signal' and 'time' packages in order to create some helpful boilerplate code to handle exceptions and CTRL+C commands to stop the algorithm. Then we will save the API KEY for easy access.

```python
# This is a python example algorithm using REST API for the RIT ALGO2 Case
import signal
import requests
from time import sleep
import sys

# this class definition allows us to print error messages and stop the program
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

# set your API key to authenticate to the RIT client
API_KEY = {'X-API-Key': 'XC904YR5'}
shutdown = False
```

We now need to define some simple constants that will act as 'settings' for our algorithm.

```python
#SETTINGS
# how long to wait after submitting buy or sell orders
SPEEDBUMP = 0.5
# maximum number of shares to purchase each order
MAX_VOLUME = 5000
# maximum number of orders we can submit
MAX_ORDERS = 5
# allowed spread before we sell or buy shares
SPREAD = .05
```

In order to have stable execution we need to pause the program after submitting our orders. The 'SPEEDBUMP' constant[3] is how long the program will pause after submitting each set of orders.

[1] "RIT – User Guide – REST API Documentation.pdf"
[2] "RIT – Case Brief – ALGO2 – Algorithmic Market Making.pdf"
[3] We will further discuss and improve this logic in a separate tutorial document, "RIT – Algo Tutorial – Python - Speedbump.pdf"

In order to capture the maximum amount of profit when we submit orders we should be submitting the maximum amount of shares. The 'MAX_VOLUME' constant represents the maximum amount of shares we can purchase each order.

The 'MAX_ORDERS' constant is the maximum number of orders with 'MAX_VOLUME' we can submit without exceeding our position limit. Our position limit in this case is 25,000 and the 'MAX_VOLUME' is 5000. Therefore our 'MAX_ORDERS' in this case is 5.

In order to capture profit between the bid and ask prices we need to set a minimum spread between the bid and ask prices before submitting our orders. The 'SPREAD' constant is the minimum bid ask spread before submitting our orders. Having a set spread equal to .05 insures we are always capturing a 4 cent profit between the bid and ask prices. This is because our bid price is 5 cents lower than our ask price. Then we are losing 1 cents due to a commission fee of 0.5 cent per transaction (in case of ALGO2a for example). Students are suggested to improve their logic on determining the spread according to the case dynamics.

While there are many ways to keep track where we are in an algorithm, we will use the current time (or 'tick') of the simulation case to signal when the algorithm should run. Therefore, we then need a method to get the current case status and return the current time (or 'tick'). So we create a helper method to send a GET request to http://localhost:9999/v1/case.

```
31 # This helper method returns the current 'tick' of the running case.
32 def get_tick(session):
33     resp = session.get('http://localhost:9999/v1/case')
34     if resp.ok:
35         case = resp.json()
36         return case['tick']
37     raise ApiException('Authorization error Please check API key.')
```

We'll now set up the basic setup of a main() method as shown below.

```
39 def main():
40     # creates a session to manage connections and requests to the RIT Client
41     with requests.Session() as s:
42         s.headers.update(API_KEY)
43         tick = get_tick(s)
44 
45         # while the time is between 5 and 295, do the following
46         while tick > 5 and tick < 295 and not shutdown:
47 
48             # refresh the case time. THIS IS IMPORTANT FOR THE WHILE LOOP
49             tick = get_tick(s)
50 
51 if __name__ == '__main__':
52     signal.signal(signal.SIGINT, signal_handler)
53     main()
```

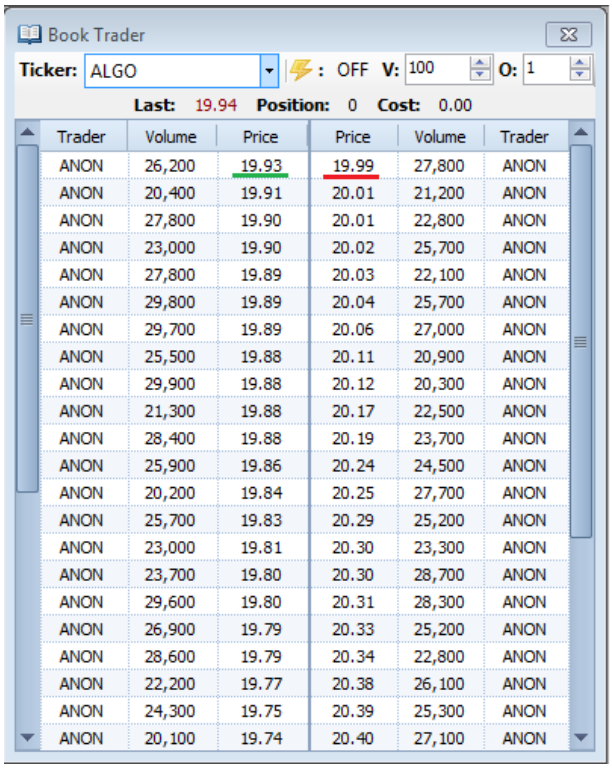
Operationally, when the file is run with python <FILENAME>.py, the get_tick(session) method will be called to return the current time of the case, and while (a) the time is greater than 5 seconds into the case and less than 295 seconds into the case, and (b) the 'shutdown' flag is false, the code in the while-loop will run. As the inline comment notes, it's important to update the tick variable at the end of the loop, so that the algorithm knows whether to continue running the while-loop or not.

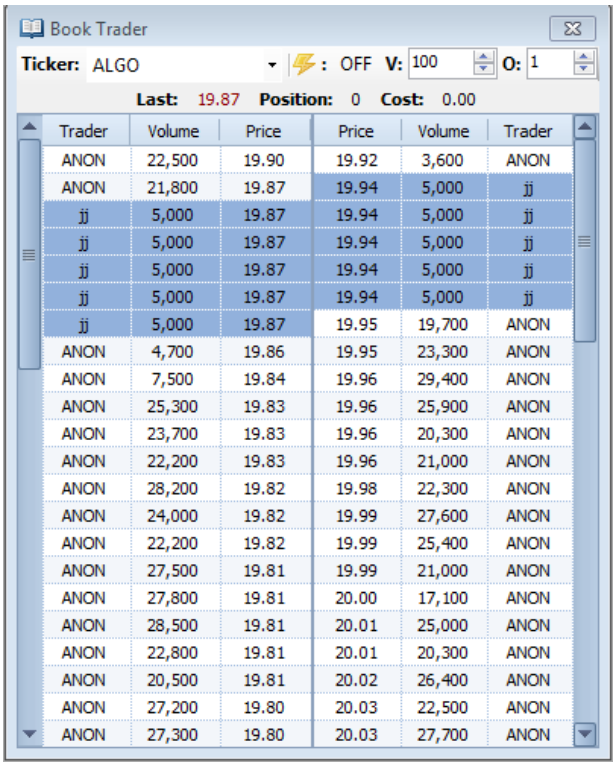
Algorithm Logic

Overview

Now that we have our basic main() method setup, we need to program the trading logic for our algorithm.

Let's start with a simple version of our algorithm that doesn't account for market risk and will just buy and sell shares. The algorithm will buy and sell the maximum amount of shares possible when (a) there is no open orders and (b) the spread between the bid price and the ask price is greater than our equal to the set 'SPREAD' defined above. To better illustrate this look below.





The book trader on the left shows a condition in which we would buy and sell the maximum number of shares. The current bid price is underlined in green. The current ask price is underlined in red. The bid ask spread is the bid price – ask price which is equal to .06. Since (a) .06 is greater than or equal to our set 'SPREAD' defined earlier of .05 and (b) there is no open orders in the book. This would be a condition were we would buy and sell the maximum number of shares.

The book trader on the right shows the result after buying and selling the maximum amount of shares. This is done by submitting the maximum number of orders with the maximum volume each order. This was defined earlier as 'MAX_VOLUME' and 'MAX_ORDERS'. If this is done correctly when one side gets filled completely it will never exceed our position limit. In this case it is true due to the fact our position limit is 25,000. If either side gets filled completely it will equal our position limit.

Helper Methods

In order to capture the bid ask spread. We need a way to get the current bid and ask prices for our security. Let's add a method to get the current bid and ask prices.

```
# This helper method returns the bid and ask first row for a given security.
def ticker_bid_ask(session, ticker):
    payload = {'ticker': ticker}
    resp = session.get('http://localhost:9999/v1/securities/book', params=payload)
    if resp.ok:
        book = resp.json()
        return book['bids'][0]['price'], book['asks'][0]['price']
    raise ApiException('Authorization error Please check API key.')
```

We can get the market book for a security by submitting a `GET` request to http://localhost:9999/v1/securities/book, with a query parameter of ticker equal to the ticker. After checking the response is 'OK', we then parse the response. Finally, we return the price of the first bid and price of the first as a tuple, as they are sorted in order of competitive price.

In order to figure out if there are open orders we need to find the status of the current open orders in the case. Let's add the following two methods that will return information about our open buy and sell orders.

```
# This helper method returns information about all the open sell orders
def open_sells(session):
    resp = session.get('http://localhost:9999/v1/orders?status=OPEN')
    if resp.ok:
        open_sells_volume = 0 # total combined volume of all open sells
        ids = []              # all open sell ids
        prices = []           # all open sell prices
        order_volumes = []    # all open sell volumes
        volume_filled = []    # volume filled for each open sell order

        open_orders = resp.json()
        for order in open_orders:
            if order['action'] == 'SELL':
                volume_filled.append(order['quantity_filled'])
                order_volumes.append(order['quantity'])
                open_sells_volume = open_sells_volume + order['quantity']
                prices.append(order['price'])
                ids.append(order['order_id'])
    return volume_filled, open_sells_volume, ids, prices, order_volumes

# this helper method returns information about all open buy orders
def open_buys(session):
    resp = session.get('http://localhost:9999/v1/orders?status=OPEN')
    if resp.ok:
        open_buys_volume = 0 # total combined volume of all open buys
        ids = []             # all open buy ids
        prices = []          # all open buy prices
        order_volumes = []   # all open buy volumes
        volume_filled = []   # volume filled of each open buy order

        open_orders = resp.json()
        for order in open_orders:
            if order['action'] == 'BUY':
                open_buys_volume = open_buys_volume + order['quantity']
                volume_filled.append(order['quantity_filled'])
                order_volumes.append(order['quantity'])
                prices.append(order['price'])
                ids.append(order['order_id'])

    return volume_filled, open_buys_volume, ids, prices, order_volumes
```

We can get all open orders by sending a `GET` request to http://localhost:9999/v1/orders?status=OPEN. If the response is 'ok' we instantiate the total volume and the open orders attributes we are going to return. Each list position represents one order. For example position 2 of 'ids', 'prices', 'order_volumes', and 'volume_filled' would represent the attributes of one open order.

We then loop through all open orders and check if it is a 'BUY' or a 'SELL' order. If this the case we take that order and append its volume, volume filled, price, and order id to the lists instantiated previously. Then add its volume to the total volume. After we have looped through all open orders we finally return the lists representing each open sell/buy orders attributes and the total volume.

We need a way to buy and sell our orders. Let's add a method that buys and sells the maximum amount of shares.

```
89 # this helper method will buy and sell the maximum number of shares
90 def buy_sell(session, sell_price, buy_price):
91     for i in range(MAX_ORDERS):
92         session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
93         'type': 'LIMIT', 'quantity': MAX_VOLUME, 'price': sell_price, 'action': 'SELL'})
94         session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
95         'type': 'LIMIT', 'quantity': MAX_VOLUME, 'price': buy_price, 'action': 'BUY'})
```

This method takes in 3 parameters the current session, the price we will sell, and the price we buy at. We loop for the maximum number of orders defined earlier as 'MAX_ORDERS'. Each time we submit two `POST` requests to http://localhost:9999/v1/orders. These two requests represent the buy and sell orders. By the end of the method both bid and ask side should contain the maximum number of orders we can submit and the maximum volume for each order.

Implementation

In order to figure out when to submit orders, we need to get information about current bid and ask prices and the state of our current open orders.

```
def main():
    # instantiate variables about all the open buy orders
    buy_ids = []                # order ids
    buy_prices = []             # order prices
    buy_volumes = []            # order volumes
    volume_filled_buys = []     # amount of volume filled for each order
    open_buys_volume = 0        # combined volume from all open buy orders

    # instantiate variables about all the open sell orders
    sell_ids = []
    sell_prices = []
    sell_volumes = []
    volume_filled_sells = []
    open_sells_volume = 0

    # creates a session to manage connections and requests to the RIT Client
    with requests.Session() as s:
        s.headers.update(API_KEY)
        tick = get_tick(s)

        # while the time is between 5 and 295, do the following
        while tick > 5 and tick < 295 and not shutdown:
            # update information about the case
            volume_filled_sells, open_sells_volume, sell_ids, sell_prices, sell_volumes = open_sells(s)
            volume_filled_buys, open_buys_volume, buy_ids, buy_prices, buy_volumes = open_buys(s)
            bid_price, ask_price = ticker_bid_ask(s, 'ALGO')

            #refresh the case time. THIS IS IMPORTANT FOR THE WHILE LOOP
            tick = get_tick(s)
```

In order to keep track of the state of our current open order, we need a set of variables to hold the information about our open orders. We instantiate a set of lists each representing an attribute of our open orders. The position of each list corresponds to the same order. For example, position 1 of the 'sell_ids', 'sell_prices', 'sell_volumes', and 'volume_filled_sells' lists would represent one sell order. Then we instantiate a variable to hold our total open sell orders or open buy orders volume.

At the start of our 'while loop', we call our open_sells(), open_buys() methods and assign the output to the variables we instantiated above. We then call our ticker_bid_ask() method and assign it to our 'bid_price', and 'ask_price' variables. This will insure that, as the case is running, we will be able to keep track of the current open buy and sell orders. As well as the current bid and ask prices.

Let's now set up when to buy and sell shares as well as the prices we will sell and buy them at.

```
        # while the time is between 5 and 295, do the following
        while tick > 5 and tick < 295 and not shutdown:
            # update information about the case
            volume_filled_sells, open_sells_volume, sell_ids, sell_prices, sell_volumes = open_sells(s)
            volume_filled_buys, open_buys_volume, buy_ids, buy_prices, buy_volumes = open_buys(s)
            bid_price, ask_price = ticker_bid_ask(s, 'ALGO')

            # check if you have 0 open orders
            if(open_sells_volume == 0 and open_buys_volume == 0):

                # calculate the spread between the bid and ask prices
                bid_ask_spread = ask_price - bid_price

                # set the prices
                sell_price = ask_price
                buy_price = bid_price

                # the calculated spread is greater or equal to our set spread
                if(bid_ask_spread >= SPREAD):
                    # buy and sell the maximum number of shares
                    buy_sell(s, sell_price, buy_price)
                    sleep(SPEEDBUMP)

            #refresh the case time. THIS IS IMPORTANT FOR THE WHILE LOOP
            tick = get_tick(s)
```

We set the sell price equal to the current ask price and the buy price equal to the current bid price. This insures our order will be the best price when submitted.

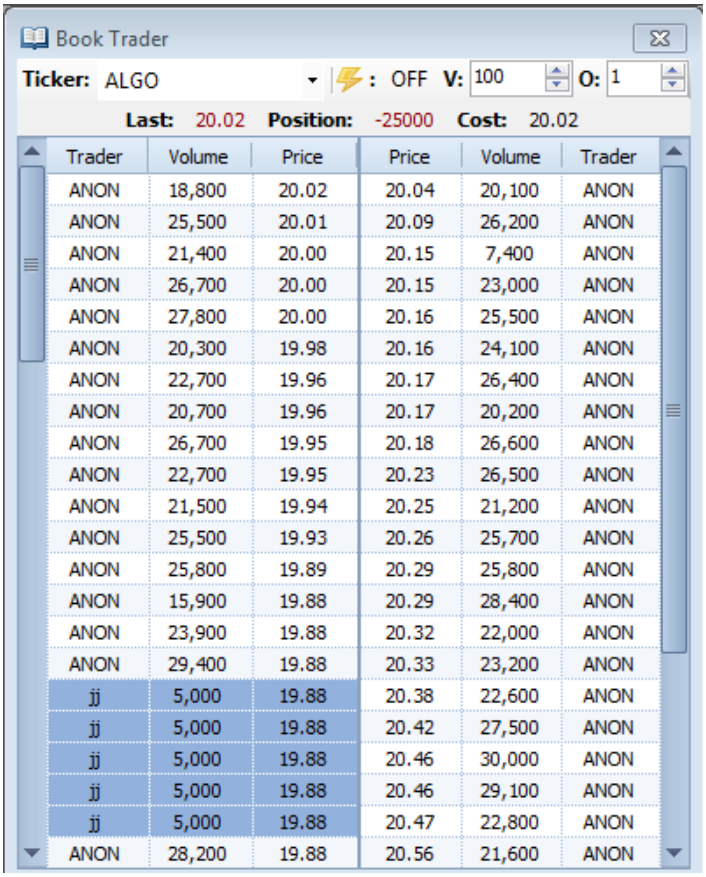
We will buy and sell when (a) there is no open orders and (b) when the bid ask spread is greater than or equal to our set 'SPREAD' defined earlier.

To order our order we call our buy_sell() method defined with our current session, sell price, and buy price as parameters.

Re-Submitting Orders

Overview

We now have a basic working version working of our algorithm. However our algorithm is still subject to significant market risk. To better illustrate this, look below.



Trader	Volume	Price	Price	Volume	Trader
ANON	18,800	20.02	20.04	20,100	ANON
ANON	25,500	20.01	20.09	26,200	ANON
ANON	21,400	20.00	20.15	7,400	ANON
ANON	26,700	20.00	20.15	23,000	ANON
ANON	27,800	20.00	20.16	25,500	ANON
ANON	20,300	19.98	20.16	24,100	ANON
ANON	22,700	19.96	20.17	26,400	ANON
ANON	20,700	19.96	20.17	20,200	ANON
ANON	26,700	19.95	20.18	26,600	ANON
ANON	22,700	19.95	20.23	26,500	ANON
ANON	21,500	19.94	20.25	21,200	ANON
ANON	25,500	19.93	20.26	25,700	ANON
ANON	25,800	19.89	20.29	25,800	ANON
ANON	15,900	19.88	20.29	28,400	ANON
ANON	23,900	19.88	20.32	22,000	ANON
ANON	29,400	19.88	20.33	23,200	ANON
jj	5,000	19.88	20.38	22,600	ANON
jj	5,000	19.88	20.42	27,500	ANON
jj	5,000	19.88	20.46	30,000	ANON
jj	5,000	19.88	20.46	29,100	ANON
jj	5,000	19.88	20.47	22,800	ANON
ANON	28,200	19.88	20.56	21,600	ANON

When orders are submitted, it is possible one side gets filled and one side does not. This is seen in the case above as the ask sides orders have gotten completely filled while the bid side still has all open orders pending. This is not ideal because it results in a positive or negative position exposing us to market risk. The longer this is the case, the longer we are exposed to market risk.

The way to solve this is to cancel our current open orders once one side has been completely filled, and re-submit orders at a more competitive price. This will bring our position back to zero quicker because are open orders will get filled quicker. As a result this will decrease our market risk.

Helper Methods

We need a way to cancel our open orders and re-submit them. Let's create a new method for this logic.

```
# this helper method re-orders all open buys or sells
def re_order(session, number_of_orders, ids, volumes_filled, volumes, price, action):
    for i in range(number_of_orders):
        id = ids[i]
        volume = volumes[i]
        volume_filled = volumes_filled[i]
        # if the order is partially filled.
        if(volume_filled != 0):
            volume = MAX_VOLUME - volume_filled

        # delete then re-purchase.
        deleted = session.delete('http://localhost:9999/v1/orders/{}'.format(id))
        if(deleted.ok):
            session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
            'type': 'LIMIT', 'quantity': volume, 'price': price, 'action': action})

```

The method takes in the current session, how many open orders are in the current case, a set of lists containing the attributes for each order, the price we will sell or buy the new orders at, and an action to communicate whether to re-buy or re-sell. The position of each lists corresponds to an individual order. For example position 2 in the 'ids', 'volumes_filled', and 'volumes' lists corresponds to the attributes of one order.

The method loops through all the open orders. First we delete the order by sending a `DELETE` request to `http://localhost:9999/v1/orders/id` where the 'id' is the id of the open order we are going to delete. If the delete is 'ok' we will re-buy or re-sell the order depending on the 'action' taken in earlier. This is done by sending a `POST` request to `http://localhost:9999/v1/orders` with the query parameters equal to our ticker, the volume of the order we just deleted, the price we want to re-order our order at and the action to take.

Implementation

In order to figure out when to implement this logic, we need to figure out when a single side of the book has been completely filled.

```
    # instantiated variables when just one side of the book has been completely filled
    single_side_filled = False
    single_side_transaction_time = 0

    # creates a session to manage connections and requests to the RIT Client
    with requests.Session() as s:
        s.headers.update(API_KEY)
        tick = get_tick(s)

        # while the time is between 5 and 295, do the following
        while tick > 5 and tick < 295 and not shutdown:
            # update information about the case
            volume_filled_sells, open_sells_volume, sell_ids, sell_prices, sell_volumes = open_sells(s)
            volume_filled_buys, open_buys_volume, buy_ids, buy_prices, buy_volumes = open_buys(s)
            bid_price, ask_price = ticker_bid_ask(s, 'ALGO')

            # check if you have 0 open orders
            if(open_sells_volume == 0 and open_buys_volume == 0):
                # both sides are filled now
                single_side_filled = False

                # calculate the spread between the bid and ask prices
                bid_ask_spread = ask_price - bid_price

                # set the prices
                sell_price = ask_price
                buy_price = bid_price

                # the calculated spread is greater or equal to our set spread
                if(bid_ask_spread >= SPREAD):
                    # buy and sell the maximum number of shares
                    buy_sell(s, sell_price, buy_price)
                    sleep(SPEEDBUMP)

            # there are oustanding open orders
            else:
                # one side of the book has no open orders
                if(not single_side_filled and (open_buys_volume == 0 or open_sells_volume == 0)):
                    single_side_filled = True
                    single_side_transaction_time = tick
```

In order to keep track of a single side has been filled we instantiate two important variables. The ‘single_side_filled’ variable represents if just one side of the book has been completely filled. The ‘single_side_transaction_time’ represents the last time a single side of the book was filled.

If both sides orders have been filled we set the ‘single_side_filled’ variable to false. This is due to the fact that a single side is not filled because both sides have been filled.

We mark when a single side has been filled when a) there are outstanding orders, b) our ‘single_side_filled’ has not been marked as true already and c) the bid or ask side has been completely filled. If these conditions are met we will set ‘single_side_filled’ equal to true. Then set when it was filled by getting the current tick and setting it to our ‘single_side_transaction_time’.

We'll now set up when to re-submit our orders.

```
            # there are oustanding open orders
            else:
                # one side of the book has no open orders
                if(not single_side_filled and (open_buys_volume == 0 or open_sells_volume == 0)):
                    single_side_filled = True
                    single_side_transaction_time = tick

                # ask side has been completely filled
                if(open_sells_volume == 0):
                    # current buy orders are at the top of the book
                    if(buy_price == bid_price):
                        continue # next iteration of loop

                    # its been more than 3 seconds since a single side has been completely filled
                    elif(tick - single_side_transaction_time >= 3):
                        # calulate the potential profits you can make
                        next_buy_price = bid_price + .01
                        potential_profit = sell_price - next_buy_price - .02

                        # potential profit is greater than or equal to a cent or its been more than 6 seconds
                        if(potential_profit >= .01 or tick - single_side_transaction_time >= 6):
                            action = 'BUY'
                            number_of_orders = len(buy_ids)
                            buy_price = bid_price + .01
                            price = buy_price
                            ids = buy_ids
                            volumes = buy_volumes
                            volumes_filled = volume_filled_buys

                            # delete buys and re-buy
                            re_order(s, number_of_orders, ids, volumes_filled, volumes, price, action)
                            sleep(SPEEDBUMP)

                # bid side has been completely filled
                elif(open_buys_volume == 0):
                    # current sell orders are at the top of the book
                    if(sell_price == ask_price):
                        continue # next iteration of loop

                    # its been more than 3 seconds since a single side has been completely filled
                    elif(tick - single_side_transaction_time >= 3):
                        # calculate the potential profit you can make
                        next_sell_price = ask_price - .01
                        potential_profit = next_sell_price - buy_price - .02

                        # potential profit is greater than or equal to a cent or its been more than 6 seconds
                        if(potential_profit >= .01 or tick - single_side_transaction_time >= 6):
                            action = 'SELL'
                            number_of_orders = len(sell_ids)
                            sell_price = ask_price - .01
                            price = sell_price
                            ids = sell_ids
                            volumes = sell_volumes
                            volumes_filled = volume_filled_sells

                            # delete sells then re-sell
                            re_order(s, number_of_orders, ids, volumes_filled, volumes, price, action)
                            sleep(SPEEDBUMP)
```

In order to cancel and re-submit our open orders, we need to figure out when one side is completely filled and which one. Once we figure this out we can figure out to re-buy or re-sell. This is done by checking the volume of each side and if it is equal to 0.

We then check if our current open orders prices are at the top of the book. If this is the case we don't re-order any orders and go to the next iteration of the loop.

If this is not the case we check if it has been 3 seconds or more since one side of the book has gotten filled. This insures that we give enough time for the original orders to be filled.

We will then order under two conditions. We first check if a) the price we will re-sell or re-buy at makes a profit. We do this by looking at the price we will re-order at and the price of our side that got filled at was. To calculate the profit we find the different between the buy and the sell order then subtract 2 cents which would represent the commission fee for both orders. When then check if b) it has been more than 6 seconds since one side of the book was filled.

If one of these two conditions are met we set up the parameters to re-order. We then re-order our open orders by calling the re_order() method.

Now that we implemented this logic, we are going to take a look at the entire code of our algorithm and try running it in the next chapter.

Running the Algorithm

Here's how the complete algorithmic command should look like:

```python
# This is a python example algorithm using REST API for the RIT ALGO2 Case
import signal
import requests
from time import sleep
import sys

# this class definition allows us to print error messages and stop the program
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

# set your API key to authenticate to the RIT client
API_KEY = {'X-API-Key': 'XC904YR5'}
shutdown = False

#SETTINGS
# how long to wait after submitting buy or sell orders
SPEEDBUMP = 0.5
# maximum number of shares to purchase each order
MAX_VOLUME = 5000
# maximum number of orders we can submit
MAX_ORDERS = 5
# allowed spread before we sell or buy shares
SPREAD = .05

# This helper method returns the current 'tick' of the running case.
def get_tick(session):
    resp = session.get('http://localhost:9999/v1/case')
    if resp.ok:
        case = resp.json()
        return case['tick']
    raise ApiException('Authorization error Please check API key.')

# This helper method returns the bid and ask first row for a given security.
def ticker_bid_ask(session, ticker):
    payload = {'ticker': ticker}
    resp = session.get('http://localhost:9999/v1/securities/book', params=payload)
    if resp.ok:
        book = resp.json()
        return book['bids'][0]['price'], book['asks'][0]['price']
    raise ApiException('Authorization error Please check API key.')

```

```python
# This helper method returns information about all the open sell orders
def open_sells(session):
    resp = session.get('http://localhost:9999/v1/orders?status=OPEN')
    if resp.ok:
        open_sells_volume = 0 # total combined volume of all open sells
        ids = []              # all open sell ids
        prices = []           # all open sell prices
        order_volumes = []    # all open sell volumes
        volume_filled = []    # volume filled for each open sell order

        open_orders = resp.json()
        for order in open_orders:
            if order['action'] == 'SELL':
                volume_filled.append(order['quantity_filled'])
                order_volumes.append(order['quantity'])
                open_sells_volume = open_sells_volume + order['quantity']
                prices.append(order['price'])
                ids.append(order['order_id'])
    return volume_filled, open_sells_volume, ids, prices, order_volumes

# this helper method returns information about all open buy orders
def open_buys(session):
    resp = session.get('http://localhost:9999/v1/orders?status=OPEN')
    if resp.ok:
        open_buys_volume = 0 # total combined volume of all open buys
        ids = []             # all open buy ids
        prices = []          # all open buy prices
        order_volumes = []   # all open buy volumes
        volume_filled = []   # volume filled of each open buy order

        open_orders = resp.json()
        for order in open_orders:
            if order['action'] == 'BUY':
                open_buys_volume = open_buys_volume + order['quantity']
                volume_filled.append(order['quantity_filled'])
                order_volumes.append(order['quantity'])
                prices.append(order['price'])
                ids.append(order['order_id'])

    return volume_filled, open_buys_volume, ids, prices, order_volumes

# this helper method will buy and sell the maximum number of shares
def buy_sell(session, sell_price, buy_price):
    for i in range(MAX_ORDERS):
        session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
        'type': 'LIMIT', 'quantity': MAX_VOLUME, 'price': sell_price, 'action': 'SELL'})
        session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
```

```
95          'type': 'LIMIT', 'quantity': MAX_VOLUME, 'price': buy_price, 'action': 'BUY'})
96
97  # this helper method re-orders all open buys or sells
98  def re_order(session, number_of_orders, ids, volumes_filled, volumes, price, action):
99      for i in range(number_of_orders):
100         id = ids[i]
101         volume = volumes[i]
102         volume_filled = volumes_filled[i]
103         # if the order is partially filled.
104         if(volume_filled != 0):
105             volume = MAX_VOLUME - volume_filled
106
107         # delete then re-purchase.
108         deleted = session.delete('http://localhost:9999/v1/orders/{}'.format(id))
109         if(deleted.ok):
110             session.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
111             'type': 'LIMIT', 'quantity': volume, 'price': price, 'action': action})
112
113 def main():
114     # instantiate variables about all the open buy orders
115     buy_ids = []                # order ids
116     buy_prices = []             # order prices
117     buy_volumes = []            # order volumes
118     volume_filled_buys = []     # amount of volume filled for each order
119     open_buys_volume = 0        # combined volume from all open buy orders
120
121     # instantiate variables about all the open sell orders
122     sell_ids = []
123     sell_prices = []
124     sell_volumes = []
125     volume_filled_sells = []
126     open_sells_volume = 0
127
128     # instantiated variables when just one side of the book has been completely filled
129     single_side_filled = False
130     single_side_transaction_time = 0
131
132     # creates a session to manage connections and requests to the RIT Client
133     with requests.Session() as s:
134         s.headers.update(API_KEY)
135         tick = get_tick(s)
136
137         # while the time is between 5 and 295, do the following
138         while tick > 5 and tick < 295 and not shutdown:
139             # update information about the case
140             volume_filled_sells, open_sells_volume, sell_ids, sell_prices, sell_volumes = open_sells(s)
141             volume_filled_buys, open_buys_volume, buy_ids, buy_prices, buy_volumes = open_buys(s)
142             bid_price, ask_price = ticker_bid_ask(s, 'ALGO')
```

```
143
144             # check if you have 0 open orders
145             if(open_sells_volume == 0 and open_buys_volume == 0):
146                 # both sides are filled now
147                 single_side_filled = False
148
149                 # calculate the spread between the bid and ask prices
150                 bid_ask_spread = ask_price - bid_price
151
152                 # set the prices
153                 sell_price = ask_price
154                 buy_price = bid_price
155
156                 # the calculated spread is greater or equal to our set spread
157                 if(bid_ask_spread >= SPREAD):
158                     # buy and sell the maximum number of shares
159                     buy_sell(s, sell_price, buy_price)
160                     sleep(SPEEDBUMP)
161
162             # there are oustanding open orders
163             else:
164                 # one side of the book has no open orders
165                 if(not single_side_filled and (open_buys_volume == 0 or open_sells_volume == 0)):
166                     single_side_filled = True
167                     single_side_transaction_time = tick
168
169                 # ask side has been completely filled
170                 if(open_sells_volume == 0):
171                     # current buy orders are at the top of the book
172                     if(buy_price == bid_price):
173                         continue # next iteration of loop
174
175                     # its been more than 3 seconds since a single side has been completely filled
176                     elif(tick - single_side_transaction_time >= 3):
177                         # calulate the potential profits you can make
178                         next_buy_price = bid_price + .01
179                         potential_profit = sell_price - next_buy_price - .02
180
181                         # potential profit is greater than or equal to a cent or its been more than 6 seconds
182                         if(potential_profit >= .01 or tick - single_side_transaction_time >= 6):
183                             action = 'BUY'
184                             number_of_orders = len(buy_ids)
185                             buy_price = bid_price + .01
186                             price = buy_price
187                             ids = buy_ids
188                             volumes = buy_volumes
189                             volumes_filled = volume_filled_buys
190
191                             # delete buys and re-buy
192                             re_order(s, number_of_orders, ids, volumes_filled, volumes, price, action)
193                             sleep(SPEEDBUMP)
```

```
194
195                 # bid side has been completely filled
196                 elif(open_buys_volume == 0):
197                     # current sell orders are at the top of the book
198                     if(sell_price == ask_price):
199                         continue # next iteration of loop
200
201                     # its been more than 3 seconds since a single side has been completely filled
202                     elif(tick - single_side_transaction_time >= 3):
203                         # calculate the potential profit you can make
204                         next_sell_price = ask_price - .01
205                         potential_profit = next_sell_price - buy_price - .02
206
207                         # potential profit is greater than or equal to a cent or its been more than 6 seconds
208                         if(potential_profit >= .01 or tick - single_side_transaction_time >= 6):
209                             action = 'SELL'
210                             number_of_orders = len(sell_ids)
211                             sell_price = ask_price - .01
212                             price = sell_price
213                             ids = sell_ids
214                             volumes = sell_volumes
215                             volumes_filled = volume_filled_sells
216
217                             # delete sells then re-sell
218                             re_order(s, number_of_orders, ids, volumes_filled, volumes, price, action)
219                             sleep(SPEEDBUMP)
220
221             #refresh the case time. THIS IS IMPORTANT FOR THE WHILE LOOP
222             tick = get_tick(s)
223
224 if __name__ == '__main__':
225     signal.signal(signal.SIGINT, signal_handler)
226     main()
```

In order to run the algorithm, ensure that the RIT client is connected and the REST API is enabled. Then, from the working directory, enter `python <FILENAME>.py` into the prompt. To stop the algorithm before the case is finished, press `CTRL+C`. If the file name has any space in it, please enter `python "<FILENAME>.py"`

Note: if students make changes to the algorithm's code while it is running in the prompt, those changes will not be reflected in what is running. Students will have to stop and restart the algorithm.