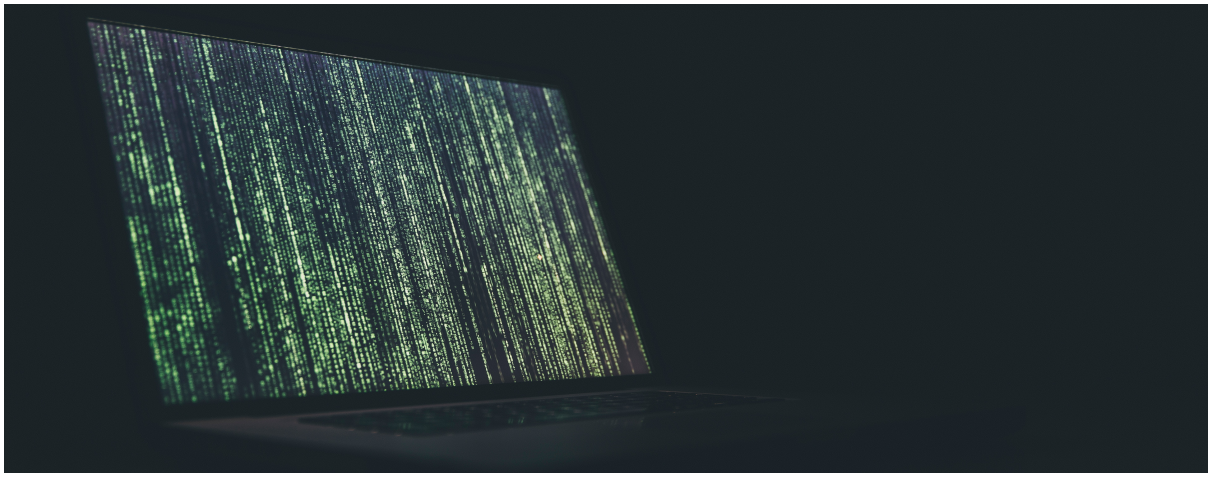


CSE321 - Introduction to Algorithm Design - HW05

Name Surname	Muhammed Oğuz
Number	1801042634

Table Of Contents

General Explanation About Python

Typing Information

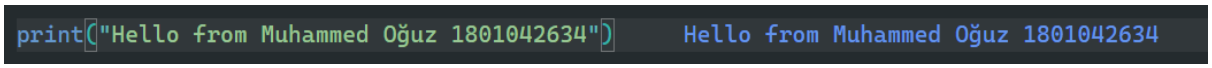
I use my python with typing module. It is a hobbit from my Java and C++ knowledge. So you will see an import and type notations.

```
from typing import List # a typing library from standart python3.8
def foo(variable : int) -> int: # I specify variable type and return type like this
```

Test Result Information

Additionally, I use an extension that directly shows output of the results in VSCode. It is called `Wolf`.

Here is an example



Left Side is the code and right side with blue color is the output. With help of that, I can show my input and output directly in one screenshot.

VS Marketplace Link: https://marketplace.visualstudio.com/items?itemName=traBpUkciP.wolf

First Problem - Max Profit Cluster

Problem Approach

This is the same problem at HW03. I already understand the problem while doing this. But for this homework I have to find a solution with dynamic programming. So I changed my old function to behave smarter. Now it is more efficient.

Solution and Analysis

```
def max_profit(prices : List[int]) -> int:
    max_profit = -math.inf                          # O(1)
    cur_profit = -math.inf                          # O(1)
    for price in prices:                            # O(n)
        cur_profit = max(price, cur_profit + price) # O(1)
        max_profit = max(max_profit, cur_profit)    # O(1)

    return int(max_profit)                          # O(1)
```

Complexity: $\theta(n)$

Running Results

```
# test
print(max_profit([3, -5, 2, 11, -8, 9, -5]))     14
print(max_profit([1, 2, 3, 4, 5]))     15
print(max_profit([7, 6, 4, 3, 1]))     21
print(max_profit([7, 1, -10, 3, 6, 4]))     13
```

Compare with Older Implementation

My older implementation was a brute force approach. It had two loops. So it was $\theta(n^2)$ complexity and has more line. But this is faster and more readable.

Second Problem - Candy Shop

Problem Approach

This one could be hard if we don't see exact same thing in lectures. It is basically a knapsack problem. So I implement my own knapsack algorithm for this problem.

I first create a matrix to hold all values. Purpose of this step is making our solution dynamic. After that, I calculate each weight and insert to the matrix. If matrix has already the answer, I used.

Solution and Analysis

```
def candy_metric(prices : List[int], lengths : List[int], target : int) -> int:
    # create a 2d Matrix
    matrix = [[0] * (target + 1)] * (len(lengths) + 1)

    # fill the matrix
    for i in range(1, len(lengths) + 1):                                    # O(n)
        for j in range(1, target + 1):                                      # O(m)
            if lengths[i - 1] <= j:                                         # O(1)
                matrix[i][j] = max(matrix[i - 1][j], matrix[i - 1][j - lengths[i - 1]] + prices[i - 1])
            else:
                matrix[i][j] = matrix[i - 1][j]                             # O(1)

    # return the maximum value
    return matrix[len(lengths)][target]
```

Complexity: $\theta(n^2)$

Running Results

```
# Test
# ———————————————— values ———————————————— weights ———————— W
print(candy_metric([1, 5, 8, 9, 10, 17, 17, 20], [1, 2, 3, 4, 5, 6, 7, 8], 8))     22
print(candy_metric([1, 5, 8, 9, 10, 17, 17, 20], [1, 2, 3, 4, 5, 6, 7, 8], 15))     42
print(candy_metric([1, 5, 7, 4, 1, 5, 14, 12], [1, 3, 4, 5, 5, 6, 7, 8], 10))     19
```

Third Problem - Cheese 🧀

Problem Approach

This one is also a knapsack problem. But It wants a solution with greedy way. I am not sure I am right but here is my greeydy approach to knapsack problem.

Solution and Analysis

```
def cheese_metric(p : List[int], w : List[int], W : int) -> int:
    n = len(p)                                          # O(1)
    if n == 0:                                          # O(1)
        return 0                                        # O(1)
    if W == 0:                                          # O(1)
        return 0                                        # O(1)
    if w[0] > W:                                        # O(1)
        return cheese_metric(p[1:], w[1:], W)           # O(1)
    else:
        return max(p[0] + cheese_metric(p[1:], w[1:], W - w[0]), cheese_metric(p[1:], w[1:], W))    # O(1)
```

Complexity: $\theta(1)$

Running Results

Since this one is also a knapsack algorithm, I use same inputs as second problem.

```
print(cheese_metric([1, 5, 8, 9, 10, 17, 17, 20], [1, 2, 3, 4, 5, 6, 7, 8], 8))     22
print(cheese_metric([1, 5, 8, 9, 10, 17, 17, 20], [1, 2, 3, 4, 5, 6, 7, 8], 15))     39
print(cheese_metric([1, 5, 7, 4, 1, 5, 14, 12], [1, 3, 4, 5, 5, 6, 7, 8], 10))     19
```

Fourth Problem - Courses

Problem Approach

This problem has very different approaches. But It is called Interval Scheduling Problem. I tried several implementations. But I think my implementation is not satisfies a greedy approach.

First, I sort evet course due to their end time. After that I compare their start date.

Solution and Analysis

```
# sort 2d array by column
def sort_by_column(arr : List[List[int]]):
    return sorted(arr, key=lambda x: x[1])  # O(n^2)

def max_courses(courses : List[List[int]]) -> int:
    sortedCourses = sort_by_column(courses)         # O(n^2)

    max_count = 0
    end_time = -math.inf

    for course in sortedCourses:                    # O(n)
```

```
        if course[0] >= end_time:                   # O(1)
            max_count += 1
            end_time = course[1]

    return max_count
```

Complexity: $\theta(n^2)$ Because of sorting ☹️

Running Results

```
# Test
print(max_courses([[1, 2], [3, 4], [0, 6], [5, 7], [8, 9], [5, 9]]))     4
print(max_courses([[3, 4], [1, 2], [4, 8], [5, 9], [9, 10]]))     4
```

Have a nice day Sir!