Algorithm Design and Analysis

Final Project

Binus International University

Ir. TRI ASIH BUDIONO, M.I.T.

Jeremy Nathanael Gunawan 2802522960

Kenny Tang 2802517733

William Ekapanna Rusmana 28202523111

Class : L3BC

Course Name : Algorithm Design and Analysis

Course Code: COMP6049001

I. Introduction

1.1 Background

Portfolio optimization theory was formally established in 1952 when Harry Markowitz published his work on modern portfolio theory (Markowitz, 1952). Markowitz provided a mathematical framework that became fundamental for investors who wish to invest can balance between possible profits and potential risks (Logue, 2024). Instead of selecting individual investments, Markowitz changed his focus to the entire portfolio's structure instead, which allows investors to have a secure and better risk-adjusted returns - specifically through diversification where correlation between assets reduce overall portfolio volatility (Logue, 2024). The concept of diversification to portfolio management is essential as it reduces overall risk without sacrificing expected returns (Logue, 2024). The efficient frontier, by Markowitz, represents the set of portfolios that offer the highest expected return for a given level of risk, and vice versa (Logue, 2024). Understanding risk-return trade-off is fundamental as it becomes a framework that aids investors in creating portfolios that syncs more with an individual risk tolerance and investment objectives (GuidedChoice, 2025).

In order to implement an effective portfolio optimization, investors require a metric that measures how well the portfolio finds balance in risk and return (Fernando, 2025). The Sharpe ratio calculates excess return per unit of risk taken; it allows investors to compare the risk-adjusted performance of investment portfolios (Wall Street Prep, n.d.). In a practical circumstance, applying the Sharpe ratio is complicated by the unpredictability of the fluctuating market condition - which poses difficulty in producing reliable estimates for expected returns and volatility (Kircher & Rösch, 2021). From here, Monte Carlo simulations become the solution to this problem. The simulation generates thousands of random scenarios based on historical data to estimate expected returns and standard deviations (Alzahrani, 2024; Li, 2023). In other words, the simulations create outcomes from historical patterns. These results can then be used as a foundation to optimize investor's portfolios to achieve the highest possible Sharpe ratio (Meher & Mishra, 2024). Combining Monte Carlo simulation with Sharpe ratio optimization, this strategy improves on the traditional mean-variance optimization by reducing the estimation risk and improving overall portfolio stability (Faggi, 2021).

There exists several algorithmic approaches to determine the optimal portfolio weights once expected returns and risk estimates are calculated. Equal-weight allocation is the simplest strategy. This method treats all assets as equally important and allocating the same proportion to each (Tamplin, 2025; Dornel, 2025). Dynamic programming uses knapsack-based models to test all possible weight combinations

(Malafeyev & Awasthi, 2017; Vaezi et al., 2019). A problem arises where it becomes very slow as the number of input/assets increase. This makes it less practical with larger portfolios (Zhang et al., 2022). It is emphasized that choosing the right algorithm will matter because each algorithm has its own benefits - such as speed against quality (Liu & Xiao, 2021).

1.2 Problem Description

This project explore real world application of different algorithms for portfolio optimization; using the monte carlo method to derive the expected return estimates and standard deviations that is serves as the key parameters to in the end determine the weight of the allocations by focussing on the risk adjusted return metric called Sharpe ratio (Fernando, 2025; Li, 2023). Three distinct strategies for portfolio optimization are shown which are equal weight, greedy, and dynamic programming knapsack approaches (Zhang et al., 2022). The equal weight algorithm is simple and fast but operates in linear O(n) time complexity, but also may return lower returns compared to optimization based methods (Tamplin, 2025). Greedy algorithms show rank assets using sharpe ratio, offering a good balance between accuracy and speed with O(n log n) complexity (Roman et al., 2006; Zhang et al., 2022). Dynamic programming, using knapsack-based models and prioritizing the Sharpe ratio, can find the best solution but requires higher computational effort with $O(m^2 W^2)$ complexity (Malafeyev & Awasthi, 2017; Zhang et al., 2022).

This project will compare on how well the three algorithms balance trade-offs by using Sharpe ratio as the main performance metric or baseline for comparing the best performance (Fernando, 2025). Equal-weight divides capital equally across all assets and uses Monte Carlo simulation to estimate returns (Dornel, 2025). Greedy and dynamic programming use Monte Carlo estimates of expected returns and standard deviations to find the best weights (Li, 2023; Zhang et al., 2022). We tested all three portfolio sizes (50, 100, 250, and 500 stocks), measuring Sharpe ratios, computational speed, memory usage, and actual performance. These tests are to see which algorithm works best for different situations (Liu & Xiao, 2021).

II. Related Work

Portfolio optimization research has looked at greedy algorithms, dynamic programming, and equal-weight approaches as different solutions to allocate assets. Greedy algorithms rank stocks by Sharpe ratio and pick the top ones, which balances good results with fast computational speed (Roman et al., 2006; Zhang et al., 2022). Dynamic programming uses knapsack models to test all possible weight combinations for the best solution; the problem is it gets much slower as the portfolio grows (Vaezi et al.,

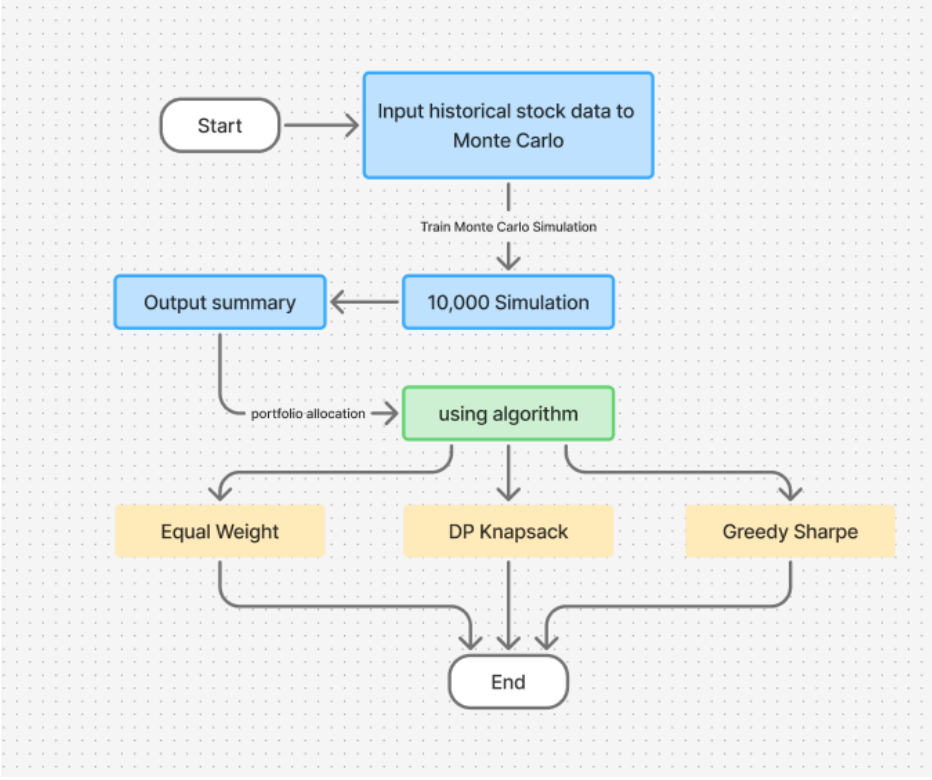
2019; Malafeyev & Awasthi, 2017). Equal-weight is the simplest approach, splitting money equally across all stocks; it handles estimation errors well but usually gives lower returns (Tamplin, 2025; Rakszawski, 2024; Dornel, 2025).

Each method has been studied on its own, but there are few comparisons testing all three together. Each algorithm makes different trade-offs between speed and quality (Liu & Xiao, 2021). This study fills that gap by comparing greedy, dynamic programming, and equal-weight using Monte Carlo simulation across portfolios of 50, 100, 250, and 500 stocks, measuring their Sharpe ratios, speed, and actual performance.

III. Monte Carlo Method & Algorithms Flow Chart

3.1 Overview of the program

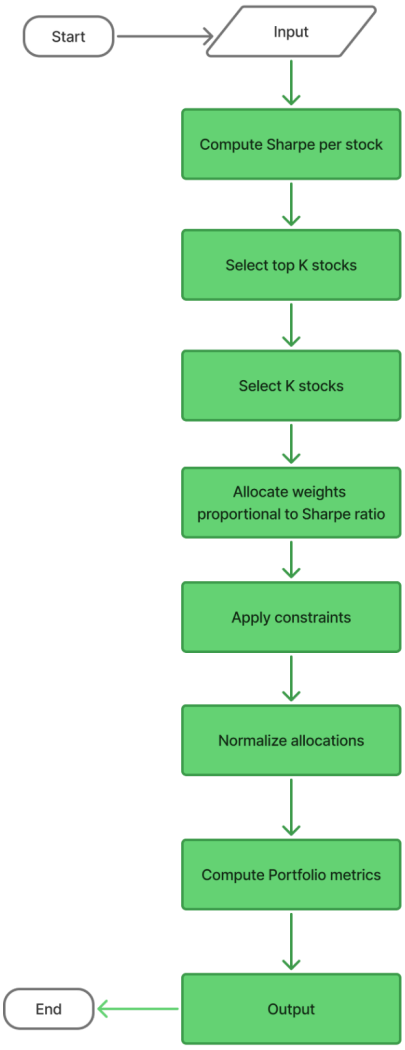
To put it simply this project is a stock allocation project where the weights of each stock are based on the Monte Carlo method's output of projected return and standard deviation; the portfolio allocation is ultimately determined by 3 different algorithms.



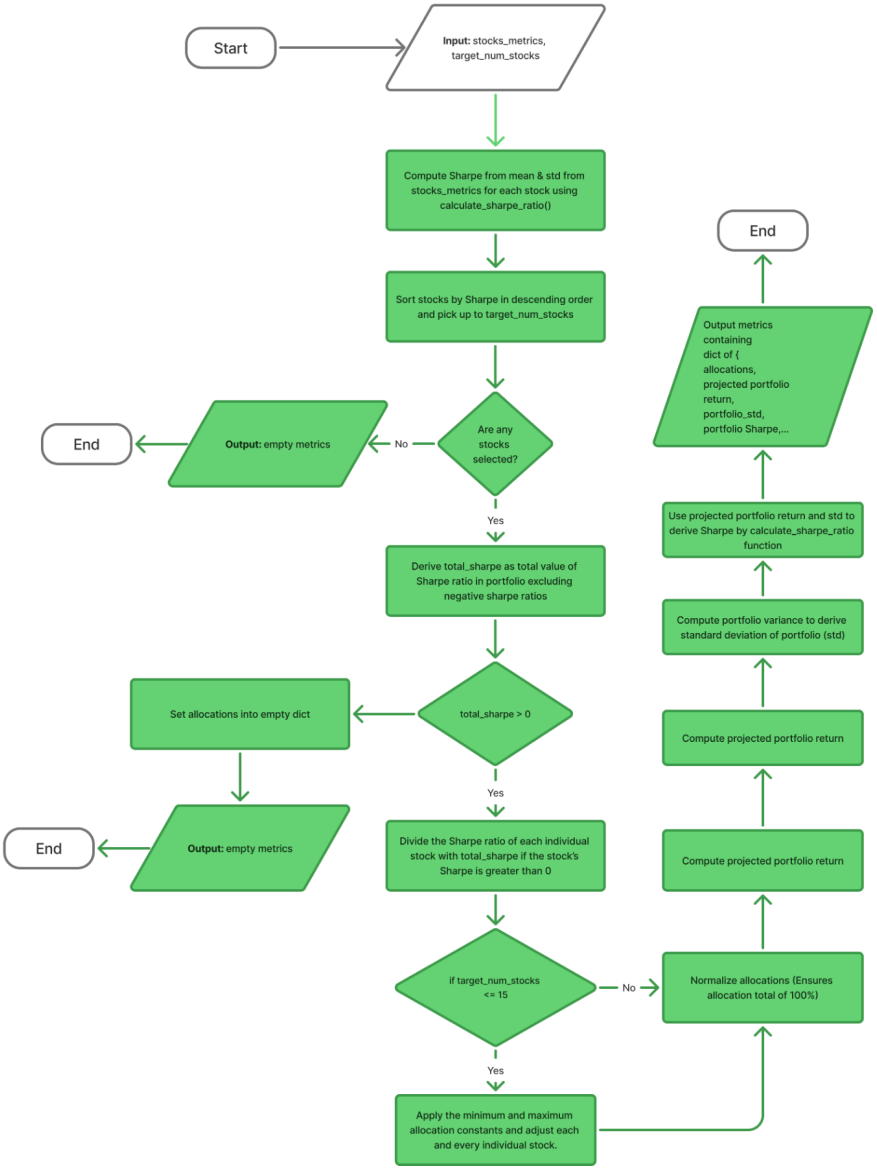
3.2 Algorithm 1: Greedy Sharpe

This algorithm is fast and memory efficient, it chooses the local best choice; in this algorithm it starts with choosing the highest sharpe stocks.

Greedy Sharpe Flow

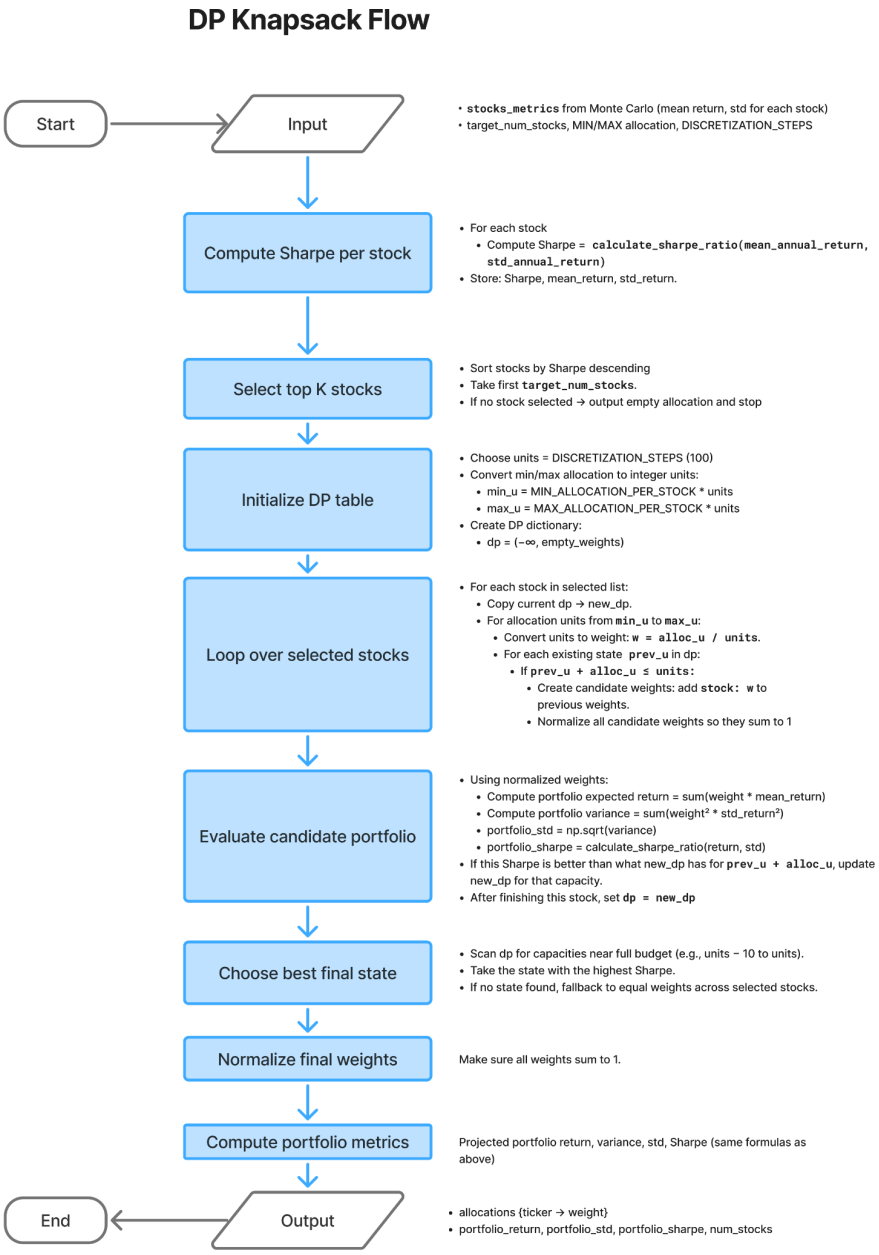


Greedy Sharpe Flow Chart

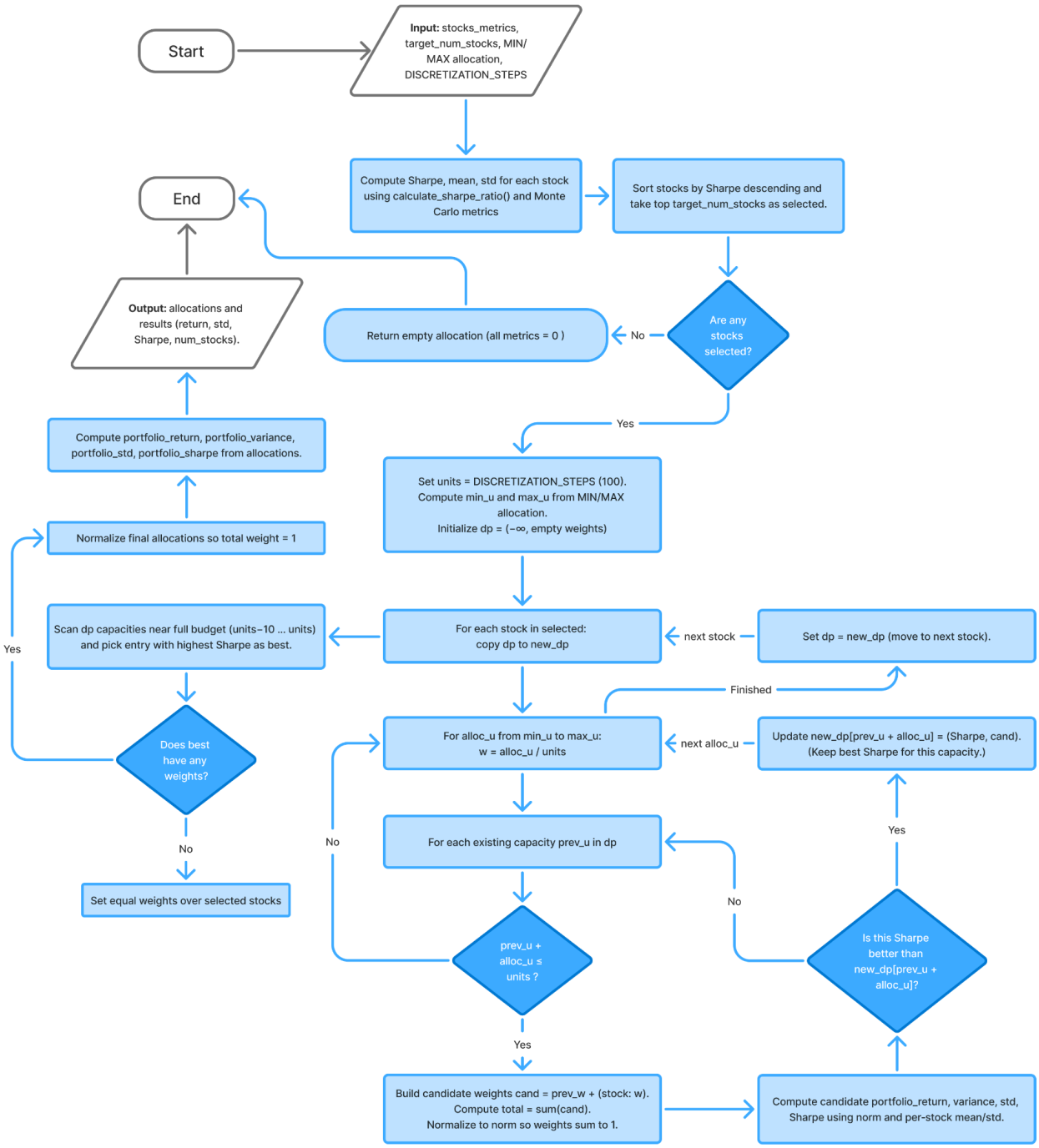


3.3 Algorithm 2: DP Knapsack

In contrast to the greedy algorithm, the dynamic programming knapsack is rather slow in run time and less memory efficient, but it is the algorithm that consistently delivers the highest Sharpe ratio among the other two.

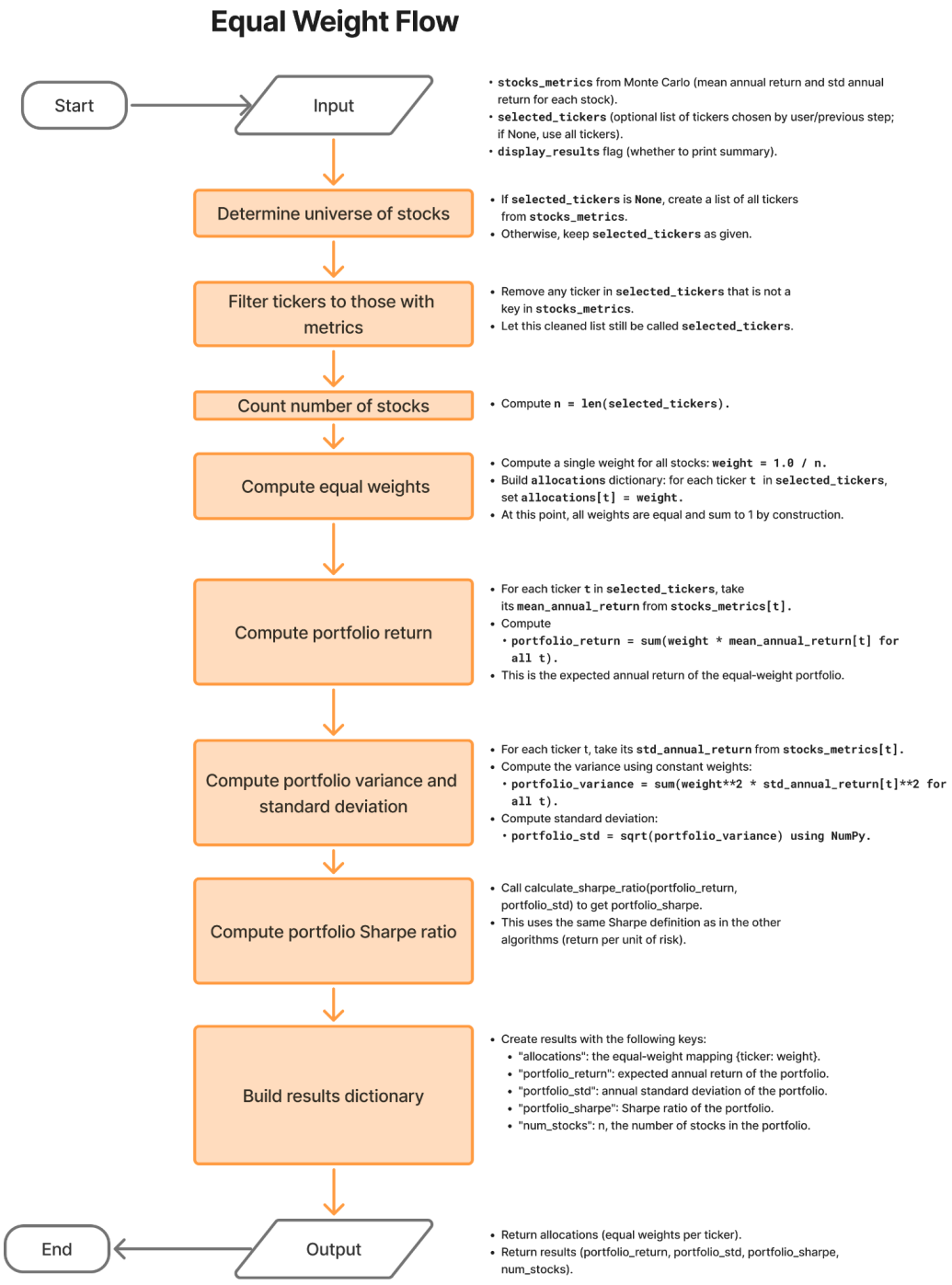


DP Knapsack Flowchart



3.4 Algorithm 3: Equal-Weight

Uniquely, this algorithm is the simplest portfolio allocation algorithm, and the monte carlo expected return is not used here for the allocation process but instead for determining its created portfolio allocation's share and projected annual return.



IV. Complexity Analysis (KENNY)

4.1 Greedy Algorithm

```
# Calculate Sharpe ratios
sharpe_ratios = {
    stock: {
        "sharpe_ratio": calculate_sharpe_ratio(
            metrics["mean_annual_return"], metrics["std_annual_return"]
        ),
        "mean_return": metrics["mean_annual_return"],
        "std_return": metrics["std_annual_return"],
    }
    for stock, metrics in stocks_metrics.items()
}
```

Loop once over all` stock_metric.items()` which means ‘n’ iterations. Each iteration is constant access and one call to `calculate_sharpe_ratio()` which is O(1). So,

Time: T(n) = c * n = O(n)

```
# Select top K stocks by Sharpe ratio
sorted_stocks = sorted(
    sharpe_ratios.items(), key=lambda x: x[1]["sharpe_ratio"], reverse=True
)
selected = sorted_stocks[:target_num_stocks]
```

sharpe_ratios has ‘n’ entries and sorted (python uses timsort for sorting) on ‘n’ elements. Timsort is O(n logn) time average case and O(n) space worst case. There is also slicing [:target_num_stocks] which copies the first ‘m’ elements which means O(m) but ‘m’ is smaller or equal to ‘n’. Since timsort is the dominant,

Time: O(n logn)

Space: O(n)

```
if not selected:
    return {}, {
        "portfolio_return": 0.0,
        "portfolio_std": 0.0,
        "portfolio_sharpe": 0.0,
        "num_stocks": 0,
    }
```

Constant time check and return so **Time: O(1)**

```
# Allocate weights proportional to Sharpe ratio
total_sharpe = sum(
    s[1]["sharpe_ratio"] for s in selected if s[1]["sharpe_ratio"] > 0
)
```

Selected has at most m elements since it is a slice and each iteration is constant time operations.

Time is O(m) but m is a slice and m <= n it means that

Time: O(n)

```
    allocations = {
        stock: metrics["sharpe_ratio"] / total_sharpe
        for stock, metrics in selected
        if metrics["sharpe_ratio"] > 0
    }
```

Iterate at most m items and per loop it is constant work. So,

Time: O(m)

```
    # Apply constraints if using few stocks
    if target_num_stocks <= 15:
        allocations = {
            s: min(max(w, MIN_ALLOCATION_PER_STOCK), MAX_ALLOCATION_PER_STOCK)
            for s, w in allocations.items()
        }
```

At most m allocations, but here we have m <= 15 so it is limited by a constant. So,

Time: O(m) since m <= 15

Space: O(m) since a dictionary of size <= m

```
    # Normalize
    total = sum(allocations.values())
    if total > 0:
        allocations = {s: w / total for s, w in allocations.items()}
```

Sum is <= m values and dictionary comprehension is <= m elements. So,

Time: O(m)

Space: O(m) for temporary dictionary of size <= m

```
# Compute portfolio metrics
portfolio_return = sum(
    allocations[s] * sharpe_ratios[s]["mean_return"] for s in allocations
)
portfolio_variance = sum(
    allocations[s] ** 2 * sharpe_ratios[s]["std_return"] ** 2 for s in allocations
)
```

```
portfolio_std = np.sqrt(portfolio_variance)
portfolio_sharpe = calculate_sharpe_ratio(portfolio_return, portfolio_std)
```

Both the sum is <= m stocks so it is 2O(m) which means

Time: O(m)

```
results = {
    "allocations": allocations,
    "portfolio_return": portfolio_return,
    "portfolio_std": portfolio_std,
    "portfolio_sharpe": portfolio_sharpe,
    "num_stocks": len(allocations),
}
```

constant number of assignment so

Time: O(1)

```
if display_results:
            ...
    for stock in sorted(
        allocations.keys(), key=lambda s: allocations[s], reverse=True
    ):
        print(...)
```

The rest of the code is printing so O(1) and since there sorting it is using timsort however it is `m` instead of `n`.So,

Time: O(mlogm)

Space: O(m)

Combine all

Time Big-O = O(nlogn)

Due to sorting stocks by sharpe ratio

Time best case = O(n), if `stock_metrics` is empty or `target_num_stocks = 0`, return early which is O(1), but for building `sharpe_ratios` becomes O(n)

Time worst case = O(nlogn), normal case with n stocks (compute, sort and post processing)

Space Big-O = O(n)

Due to storing sharpe metrics and sorted lists

Space best case = O(1), if `stock_metrics` is empty or `target_num_stocks = 0`, return early

Space worst case = O(n), normal case with n stocks (compute, sort and post processing)

4.2 DP knapsack

```
# Calculate Sharpe ratios and select top K stocks
sharpe_ratios = {
            ...
    for stock, metrics in stocks_metrics.items()
}
sorted_stocks = sorted(
    sharpe_ratios.items(), key=lambda x: x[1]["sharpe_ratio"], reverse=True
)
selected = [s for s, _ in sorted_stocks[:target_num_stocks]]
```

Similar to greedy, to build `sharpe_ratios` is O(n). sorting 'n' stocks is O(n logn). Slice and list comprehension for `selected` is O(m) where m is <= n.

Time: O(n logn)

```
# DP: dp[units] = (best_sharpe, weights)
units = DISCRETIZATION_STEPS
dp = {0: (float("-inf"), {})}
min_u = max(1, int(MIN_ALLOCATION_PER_STOCK * units))
max_u = int(MAX_ALLOCATION_PER_STOCK * units)

for stock in selected:
    new_dp = dp.copy()
    for alloc_u in range(min_u, max_u + 1):
        w = alloc_u / units
        for prev_u in list(dp.keys()):
            if prev_u + alloc_u <= units:
                                    ... compute cand, norm ...
                                    ... compute ret, var, sharpe …
                                    ... update new_dp ...
    dp = new_dp
```

So here is the part where we use the dynamic programming, the outer loop is a loop over `selected` which is 'm'. `alloc_u` loop from `min_u` to `max_u` which is proportional to 'u'. `prev_loop` loop in worst case the dp might have up to 'W' keys. So, worst case iteration count is

About m * W * W = mW^2

And for the inner loop

```
            cand = {**prev_w, stock: w}
            total = sum(cand.values())
            norm = (
```

```
                {s: v / total for s, v in cand.items()} if total > 0 else cand
            )

            ret = sum(
                norm.get(s, 0) * sharpe_ratios[s]["mean_return"]
                for s in selected
            )
            var = sum(
                norm.get(s, 0) ** 2 * sharpe_ratios[s]["std_return"] ** 2
                for s in selected
            )
            sharpe = calculate_sharpe_ratio(ret, np.sqrt(var))
```

The size of `cand` / `norm` is at most the number of stocks chosen so far and <= ‘m’. The two `sum(...)` loop over `selected` which is O(m) each. So,

Total DP time: T(n) = m * W * W * m = O(m^2 * W^2)

Since we use `DISCRETIZATION_STEPS = W` the fixed constant and in our code we use 100. We can treat W^2 as a big constant. So,

T(n) = O(m^2) with a very large constant but more descriptive:

T(n) = O(m^2 * W^2)

```
# Extract best allocation
best = max(
    (dp[u] for u in range(max(0, units - 10), units + 1) if u in dp),
    default=(0, {}),
    key=lambda x: x[0],
)
allocations = best[1] if best[1] else {s: 1.0 / len(selected) for s in selected}

# Normalize
total = sum(allocations.values())
if total > 0:
    allocations = {s: w / total for s, w in allocations.items()}
```

`max(...)` is at most 11 possible `u` values in the loop because of the range so it is constant time. Fallback equal allocation is O(m) and normalization is O(m) for the loop through `selected` which is size ‘m’. So,

Time: O(m)

```
# Compute metrics
portfolio_return = sum(
    allocations[s] * sharpe_ratios[s]["mean_return"] for s in allocations
```

```
)
portfolio_variance = sum(
    allocations[s] ** 2 * sharpe_ratios[s]["std_return"] ** 2 for s in allocations
)
```

The two sum function is looping over <= m stocks. So,

Time: O(m)

Combine all

Sharpe and sorting O(n logn) + dynamic programming O(m^2 * W^2) + the rest O(m). Since dynamic programming dominate

Time Big-O = O(n logn + m^2)

where for fixed W is constant `DISCRETIZATION_STEPS`. So instead of O(m^2 * W^2 + nlogn) it can be O(nlogn + m^2) for `DISCRETIZATION_STEPS`

Time best case = O(nlogn), if m is very small and DP cost is constant

Time worst case = O(nlogn + m^2 * W^2), select many stocks m and capacity W is fixed but non-trivial

Space Big-O = O(n)

where dp at most W + 1 entries so each storing a Sharpe and `weights` dictionary. In worst case, many `dp[u]` states may store many combinations and giving the O(W * m) weight entries. There is also `sharpe_ratios and `sorted_stocks` which are O(n). The rough bound will be O(n + W * m) but with fixed u and m <= n it will be O(n).

Space best case = O(n), still `sharpe_ratios` and `sorted_stocks`

Space worst case = O(n + W * m), select many stocks m and capacity W is fixed but non-trivial

4.3 Equal Weight

```
if selected_tickers is None:
    selected_tickers = list(stocks_metrics.keys())

# Filter to tickers present in metrics
selected_tickers = [t for t in selected_tickers if t in stocks_metrics]

n = len(selected_tickers)
```

Here we build the list of keys which is O(n). Filtering `selected_tickers` for each look up is average O(1) for a dictionary but per ticker it will be a total of O(n). So, overall

Time: O(n)

```
# Equal weight 1/N
weight = 1.0 / n
allocations = {t: weight for t in selected_tickers}
```

Here the code split the weight equally and loop one pass over n tickers. So

Time: O(n)

```
# Compute portfolio metrics
portfolio_return = sum(
    weight * stocks_metrics[t]["mean_annual_return"] for t in selected_tickers
)

portfolio_variance = sum(
    (weight**2) * (stocks_metrics[t]["std_annual_return"] ** 2)
    for t in selected_tickers
)
portfolio_std = np.sqrt(portfolio_variance)
portfolio_sharpe = calculate_sharpe_ratio(portfolio_return, portfolio_std)
```

Each `sum(...)` loop is over n ticker which is O(n) from the `selected_tickers` and loop once. `np.sqrt` and `calculate_sharpe_ratio` are both O(1). So,

Time: O(n)

Combine all

Time Big-O = O(n)

It is O(n) because all of it are O(n) with a few constant O(1)

Time best case = O(1), if no selected tickers

Time worst case = O(n), if goes through all lists

Space Big-O = O(n)

`selected_tickers` is a list of n tickers so O(n), allocations is a dictionary of n entries so O(n).

Space best case = O(1), if no selected tickers

Space worst case = O(n), if goes through all lists

V. Results

6.1 Runtime and Memory Usage results

N = 50 Stocks

5 Run time and peak memory (50 Stocks)

Algorithm	Run	Time (ms)	Memory (Bytes)
Greedy	1	0.95	9 760
Greedy	2	0.51	9 760
Greedy	3	0.70	9 760
Greedy	4	0.56	9 760
Greedy	5	0.44	9 760
DP Knapsack	1	18 447.40	198 712
DP Knapsack	2	18 081.20	195 936
DP Knapsack	3	18 057.00	195 912
DP Knapsack	4	19 372.97	195 912
DP Knapsack	5	18 266.80	195 912
Equal Weight	1	0.73	2 888
Equal Weight	2	0.93	2 888
Equal Weight	3	0.39	2 888
Equal Weight	4	0.19	2 888
Equal Weight	5	0.15	2 888

Method	Time (ms)	Memory (Bytes)	Sharpe	Stocks	Projected Return

Greedy	0.63 ± 0.20 ms	9760	~3.0194	50	~26.01%
DP Knapsack	18 445.07 ± 542.17	196477	~3.2612	47	~22.54%
Equal Weight	0.48 ± 0.34	2888	~3.0126	50	~20.30%

N = 100 Stocks

5 Run time and peak memory (100 Stocks)

Algorithm	**Run**	**Time (ms)**	**Memory (Bytes)**
Greedy	1	0.63	18 160
Greedy	2	0.57	17 664
Greedy	3	0.58	17 664
Greedy	4	0.58	17 664
Greedy	5	0.57	17 664
DP Knapsack	1	71 514.16	179 032
DP Knapsack	2	70 063.63	178 592
DP Knapsack	3	69 314.41	178 568
DP Knapsack	4	69 577.78	178 568
DP Knapsack	5	69 785.89	178 568
Equal Weight	1	0.35	5 832
Equal Weight	2	1.62	5 832
Equal Weight	3	0.28	5 832
Equal Weight	4	0.28	5 832
Equal Weight	5	0.43	5 832

Method	**Time (ms)**	**Memory (Bytes)**	**Sharpe**	**Stocks**	**Projected Return**

Greedy	0.58 ± 0.02	17 763	~4.5168	57	~69.62%
DP Knapsack	70 051.17 ± 862.81	178 666	~6.6653	43	~47.49%
Equal Weight	0.59 ± 0.58	5 832	~3.0600	100	~24.24%

N = 250 Stocks

5 Run time and peak memory (250 Stocks)

Algorithm	**Run**	**Time (ms)**	**Memory (Bytes)**
Greedy	1	2.62	62 320
Greedy	2	1.46	61 824
Greedy	3	1.44	61 824
Greedy	4	1.45	61 824
Greedy	5	1.44	61 824
DP Knapsack	1	424 814.30	292 784
DP Knapsack	2	428 963.50	292 344
DP Knapsack	3	451 807.82	292 320
DP Knapsack	4	499 481.99	292 320
DP Knapsack	5	511 487.41	292 320
Equal Weight	1	2.78	12 104
Equal Weight	2	0.88	12 104
Equal Weight	3	0.73	12 104
Equal Weight	4	1.04	12 104
Equal Weight	5	1.00	12 104

Method	Time	Memory (Bytes)	Sharpe	Stocks	Projected Return
Greedy	1.68 ± 0.52	61 923	~7.0414	143	~53.74%
DP Knapsack	463 311.00 ± 40 072.99	292 418	~9.4589	82	~41.15%
Equal Weight	1.29 ± 0.84	12 104	~3.5985	250	~16.67%

N = 500 Stocks

5 Run time and peak memory (500 Stocks)

Algorithm	Run	Time (ms)	Memory (Bytes)
Greedy	1	2.41	124 872
Greedy	2	2.22	124 376
Greedy	3	2.22	124 376
Greedy	4	2.66	124 376
Greedy	5	2.35	124 376
DP Knapsack	1	1 852 353.61	254 136
DP Knapsack	2	2 178 042.91	253 696
DP Knapsack	3	2 349 604.92	253 672
DP Knapsack	4	2 806 127.71	253 672
DP Knapsack	5	2 662 117.63	253 672
Equal Weight	1	2.94	23 876
Equal Weight	2	1.35	23 876
Equal Weight	3	1.47	23 876
Equal Weight	4	2.18	23 876
Equal Weight	5	1.29	23 876

Method	Time (ms)	Memory (Bytes)	Sharpe	Stocks	Projected Return
Greedy	2.37 ± 0.18	124 475	~2.9441	146	~503.29%
DP Knapsack	2 369 649.36 ± 381 038.30	253 770	~61.5955	34	~111.12%
Equal Weight	1.85 ± 0.71	23 876	~1.1736	500	~201.68%

Analysis

N = 50 stocks

- Fastest time: Equal Weight (0.48 ± 0.34 ms)
- Lowest memory: Equal Weight (2 888 bytes)
- Highest Sharpe: DP Knapsack (≈3.2612)
- Highest projected return: Greedy (≈26.01%)

N = 100 stocks

- Fastest time: Greedy (0.58 ± 0.02 ms)
- Lowest memory: Equal Weight (5 832 bytes)
- Highest Sharpe: DP Knapsack (≈6.6653)
- Highest projected return: Greedy (≈69.62%)

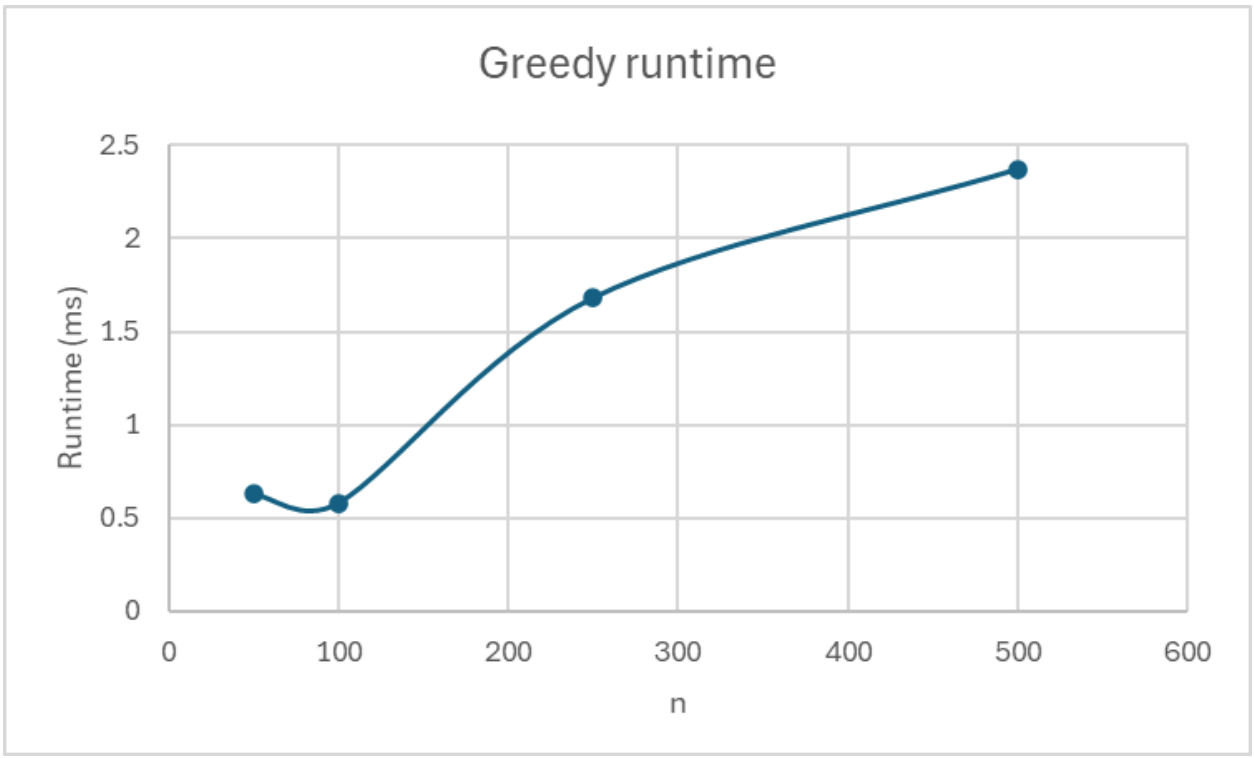
N = 250 stocks

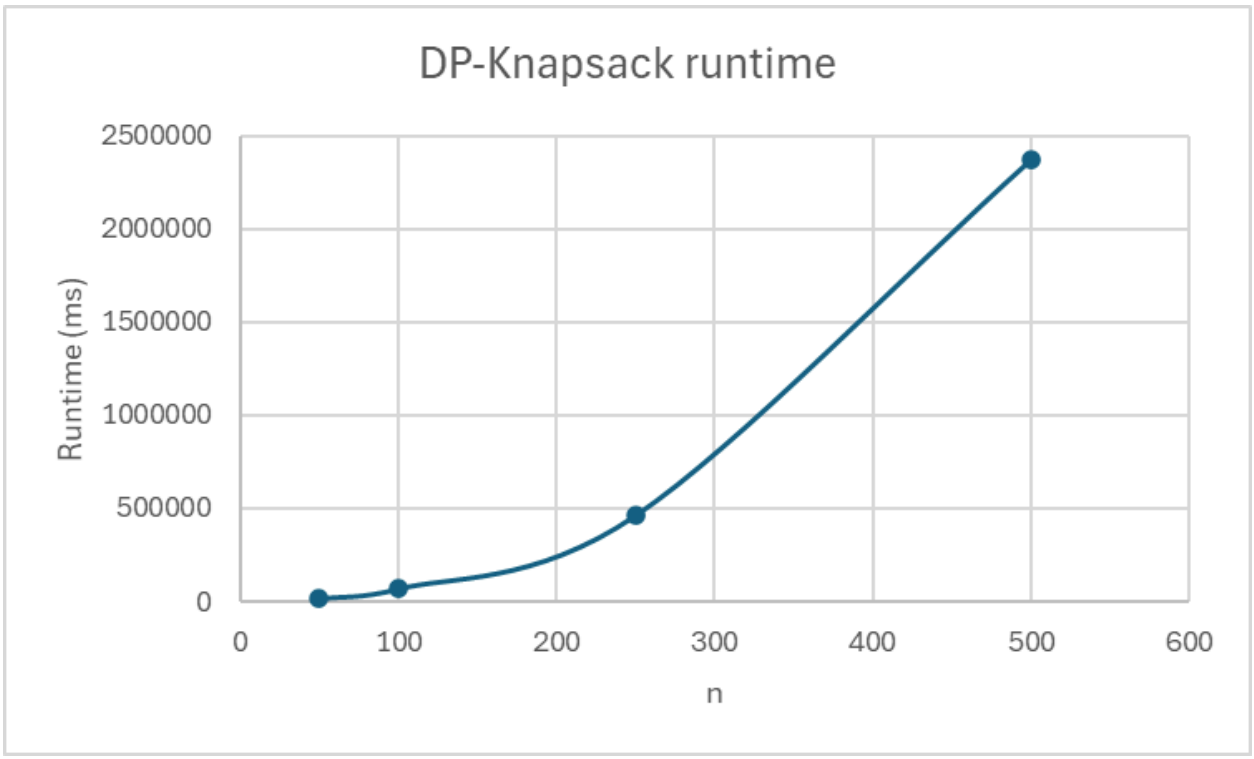
- Fastest time: Equal Weight (1.29 ± 0.84 ms)
- Lowest memory: Equal Weight (12 104 bytes)
- Highest Sharpe: DP Knapsack (≈9.4589)
- Highest projected return: Greedy (≈53.74%)

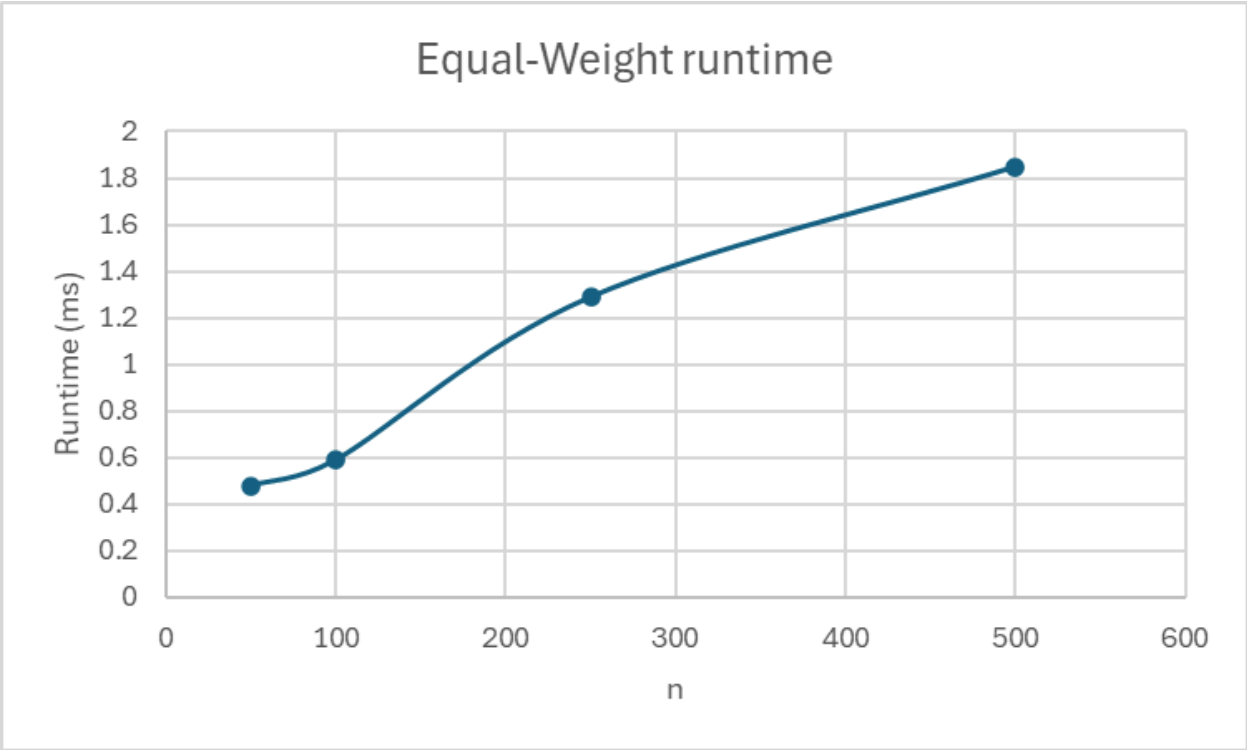
N = 500 stocks

- Fastest time: Equal Weight (1.85 ± 0.71 ms)
- Lowest memory: Equal Weight (23 876 bytes)
- Highest Sharpe: DP Knapsack (≈61.5955)
- Highest projected return: Greedy (≈503.29%)

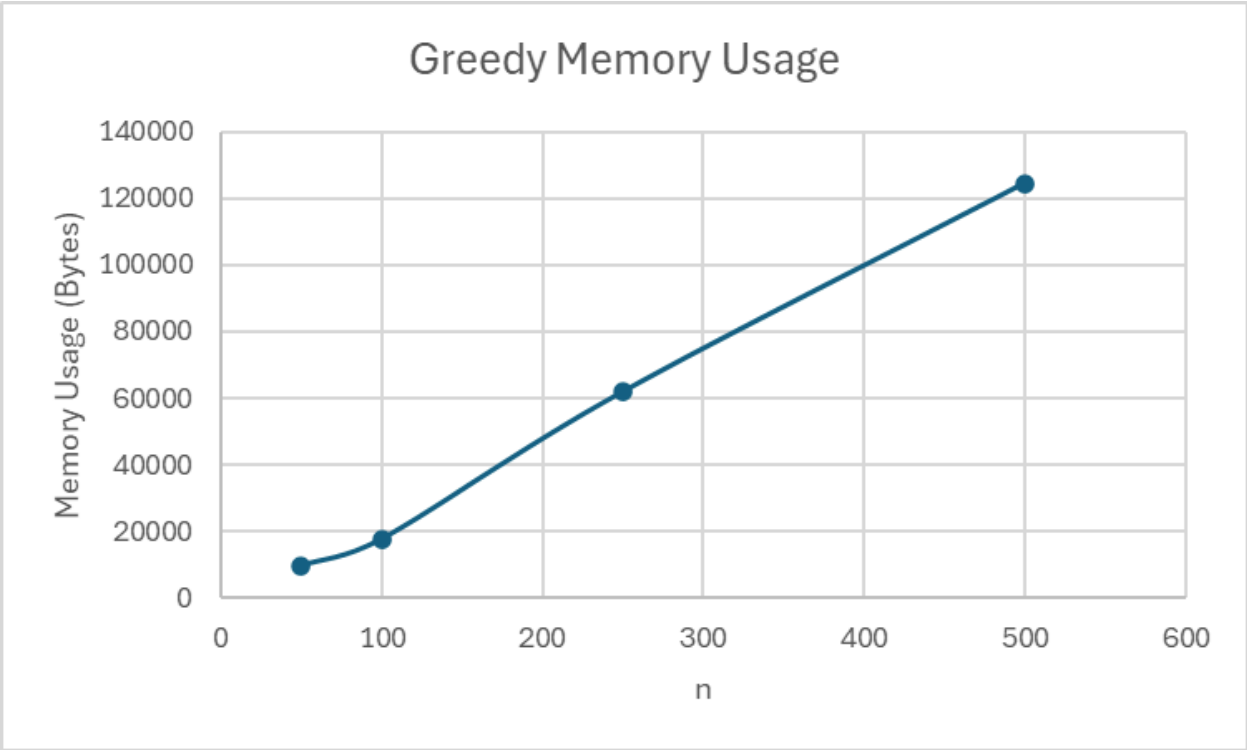
6.2 Run time between different (n)

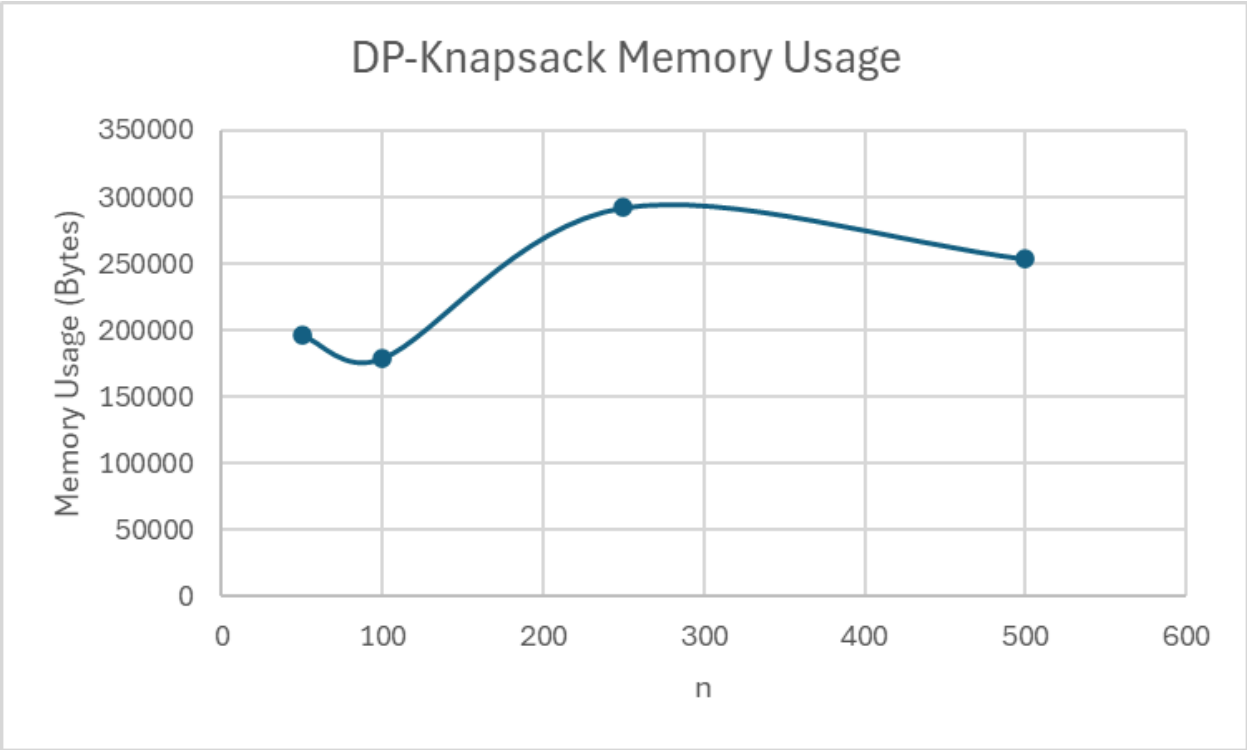

Greedy runtime
Runtime (ms)
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
n

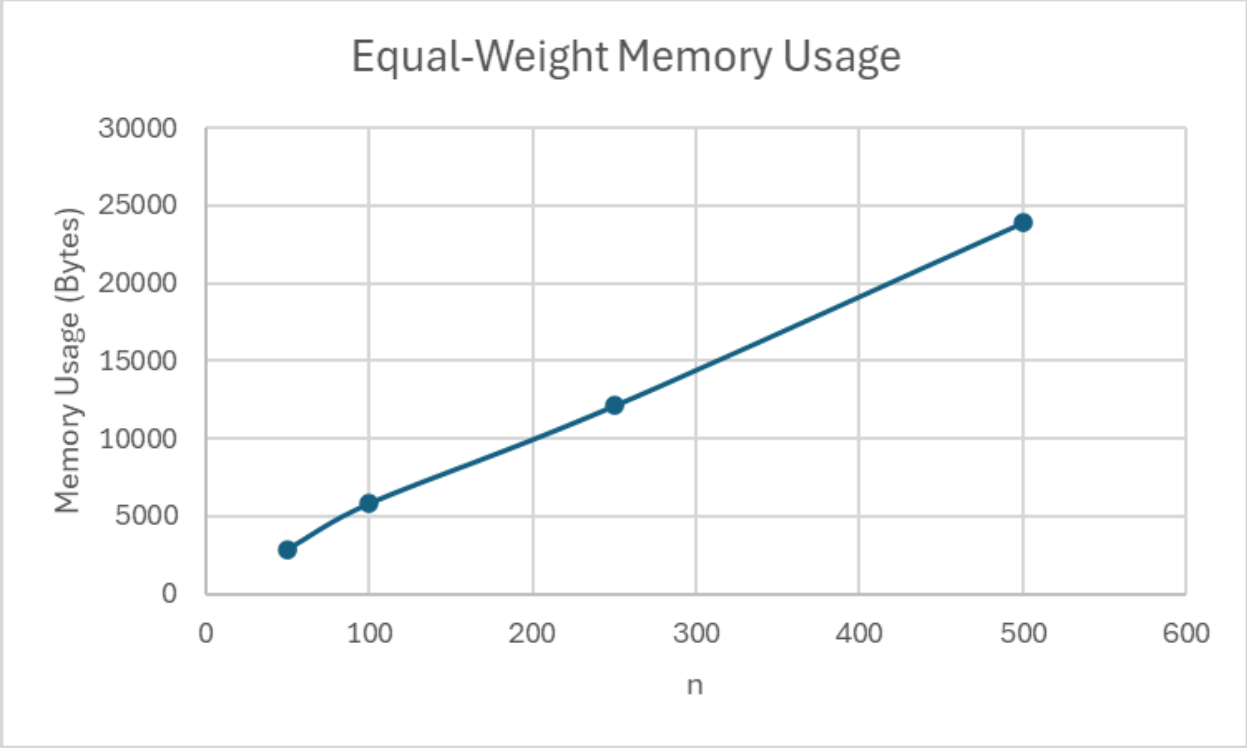

DP-Knapsack runtime
Runtime (ms)
2500000
2000000
1500000
1000000
500000
0
0
100
200
300
400
500
600
n



6.3 Memory Usage between different (n)




DP-Knapsack Memory Usage
Memory Usage (Bytes)
350000
300000
250000
200000
150000
100000
50000
0
0
100
200
300
400
500
600
n


Equal-Weight Memory Usage
Memory Usage (Bytes)
30000
25000
20000
15000
10000
5000
0
0
100
200
300
400
500
600
n

VI. Trade-off & Discussions

6.1 Trade off

1. Speed and optimal
    - Equal Weight: fastest, simplest, and lowest memory at all stock sizes, but lowest return.
    - Greedy: almost as fast as Equal Weight,consistently highest return
    - DP Knapsack: always highest Sharpe, but runtime is order of magnitude slower than the other two.
2. Risk and return
    - Knapsack improves the Sharpe very noticeably over the Equal weight (such as ~3.26 and ~3.01 at 50 stocks, ~61.60 and ~1.17 at 500 stocks) while giving lower return than greedy.
    - Greedy shows deliver the largest and highest return, but the Sharpe is always below DP knapsack which shows that it is weaker risk-adjusted.

6.2 Discussion

- Algorithm behaviour and theory

Link the runtime growth to our Big-O analysis, our knapsack shows the big difference from milliseconds to millions of milliseconds at 500 stocks which match the theory complexity, while greedy and equal weight stay almost linear and still be fast at all sizes. The optimizer (dynamic programming often selects far fewer stocks than available which is expected for us that is prioritizing and concentrating on the Sharpe maximizing for a portfolio that can make the best assets instead of using all the stocks available.

- Choice of algorithm

Equal weight or greedy is suitable when fast decisions, low complexity or when we have limited hardware to run that is more important than the highest Sharpe. The DP knapsack is only good for small amounts of stocks because it justify the long runtimes for Sharpe. It is also mentioned

that greedy gives the highest projected return but worse risk-adjusted or Sharpe, so it is better for aggressive investors, DP knapsack is more used with risk investors which focus on Sharpe.

VII. Conclusion & Recommendations

7.1 Conclusion

Our project compared three portfolio optimization algorithms (equal-weight, greedy, and dynamic programming) using Monte Carlo simulation across portfolio sizes ranging from 50 to 500 stocks. We did a thorough algorithm analysis - specifically computational speed, memory usage, Sharpe ratio, and the outcome to evaluate the trade-offs between quality and efficiency.The greedy algorithm is the most balance algorithm; it not only achieved highest projected return across all portfolio sizes, but it also has maintained computational speed comparable to equal-weight, with average runtimes under 3 milliseconds even at 500 stocks. The dynamic programming knapsack algorithm resulted with the highest Sharpe ratios, reaching 61.6 at 500 stocks compared to greedy's and equal-weight's - which is 2.94 and 1.17 respectively. However, the dynamic programming knapsack algorithm required the most computation time, with runtimes of 2 million milliseconds at 500 stocks. Equal-weight remained the fastest algorithm in all testing of size and required the least computational memory, however it provided the lowest projected returns and Sharpe ratios.

This outcome has proven that algorithm selection requires crucial thinking of choosing the right algorithm on a specific investment. Equal-weight is most suitable as a baseline when computational speed and simplicity matter most - such as in volatile markets where utilizing historical data has risk (Tamplin, 2025; Dornel, 2025). Greedy is better for quick decisions or when dealing with large quantity of stocks. It gives the fastest return while computational speed fast, and it focuses on best assets based on Sharpe ratio (Zhang et al., 2022). Dynamic programming is recommended for portfolios with under 100 stocks. This is due to the fact that it requires the most computational waiting time, however it does minimize risk and provide better results (Liu & Xiao, 2021).

For future works, testing with larger portfolios using parallel computing to speed up dynamic programming. Adding constraints like sector diversity and minimum position sizes, and also evaluating how well each algorithm works in different market conditions and time periods. Combining the algorithms or switching them based on market volatility could make a better practical strategy for practicality.

7.2 Recommendations

For this project for practical **finance** application we suggest modification of algorithms to follow specific key goals such as **diversification**, **highest risk adjusted return,** etc. The most important thing to keep in mind for this algorithm is to keep in mind its dependence on **historical performances** of the stocks, therefore a long data range is recommended; established companies that have been performing for **numerous decades** and have a relatively **stable historical growth** are best candidates to be allocated by this program.

Algorithmically, the algorithms are practical allocation algorithms with distinct strengths to one another, with both greedy and equal weight being highly efficient and would take mere seconds to allocate thousands of stock selections; DP knapsack achieving consistently high Sharpe ratios across the board.

Recommendations for each algorithm:

Equal Weight - This is your go-to strategy to start with.
Why it's good: It's a straightforward approach that can actually beat more complicated models when the market is really jumpy. This is because it doesn't depend on predictions from past data, which can sometimes be wrong.

Greedy Sharpe - This one's for making quick choices or when you're dealing with a lot of money.
Why it's good: It's super fast to calculate, but it might not see the bigger picture. It could pick investments that seem great on their own but don't actually work well together as a group.

DP Knapsack - Use this when you have a tight budget and can only buy whole shares of stocks.
Why it's good: Unlike the usual way of figuring things out, the DP Knapsack method sees stocks as individual items you can buy. This is perfect for folks with less cash who can't purchase parts of shares.

VIII. References

Alzahrani, A. A. (2024, October 6).

Modern portfolio theory, Monte Carlo simulations & CVaR for smarter investment decisions. *GoFar*. https://www.gofar.ai/p/modern-portfolio-theory-monte-carlo

Dornel, D. (2025, July 6). Diversifying with equal-weighted strategies. *BNP Paribas*.

https://viewpoint.bnpparibas-am.com/diversifying-with-equal-weighted-strategies-en/

Faggi, D. (2021).

A Monte Carlo simulation applied to portfolio management and the Constant Proportion Portfolio Insurance [Undergraduate thesis, LUISS Guido Carli University]. LUISS Theses Repository. https://tesi.luiss.it/31670/

Fernando, J. (2025, May 15).

Sharpe ratio: Definition, formula, and examples. *Investopedia*. https://www.investopedia.com/terms/s/sharperatio.asp

GuidedChoice. (2025).

Harry Markowitz's modern portfolio theory: The efficient frontier. https://guidedchoice.com/modern-portfolio-theory/

Kircher, F., & Rösch, D. (2021).

A shrinkage approach for Sharpe ratio optimal portfolios with estimation risks. *Journal of Banking & Finance*, *133*, 106281. https://doi.org/10.1016/j.jbankfin.2021.106281

Li, A. (2023). Portfolio optimization by Monte Carlo simulation.

Proceedings of the 2nd International Conference on Financial Technology and Business Analysis (pp. 136-142). https://doi.org/10.54254/2754-1169/50/20230568

Liu, S., & Xiao, C. (2021).

Application and comparative study of optimization algorithms in financial investment portfolio problems. *Mobile Information Systems*, *2021*, Article 3462715. https://doi.org/10.1155/2021/3462715

Logue, A. C. (2024, December 31).

Modern portfolio theory: Definition, examples, & limitations. In *Encyclopædia Britannica*. https://www.britannica.com/money/modern-portfolio-theory-explained

Malafeyev, O., & Awasthi, A. (2017). Dynamic optimization of a portfolio.

arXiv. https://arxiv.org/abs/1712.00585

Markowitz, H. (1952). Portfolio selection.

The Journal of Finance, *7*(1), 77-91. https://doi.org/10.2307/2975974

Meher, P., & Mishra, R. K. (2024).

Risk-adjusted portfolio optimization: Monte Carlo simulation and rebalancing. *Australasian Accounting, Business and Finance Journal*, *18*(3), 85-102. https://doi.org/10.14453/aabfj.v18i3.10

Rakszawski, B. (2024, October 30). A better way to equal weight. *VanEck*.

https://www.vaneck.com/us/en/blogs/moat-investing/a-better-way-to-equal-weight/

Roman, D., Darby-Dowman, K., & Mitra, G. (2006).

Portfolio construction based on stochastic dominance and target return distributions. *Mathematical Programming*, *108*(2-3), 541-569. https://doi.org/10.1007/s10107-006-0722-8

Swedroe, L. (2024, January 19).

Outperforming cap- (value-) weighted and equal-weighted portfolios. *Alpha Architect*. https://alphaarchitect.com/equally-weighted-portfolios/

Tamplin, T. (2025, July 10). Equal-weighted portfolio. *Finance Strategists*.

https://www.financestrategists.com/wealth-management/risk-profile/diversification/equal-weighted-portfolio/

Vaezi, F., Sadjadi, S. J., & Makui, A. (2019).

A portfolio selection model based on the knapsack problem under uncertainty. *PLOS ONE*, *14*(5), e0213652. https://doi.org/10.1371/journal.pone.0213652

Wall Street Prep. (n.d.). *Sharpe ratio*. https://www.wallstreetprep.com/knowledge/sharpe-ratio/

Zhang, X., Zhang, L., Zhou, Q., & Jin, X. (2022).

Greedy strategies with multiobjective optimization for investment portfolio problem modeling. *Computational Intelligence and Neuroscience*, *2022*, Article 4862772. https://doi.org/10.1155/2022/4862772

IX. Program manual

1. Start with cloning this repository to your local desktop (Detailed showing on tutorial video)
2. Install the requirements of the app using the command:

```
pip install -r requirements.txt
```

Once requirements are installed you are all set to running the program.
*The main 3 algorithms are stored in the algorithms folder
*data folder contains all the historical stock data for this project

Programs to run

app.py
This program is a Streamlit based project which is a python framework for creating interactive web applications for data visualization; to run this app enter the command:

```
streamlit run app.py
```

main.py
This program is the benchmarking program of the 3 algorithms, to run the standard python command below:

```
python main.py
```

X. Appendix (github link/tutorial video)

Github link to access repository:
https://github.com/WILLIAM-RUSMANA/MonteCarlo-Portfolio-Optimization

Installation and run tutorial:
https://youtu.be/FZfdJM6kSlDo