OpenClassrooms
Python Developer Path

Project 7: Solve Problems Using Algorithms in Python

Presented by: Abdoul Baki Seydou

Content

1 - Scenario

Design an algorithm that will maximize our clients' profits.

The algorithm must:

- Read a file containing information about shares.
- Try out all the different combinations of shares that fit the constraints.
- Suggest a list of the most profit-yielding shares.

Constraints:

- Each share can only be bought once.
- We cannot buy a fraction of a share.
- We can spend at most 500 euros.

List of shares

Name	Cost (eur)	Profit(%)
Share-1	20	5
Share-2	30	10
Share-3	50	15
Share-4	70	20
Share-5	60	17
Share-6	80	25
Share-7	22	7
Share-8	26	11
Share-9	48	13
Share-10	34	27
Share-11	42	17
Share-12	110	9
Share-13	38	23
Share-14	14	1
Share-15	18	3
Share-16	8	8
Share-17	4	12
Share-18	10	14
Share-19	24	21
Share-20	114	18

2 - Methodology

A straightforward method of solving this problem could be to :

1. Generate all possible combinations of shares.
2. Calculate the total profit and cost for each combination.
3. Then select the combination with the highest profit that meets the constraints.

This type of resolution approach is known as the **brute-force** method. This will be our first approach for solving the problem.

For our brute-force solution, we will use a Python built-in function, ***itertools.combinations***, to generate the combinations and select the one that yields the most profit within the constraints.

In a second approach, we will create an **optimized** version of our brute-force solution that compiles the same results in less than a second, then compare the edge cases, efficiency and performance of the algorithms used to create each solution.

Lastly we will test the optimized solution on past datasets and compare the results.

3 - Brute-force Algorithm

Pseudocode

1 - Import itertools,
2 - Define a function to read the data from a csv file,
3 - Define a function to calculate the total cost of a list,
4 - Define a function to calculate the total return of a list,
5 - Then use the formula below to create the bruteforce function that will return the calculated values.

```
# Initialize variables
best_cost = 0
best_return = 0
best_shares = []

# Try out all possible i combinations of shares
for i in range(1, len(share_list) + 1):
    for combination in itertools.combinations(share_list, i):
        total_cost = calculate_total_cost(combination)
        if total_cost <= max_cost:
            total_return = calculate_total_return(combination)
            if total_return > best_return:
                best_return = total_return
                best_shares = combination
                best_cost = total_cost
# Return the best cost, return, and combination of shares
return best_cost, best_return, best_shares
```

Efficiency and Performance

- ✓ **Exponential time complexity: O(2^n)**
  n = input list size.
  Because the algorithm generates all possible combinations of shares.

  For the 20 shares:

  The number of possible combinations is 2^20 - 1 = 1,048,575

- ✓ **Linear space complexity: O(n)**

  Because algorithm stores only the input shares and the best combination of variables found so far.

- ✓ Our brute-force solution takes about 2 seconds to return the best combination on a PC with good performance, but is not efficient for large number of shares.

4 - Optimized Algorithm

Approach

The description of our initial problem can be fitted in one known as the 01 knapsack problem, described as follow:

Given N items where each item has some weight and profit associated with it, and a bag with maximum capacity W, the goal is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: For each item, we can either put it in or exclude it from the bag, hence the name 01 knapsack problem.

We will address the resolution of our 01 Knapsack problem through an approach know as dynamic programming.

Pseudocode

Solve 01 knapsack problem with dynamic programming :

- We construct an array where the rows represent the shares and the columns represent the amount of money we can spend.
- The first row of the table is filled with zeros, and the remaining rows filled using the formula:

```
def optimized_dynamic(max_cost, dataset):
    items = len(dataset)
    table = [[0 for x in range(max_cost + 1)] for y in range(items + 1)]
    for i in range(1, len(dataset) + 1):
        for j in range(1, max_cost + 1):
            if dataset[i - 1]['cost'] > j:
                table[i][j] = table[i - 1][j]
            else:
                table[i][j] = max(table[i - 1][j], dataset[i - 1]['value']
                                  + table[i - 1][j - dataset[i - 1]['cost']])
```

- The maximum profit can be found in the last cell of the array.

4 - Optimized Algorithm

Efficiency and Performance

- ✓ We avoid re-computation by storing the solution for each row in the table for reuse.
- ✓ It iterates through the array only once.
- ✓ **Linear time & space complexity:** O(n*Max_cost)
- ✓ In contrast with the brute-force solution, the optimized algorithm only requires a two-dimensional array of size n*Max_Cost and is much more memory-efficient.
- ✓ For the given problem, n = 20 and Max_Cost = 500, the time complexity is O(20*500) = O(10,000), it returns the best combination of shares with their total cost and return values in less than a second, on average **0.0040seconds** on a PC of good performance.

Edge cases

1. When the cost values are floats with n number of decimals. In this case, rounding the cost values lead to inaccurate total cost and return, while the multiplication of the costs by 10^n to get accurate results increases the time complexity by 10^n .
    - In order to get accuracy at the expense of performance, the values of the costs are multiplied by 100 to convert them to integers then the results divided by 100.
2. When the maximum cost is less than the cost of the cheapest share. In this case, the maximum profit is zero.
3. When there are no shares available. In this case, the maximum profit is also zero.

Big O: Brute-force Algorithm vs Optimized Algorithm

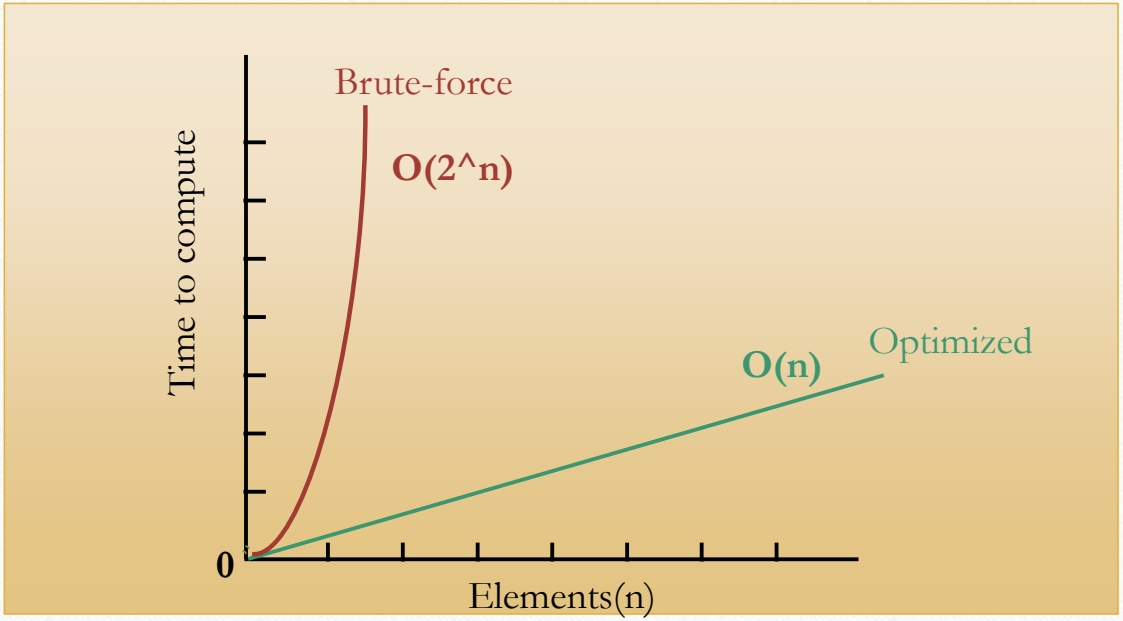

Fig 1 : Time Complexity

5 – Test optimized algorithm

Optimized Algorithm Output vs Sienna's choices

To test the accuracy of our optimized solution, we will run it on past datasets, the client Sienna's datasets.

There are two datasets in csv files, with each containing 1000 shares, and Sienna's corresponding investment decisions also available.

A side-by-side comparison between the optimized algorithm output and Sienna's choices for each dataset is appended.

5.1 - Sienna dataset 1

Optimized Algorithm Output

Budget: 500€

Best combination of shares:
['Share-IFCP', 'Share-EMOV', 'Share-KZBL', 'Share-LRBZ', 'Share-XJMO', 'Share-GIAJ', 'Share-QQTU', 'Share-SKKC', 'Share-ZSDE', 'Share-LGWG', 'Share-WPLI', 'Share-QLMK', 'Share-MLGM', 'Share-FKJW', 'Share-GTQK', 'Share-USSR', 'Share-MTLR', 'Share-LPDM', 'Share-UEZB', 'Share-NHWA', 'Share-GHIZ', 'Share-KMTG']

Total cost: 499.95€

Total return: 198.54€

Execution time: 24.9858 seconds

Sienna's choices

Budget: 500€

Sienna bought: ['Share-GRUT']

Total cost: 498.76€

Total return: 196.61€

Bought by Sienna but not in Optimized Output: ['Share-GRUT']

In Optimized Output but not bought by Sienna: ['Share-IFCP', 'Share-EMOV', 'Share-KZBL', 'Share-LRBZ', 'Share-XJMO', 'Share-GIAJ', 'Share-QQTU', 'Share-SKKC', 'Share-ZSDE', 'Share-LGWG', 'Share-WPLI', 'Share-QLMK', 'Share-MLGM', 'Share-FKJW', 'Share-GTQK', 'Share-USSR', 'Share-MTLR', 'Share-LPDM', 'Share-UEZB', 'Share-NHWA', 'Share-GHIZ', 'Share-KMTG']

5.2 - Sienna dataset 2

Optimized Algorithm Output

Budget: 500€

Best combination of shares: ['Share-ROOM', 'Share-XQII', 'Share-DWSK', 'Share-LFXB', 'Share-VCAX', 'Share-FAPS', 'Share-JGTW', 'Share-JWGF', 'Share-ALIY', 'Share-NDKR', 'Share-SCWM', 'Share-PATS', 'Share-ANFX', 'Share-YFVZ', 'Share-LXZU', 'Share-PLLK', 'Share-ZOFA', 'Share-FWBE', 'Share-IXCI', 'Share-ECAQ']

Total cost: 499.9€

Total return: 197.96€

Execution time: 13.5185 seconds

Sienna's choices

Budget: 500€

Sienna bought: ['Share-ALIY', 'Share-ANFX', 'Share-DWSK', 'Share-ECAQ', 'Share-FAPS', 'Share-FWBE', 'Share-IXCI', 'Share-JGTW', 'Share-JWGF', 'Share-LFXB', 'Share-NDKR', 'Share-PATS', 'Share-PLLK', 'Share-ROOM', 'Share-VCAX', 'Share-XQII', 'Share-YFVZ', 'Share-ZOFA']

Total cost: 489.24€

Total return: 193.78€

Bought by Sienna but not in Optimized Output: []

In Optimized Output but not bought by Sienna: ['Share-SCWM', 'Share-LXZU']

Thank you

End