Technical Report: Final Project
EECE 2560: Fundamentals of Engineering Algorithms

Noah Assam, Chance Bowman
Department of Electrical and Computer Engineering
Northeastern University
assam.n@northeastern.edu, bowman.c@northeastern.edu

December 4, 2024

Contents

1 Project Scope

This project aims to develop a stock trading simulator that leverages greedy algorithms and historical data analytics to enable users to make profitable trading decisions, in real-time. The system will address the challenge of making trading decisions in a volatile stock market. It provides data-driven trade suggestions, as well as a user-friendly interface, for easier interaction with stock market trends.

The project's main objectives are:

- To develop a user interface for real-time stock trading.
- To implement, and refine, greedy algorithms that optimize trading decisions.
- To use historical stock data to verify algorithmic accuracy and refine strategies.

The expected outcomes include a functioning web-based system, proper documentation, and a final project report.

2 Project Plan

2.1 Timeline

The overall timeline for the project is divided into phases:

- **Week 1 (October 7 - October 13)**: Define project scope, establish team roles, and outline skills/tools.
- **Week 2 (October 14 - October 20)**: Research and design system architecture.
- **Week 3 (October 21 - October 27)**: Begin UI development, greedy algorithm prototyping, and some initial tests with historical data.
- **Week 4 (October 28 - November 3)**: Further refine greedy algorithm, implement a more advanced trade logic, and begin backend integration.
- **Week 5 (November 4 - November 10)**: Develop graph algorithms for trend analysis, if believed to be necessary addition. Complete database integration.
- **Week 6 (November 11 - November 17)**: Testing and debugging.
- **Week 7 (November 18 - November 28)**: Final presentation, report submission, and project closure.

2.2 Milestones

Key milestones include:

- Completion of architecture design (End of Week 2).
- User interface prototype (End of Week 3).
- Greedy algorithms competition (End of Week 5).
- Graph algorithms completion, if chosen to pursue (End of Week 7).
- Final testing and debugging (End of Week 8).

3 Team Roles

- **Team Member 1:** Responsible for user interface and database development.
- **Team Member 2:** Responsible for backend development and documentation.

These roles are flexible and may change as the project progresses.

4 Methodology

4.1 Pseudocode and Complexity Analysis

The primary algorithms, in a pseudocode-form, are provided below. The overall time complexity of the program is O(n), such as to allow quick and efficient engagements with the simulator.

Trade Signal Generation:

```
FUNCTION generate_trade_signal(symbol, current_price, trend, volatility):
    IF volatility > (current_price * 0.05):     //Risk management threshold
        RETURN 'hold'

IF trend == 'uptrend' AND symbol NOT IN current_holdings:
    IF balance >= current_price:
        RETURN 'buy'

IF trend == 'downtrend' AND symbol IN current_holdings:
    RETURN 'sell'

RETURN 'hold'
```

The time complexity for the above function is O(1). It performs constant-time checks for current market conditions. The space complexity is O(1).

Technical Indicator Calculations:

```
FUNCTION calculate_technical_indicators(price_data):
    Calculate short-term moving average:
        Use 20-day rolling window on closing prices

    Calculate long-term moving average:
        Use 50-day rolling window on closing prices

    Calculate volatility:
        Use 20-day rolling standard deviation of closing prices

    RETURN (short_term_ma, long_term_ma, volatility)
```

The time complexity for the above function is O(n), as it performs rolling operations on the price column DataFrame. In this case, n is the number of rows in the price data DataFrame. The space complexity, similarly, is O(n). There are three new series created, all of which have the same size as the input DataFrame. Again, n is the number of rows in the DataFrame.

Execute Trades:

```
FUNCTION execute_trade(symbol, action, price, quantity):
    CREATE timestamp of current moment

    IF action == 'buy':
        CALCULATE total_cost = price * quantity
        IF total_cost <= current_balance:
            Subtract total_cost from balance
            Add quantity to holdings for symbol
            Record trade in trade history
            RETURN true

    IF action == 'sell':
        IF symbol exists in holdings AND quantity <= current holdings:
            CALCULATE total_value = price * quantity
            Add total_value to balance
            Subtract quantity from holdings for symbol
            IF holdings for symbol become zero:
                Remove symbol from holdings
            Record trade in trade history
            RETURN true
```

```
    RETURN false
```

The time complexity of the function above is O(1), as the function only performs conditional checks, updating the internal state, as necessary. The space complexity, on the other hand, is O(t), where t is the number of trades executed.

Fetching Stock Data:

```
FUNCTION fetch_stock_data(symbol, duration):
    TRY:
        Retrieve historical stock data for given symbol
        Use Yahoo Finance API with specified duration
        RETURN historical price data as DataFrame
    CATCH any errors:
        Raise error with descriptive message
```

The time complexity of the above function, theoretically, is O(1), as it doesn't involve any iterative processing, but is instead I/O bound. The space complexity is rather dependent on the size of the data returned. As such, the it was defined as O(n), where n is the number of rows of stock data retrieved.

4.2 Data Collection and Preprocessing

The data was collected, simply, by an open-source Yahoo Finance API. Such allowed for real-time updates to the ever-fluctuating stock market. It was decided over a standard, SQL database of historical data for the reasons of efficiency and accuracy. It involves much fewer operations to preprocess the algorithm. A timestamp is created for the moment of simulation run. That timestamp, in collection with a given symbol, is searched within the API database, and retrieved.

5 Results

The simulator makes relatively smart decisions, based on volatility and moving average calculations. The below simulations were run on 11/21/2024, at roughly 14:00:00. The algorithms were, subjectively, effective in making trade decisions.

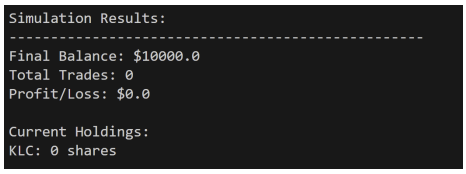


Figure 3: A simulation run on the KLC symbol. The company KinderCare Learning Company is in a defined 'downtrend', and as such is not bought. This was run on 11/21/2024 at 14:05:00.

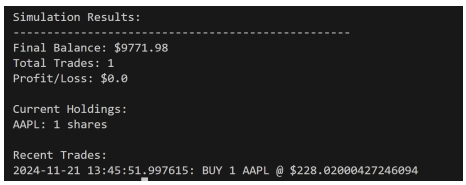


Figure 1: A simulation run on the AAPL symbol. The company Apple is in a defined 'uptrend', and as such is bought. This was run on 11/21/2024 at 13:45:00.

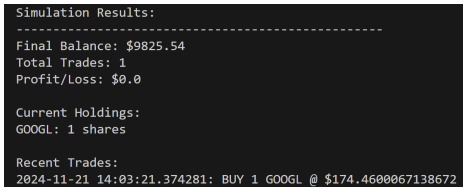


Figure 2: A simulation run on the GOOGL symbol. The company Google is in a defined 'uptrend', and as such is bought. This was run on 11/21/2024 at 14:03:00.

6 Discussion

The simulator was a relative success, with a modular structure, open to further expansion. The results imply that data-driven trading strategies can simply the stock market, making it more accessible to non-financial experts. An additional finding is that this simulator may prove to serve as an educational platform for understanding market trends, again making the stock market more accessible. The final objectives are similar to the initial objectives, though due to time constraints, the final product was not fully actualized as had been planned. There is a lack detailed performance metrics between encounters, which limits a complete assessment of the simulator's efficacy. It was found, and though possibly

subjective, that an open source API, Yahoo Finance, was more efficient for data collection than the standard SQL database. Additionally, the greedy algorithm may 'get the job done', so-to-say, but it's simplicity overlooks more complex market conditions, where dynamic programming or reinforcement learning may prove more efficient.

7 Conclusion

This section is, essentially, a restatement of the discussion (based on the interpretation of provided requirements). It was discovered that when paired with basic technical indicators (i.e., moving averages), greedy algorithms can facilitate effective trading decisions. Though, per most cases, it can always be better. A key goal was to provide greater accessibility to the stock market. As such, the development of the simulator does, indeed, provide greater accessibility. It demonstrates how trading tools can be made more accessible, bridging the gap between complex financial systems and general consumers. For future research, the following came to mind: the use of more enhanced algorithms, as a way to implement more sophisticated strategies; using more enhanced risk management (i.e., RSI, ADX, or Bollinger Bands); and, simply, creating a better interface.

8 References

"Bloomberg Terminal — Bloomberg Professional Services." *Bloomberg.Com*, Bloomberg, www.bloomberg.com/professional/products/bloomberg-terminal/#terminal-in-action. Accessed 4 Dec. 2024.

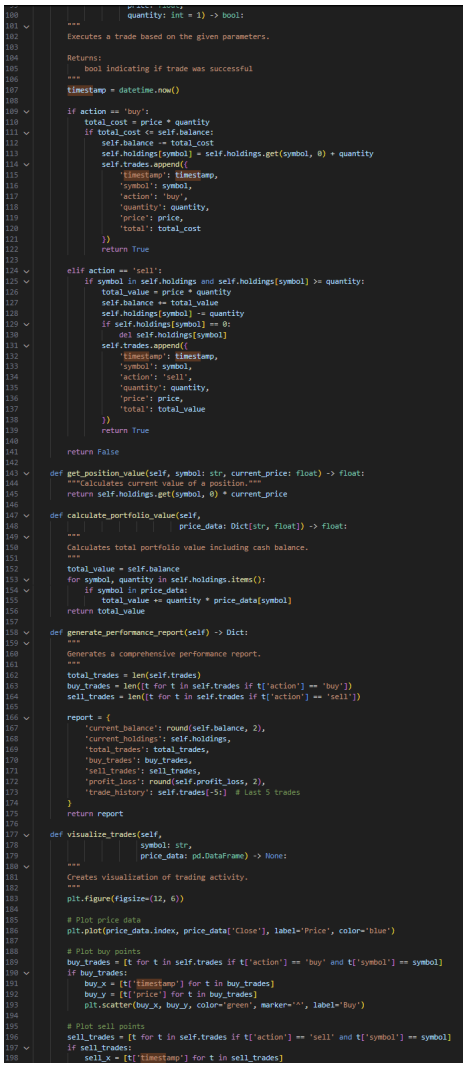
Seth, Shobhit. "Basics of Algorithmic Trading: Concepts and Examples." *Investopedia*, Investopedia, www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp. Accessed 4 Dec. 2024.

"Ultimate Guide to Algorithmic Trading Strategies." *TradingCanyon*, 6 Nov. 2023, www.tradingcanyon.com/trading/algorithmic-trading-strategies/.

A Appendix A: Code

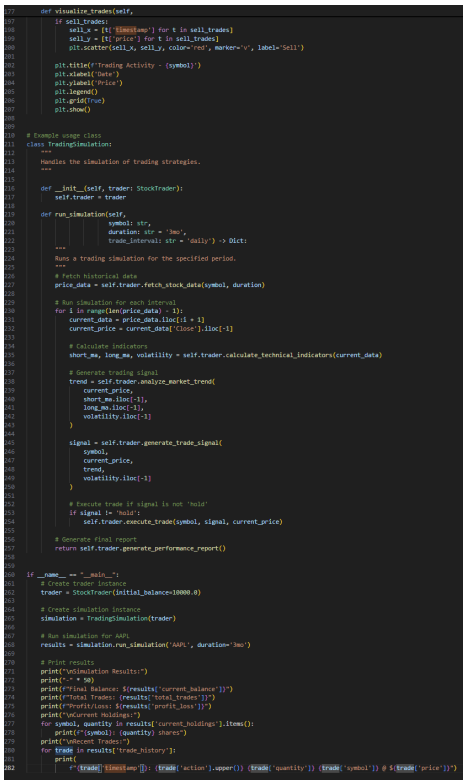
```python
import yfinance as yf
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
from typing import Dict, List, Optional, Tuple


class StockTrader:
    """
    A stock trading system that implements greedy algorithms and technical analysis
    for making trading decisions.
    """

    def __init__(self, initial_balance: float = 10000.0):
        self.balance = initial_balance
        self.holdings: Dict[str, int] = {}  # {stock_symbol: quantity}
        self.trades: List[dict] = []
        self.profit_loss = 0.0

    def fetch_stock_data(self, symbol: str, duration: str = '1mo') -> pd.DataFrame:
        """
        Fetches historical stock data from Yahoo Finance.

        Args:
            symbol: Stock ticker symbol
            duration: Time period for data (e.g., '1mo', '3mo', '1y')

        Returns:
            DataFrame with stock price history
        """
        try:
            stock = yf.Ticker(symbol)
            data = stock.history(period=duration)
            return data
        except Exception as e:
            raise ValueError(f"Failed to fetch data for {symbol}: {str(e)}")

    def calculate_technical_indicators(self,
                                       price_data: pd.DataFrame) -> Tuple[pd.Series, pd.Series, pd.Series]:
        """
        Calculates technical indicators for trading decisions.

        Args:
            price_data: DataFrame with price history

        Returns:
            Tuple of (short_term_ma, long_term_ma, volatility)
        """
        short_term_ma = price_data['Close'].rolling(window=20).mean()
        long_term_ma = price_data['Close'].rolling(window=50).mean()
        volatility = price_data['Close'].rolling(window=20).std()

        return short_term_ma, long_term_ma, volatility

    def analyze_market_trend(self,
                             current_price: float,
                             short_ma: float,
                             long_ma: float,
                             volatility: float) -> str:
        """
        Analyzes market trend using technical indicators.

        Returns:
            'uptrend', 'downtrend', or 'neutral'
        """
        if short_ma > long_ma and current_price > short_ma:
            return 'uptrend'
        elif short_ma < long_ma and current_price < short_ma:
            return 'downtrend'
        return 'neutral'

    def generate_trade_signal(self,
                              symbol: str,
                              current_price: float,
                              trend: str,
                              volatility: float) -> str:
        """
        Generates trading signal based on market analysis.

        Returns:
            'buy', 'sell', or 'hold'
        """
        # Risk management: Don't trade if volatility is too high
        if volatility > current_price * 0.05:  # 5% volatility threshold
            return 'hold'

        if trend == 'uptrend' and symbol not in self.holdings:
            if self.balance >= current_price:
                return 'buy'
        elif trend == 'downtrend' and symbol in self.holdings:
            return 'sell'

        return 'hold'

    def execute_trade(self,
                      symbol: str,
                      action: str,
                      price: float,
                      quantity: int = 1) -> bool:
        """
```

Figure 4: Code (Part I/III)



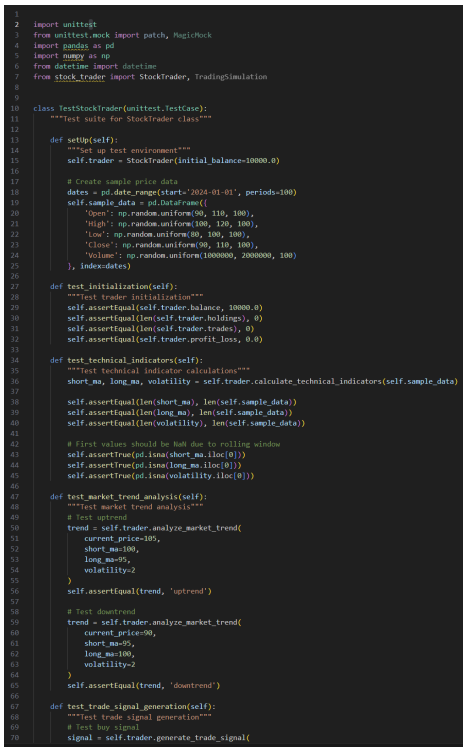
```
                    quantity: int = 1) -> bool:
        """
        Executes a trade based on the given parameters.

        Returns:
            bool indicating if trade was successful
        """
        timestamp = datetime.now()

        if action == 'buy':
            total_cost = price * quantity
            if total_cost <= self.balance:
                self.balance -= total_cost
                self.holdings[symbol] = self.holdings.get(symbol, 0) + quantity
                self.trades.append({
                    'timestamp': timestamp,
                    'symbol': symbol,
                    'action': 'buy',
                    'quantity': quantity,
                    'price': price,
                    'total': total_cost
                })
                return True

        elif action == 'sell':
            if symbol in self.holdings and self.holdings[symbol] >= quantity:
                total_value = price * quantity
                self.balance += total_value
                self.holdings[symbol] -= quantity
                if self.holdings[symbol] == 0:
                    del self.holdings[symbol]
                self.trades.append({
                    'timestamp': timestamp,
                    'symbol': symbol,
                    'action': 'sell',
                    'quantity': quantity,
                    'price': price,
                    'total': total_value
                })
                return True

        return False

    def get_position_value(self, symbol: str, current_price: float) -> float:
        """Calculates current value of a position."""
        return self.holdings.get(symbol, 0) * current_price

    def calculate_portfolio_value(self,
                                  price_data: Dict[str, float]) -> float:
        """
        Calculates total portfolio value including cash balance.
        """
        total_value = self.balance
        for symbol, quantity in self.holdings.items():
            if symbol in price_data:
                total_value += quantity * price_data[symbol]
        return total_value

    def generate_performance_report(self) -> Dict:
        """
        Generates a comprehensive performance report.
        """
        total_trades = len(self.trades)
        buy_trades = len([t for t in self.trades if t['action'] == 'buy'])
        sell_trades = len([t for t in self.trades if t['action'] == 'sell'])

        report = {
            'current_balance': round(self.balance, 2),
            'current_holdings': self.holdings,
            'total_trades': total_trades,
            'buy_trades': buy_trades,
            'sell_trades': sell_trades,
            'profit_loss': round(self.profit_loss, 2),
            'trade_history': self.trades[-5:]  # Last 5 trades
        }
        return report

    def visualize_trades(self,
                         symbol: str,
                         price_data: pd.DataFrame) -> None:
        """
        Creates visualization of trading activity.
        """
        plt.figure(figsize=(12, 6))

        # Plot price data
        plt.plot(price_data.index, price_data['Close'], label='Price', color='blue')

        # Plot buy points
        buy_trades = [t for t in self.trades if t['action'] == 'buy' and t['symbol'] == symbol]
        if buy_trades:
            buy_x = [t['timestamp'] for t in buy_trades]
            buy_y = [t['price'] for t in buy_trades]
            plt.scatter(buy_x, buy_y, color='green', marker='^', label='Buy')

        # Plot sell points
        sell_trades = [t for t in self.trades if t['action'] == 'sell' and t['symbol'] == symbol]
        if sell_trades:
            sell_x = [t['timestamp'] for t in sell_trades]
```

Figure 5: Code (Part II/III)



```
    def visualize_trades(self,
        if sell_trades:
            sell_x = [t['timestamp'] for t in sell_trades]
            sell_y = [t['price'] for t in sell_trades]
            plt.scatter(sell_x, sell_y, color='red', marker='v', label='Sell')

        plt.title(f'Trading Activity - {symbol}')
        plt.xlabel('Date')
        plt.ylabel('Price')
        plt.legend()
        plt.grid(True)
        plt.show()


# Example usage class
class TradingSimulation:
    """
    Handles the simulation of trading strategies.
    """

    def __init__(self, trader: StockTrader):
        self.trader = trader

    def run_simulation(self,
                       symbol: str,
                       duration: str = '1mo',
                       trade_interval: str = 'daily') -> Dict:
        """
        Runs a trading simulation for the specified period.
        """
        # Fetch historical data
        price_data = self.trader.fetch_stock_data(symbol, duration)

        # Run simulation for each interval
        for i in range(len(price_data) - 1):
            current_data = price_data.iloc[:i + 1]
            current_price = current_data['Close'].iloc[-1]

            # Calculate indicators
            short_ma, long_ma, volatility = self.trader.calculate_technical_indicators(current_data)

            # Generate trading signal
            trend = self.trader.analyze_market_trend(
                current_price,
                short_ma.iloc[-1],
                long_ma.iloc[-1],
                volatility.iloc[-1]
            )

            signal = self.trader.generate_trade_signal(
                symbol,
                current_price,
                trend,
                volatility.iloc[-1]
            )

            # Execute trade if signal is not 'hold'
            if signal != 'hold':
                self.trader.execute_trade(symbol, signal, current_price)

        # Generate final report
        return self.trader.generate_performance_report()


if __name__ == "__main__":
    # Create trader instance
    trader = StockTrader(initial_balance=10000.0)

    # Create simulation instance
    simulation = TradingSimulation(trader)

    # Run simulation for AAPL
    results = simulation.run_simulation('AAPL', duration='3mo')

    # Print results
    print("\n=Simulation Results:")
    print("-" * 50)
    print(f"Final Balance: ${results['current_balance']}")
    print(f"Total Trades: {results['total_trades']}")
    print(f"Profit/Loss: ${results['profit_loss']}")
    print("\nCurrent Holdings:")
    for symbol, quantity in results['current_holdings'].items():
        print(f"{symbol}: {quantity} shares")
    print("\nRecent Trades:")
    for trade in results['trade_history']:
        print(
            f"{trade['timestamp']}: {trade['action'].upper()} {trade['quantity']} {trade['symbol']} @ ${trade['price']}")
```

Figure 6: Code (Part III/III)



```python

import unittest
from unittest.mock import patch, MagicMock
import pandas as pd
import numpy as np
from datetime import datetime
from stock_trader import StockTrader, TradingSimulation


class TestStockTrader(unittest.TestCase):
    """Test suite for StockTrader class"""

    def setUp(self):
        """Set up test environment"""
        self.trader = StockTrader(initial_balance=10000.0)

        # Create sample price data
        dates = pd.date_range(start='2024-01-01', periods=100)
        self.sample_data = pd.DataFrame({
            'Open': np.random.uniform(90, 110, 100),
            'High': np.random.uniform(100, 120, 100),
            'Low': np.random.uniform(80, 100, 100),
            'Close': np.random.uniform(90, 110, 100),
            'Volume': np.random.uniform(1000000, 2000000, 100)
        }, index=dates)

    def test_initialization(self):
        """Test trader initialization"""
        self.assertEqual(self.trader.balance, 10000.0)
        self.assertEqual(len(self.trader.holdings), 0)
        self.assertEqual(len(self.trader.trades), 0)
        self.assertEqual(self.trader.profit_loss, 0.0)

    def test_technical_indicators(self):
        """Test technical indicator calculations"""
        short_ma, long_ma, volatility = self.trader.calculate_technical_indicators(self.sample_data)

        self.assertEqual(len(short_ma), len(self.sample_data))
        self.assertEqual(len(long_ma), len(self.sample_data))
        self.assertEqual(len(volatility), len(self.sample_data))

        # First values should be NaN due to rolling window
        self.assertTrue(pd.isna(short_ma.iloc[0]))
        self.assertTrue(pd.isna(long_ma.iloc[0]))
        self.assertTrue(pd.isna(volatility.iloc[0]))

    def test_market_trend_analysis(self):
        """Test market trend analysis"""
        # Test uptrend
        trend = self.trader.analyze_market_trend(
            current_price=105,
            short_ma=100,
            long_ma=95,
            volatility=2
        )
        self.assertEqual(trend, 'uptrend')

        # Test downtrend
        trend = self.trader.analyze_market_trend(
            current_price=90,
            short_ma=95,
            long_ma=100,
            volatility=2
        )
        self.assertEqual(trend, 'downtrend')

    def test_trade_signal_generation(self):
        """Test trade signal generation"""
        # Test buy signal
        signal = self.trader.generate_trade_signal(
```

Figure 7: Testing Of Code (Part I/II)

```
        signal = self.trader.generate_trade_signal(
            symbol='AAPL',
            current_price=100,
            trend='uptrend',
            volatility=2
        )
        self.assertEqual(signal, 'buy')

        # Test high volatility hold
        signal = self.trader.generate_trade_signal(
            symbol='AAPL',
            current_price=100,
            trend='uptrend',
            volatility=10
        )
        self.assertEqual(signal, 'hold')

    def test_trade_execution(self):
        """Test trade execution"""
        # Test successful buy
        success = self.trader.execute_trade('AAPL', 'buy', 100, 1)
        self.assertTrue(success)
        self.assertEqual(self.trader.balance, 9900.0)
        self.assertEqual(self.trader.holdings['AAPL'], 1)

        # Test successful sell
        success = self.trader.execute_trade('AAPL', 'sell', 110, 1)
        self.assertTrue(success)
        self.assertEqual(self.trader.balance, 10010.0)
        self.assertEqual(len(self.trader.holdings), 0)

    def test_invalid_trades(self):
        """Test invalid trade scenarios"""
        # Test insufficient balance
        success = self.trader.execute_trade('AAPL', 'buy', 20000, 1)
        self.assertFalse(success)

        # Test selling without holdings
        success = self.trader.execute_trade('AAPL', 'sell', 100, 1)
        self.assertFalse(success)

    def test_portfolio_calculations(self):
        """Test portfolio value calculations"""
        self.trader.execute_trade('AAPL', 'buy', 100, 1)
        position_value = self.trader.get_position_value('AAPL', 110)
        self.assertEqual(position_value, 110)

        portfolio_value = self.trader.calculate_portfolio_value({'AAPL': 110})
        self.assertEqual(portfolio_value, 9900 + 110)

    def test_performance_report(self):
        """Test performance report generation"""
        self.trader.execute_trade('AAPL', 'buy', 100, 1)
        self.trader.execute_trade('AAPL', 'sell', 110, 1)

        report = self.trader.generate_performance_report()

        self.assertIn('current_balance', report)
        self.assertIn('current_holdings', report)
        self.assertIn('total_',report)
```

Figure 8: Testing Of Code (Part II/II)

B Appendix B: Iteration Updates

B.0.1 Iteration 2 (October 10)

The team successfully began the initial setup of the development environment and GitHub repository. Research on backend development has begun. The team has encountered issues with the plan for constructing the greedy algorithms. Will resolve by reviewing class resources and utilizing tutorials.

B.0.2 Iteration 3 + 4 (October 27)

Greedy Algorithm Development

Following the team's initial challenges, we implemented a foundational version of the greedy algorithm. It focuses on optimizing buying and selling actions, based on trend indicators. Such an iterative approach has allowed for the start of a basic setup, enabling a gradual improvement of the algorithm by:

- Introducing technical indicators, such as short- and long-term moving averages.
- Integrating a volatility check such as to minimize risk.
- Testing with historical stock data to evaluate decision-making accuracy.

Next steps involve the refinement of this algorithm, as well as testing against edge cases to ensure proper performance.

Evidence of code execution may be viewed in the Appendix, as Fig. 1, Fig. 2, Fig. 3. Evidence of output may be viewed as Fig. 4.

User Interface Development

The team has developed a relatively intuitive interface for the simulator, enabling users to interact with real-time stock trends and make guided trading decisions. Currently, the interface includes:

- Real-time visualization of stock trends.
- Simple trade actions (buy, sell, hold), which are based on algorithmic suggestions.
- A dashboard displaying portfolio balance, stock holdings, and a trade history.

Future iterations will enhance usability, possibly adding features for a deeper interaction with market data.

Backend and Database Integration The backend development is underway, though more central efforts will be completed in the upcoming weeks.

Algorithm Testing

Early testing has involved simulated datasets to verify some basic functionalities. Such has included:

- The testing of market indicators and market trend analysis.
- The testing of trade signal generation and execution.
- The testing of invalid trades, (e.g., selling unowned holdings, buying holdings without sufficient stock).

Evidence of testing may be viewed in the Appendix, as Fig. 5, Fig. 6.

System Architecture
The system architecture follows a layered approach:

- *User Interface:* This layer allows users to interact with the simulator. It includes a graphical interface displaying stock trends, trade options, and user account details.
- *Backend Services:* This layer is responsible for processing user requests, managing trade logic, and handling data communications. It includes:
  - Greedy Algorithm Module: This component handles trading decisions, based on implemented greedy algorithms. It continuously analyzes market data in order to optimize trade decisions.
  - Data Processing Module (TODO): This component manages the retrieval and analysis of historical stock data, utilizing SQL for database interactions.
- *Database Layer (TODO):* This layer uses SQL to store user data, historical stock information, as well as trade history.

Data Representation:
The system utilizes various data structures for data management:

- *Stock Data Representation:* Each stock is represented as an object, containing attributes such as stock symbol, current price, historical prices, and trading volume.
- *User Portfolio:* User portfolios are stored as a collection of stock object, allowing for access to both stock holdings and performance metrics.
- *Trade History:* All trade actions are recorded in a format that includes timestamps, actions, and quantities.

New Challenges:
As of this week, there have not yet been any new challenges encountered. Such is most probably due to the relative generality of the current algorithm and lack of a backend. Ergo, no new solutions are proposed. This will be kept updated for future iterations.

B.0.3 Iteration 5 (November 10)

Plans Made:

Team Member 1:

- Unit Testing
  - Implement edge case scenarios and stress testing

  - Create error handling
  - Design automated validation protocols for consistency
- New Analysis Algorithm
  - Develop implementation
  - Create integration with existing systems
- Initial Presentation
  - Create system diagrams and performance metrics
  - Outline implementation progress and milestones
  - Draft initial findings and technical achievements

Team Member 2:

- Storage Improvements
  - Implement data compression strategies
  - Optimize database operations and indexing
  - Design efficient caching mechanisms
- Runtime Enhancement
  - Implement processing optimizations
  - Optimize memory usage patterns
- Final Presentation Refinement
  - Content development
  - Create data visualizations and demonstrations
  - Prepare summary and key messages