HW3 Solutions

1.

(a) Consider the sequence of weights 2, 3, 2. The greedy algorithm will pick the middle node, while the maximum weight independent set consists of the first and third.

(b) Consider the sequence of weights 3, 1, 2, 3. The given algorithm will pick the first and third nodes, while the maximum weight independent set consists of the first and fourth.

(c) Let S_i denote an independent set on $\{v_1, \ldots, v_i\}$, and let X_i denote its weight. Define $X_0 = 0$ and note that $X_1 = w_1$. Now, for $i > 1$, either v_i belongs to S_i or it doesn't. In the first case, we know that v_{i-1} cannot belong to S_i, and so $X_i = w_i + X_{i-2}$. In the second case, $X_i = X_{i-1}$. Thus we have the recurrence

$$X_i = \max(X_{i-1}, w_i + X_{i-2}).$$

We thus can compute the values of X_i, in increasing order from $i = 1$ to n. X_n is the value we want, and we can compute S_n by tracing back through the computations of the *max* operator. Since we spend constant time per iteration, over n iterations, the total running time is $O(n)$.

2.

Use dynamic programming

```
def planSchedule(lowStress, highStress):
    nWeeks = len(highStress)
    OPT = [0]*nWeeks
    OPT[0] = max(lowStress[0], highStress[0])
    OPT[1] = max(OPT[0] + lowStress[1], highStress[1])

    for i in range(2, nWeeks):
            OPT[i] = max(OPT[i-1] + lowStress[i], OPT[i-2] + highStress[i])

    maxRevenue = OPT[nWeeks-1];

    return maxRevenue
```

3.

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [float('inf')] * (amount + 1)
        dp[0] = 0

        for coin in coins:
            for x in range(coin, amount + 1):
                dp[x] = min(dp[x], dp[x - coin] + 1)
        return dp[amount] if dp[amount] != float('inf') else -1
```

Time complexity : $O(S * n)$.

On each step the algorithm finds the next $F(i)$ in n iterations, where $1 \leq i \leq S$. Therefore in total the iterations are $S * n$.

4.

The key observation to make in this problem is that if the segmentation $y_1 y_2 \dots y_n$ is an optimal one for the string y, then the segmentation $y_1 y_2 \dots y_{n-1}$ would be an optimal segmentation for the prefix of y that excludes y_n (because otherwise we could substitute the optimal solution for the prefix in the original problem and get a better solution).

Given this observation, we design the subproblems as follows. Let $Opt(i)$ be the score of the best segmentation of the prefix consisting of the first i characters of y. We claim that the recurrence

$$Opt(i) = \max_{j \le i} \{Opt(j-1) + Quality(j \dots n)\}$$

would give us the correct optimal segmentation (where $Quality(\alpha \dots \beta)$ means the quality of the word that is formed by the characters starting from position α and ending in position β). Notice that the desired solution is $Opt(n)$.

We prove the correctness of the above formula by induction on the index i. The base case is trivial, since there is only one word with one letter.

For the inductive step, assume that we know that the Opt function as written above finds the optimum solution for the indices less than i, and we want to show that the value $Opt(i)$ is the optimum cost of any segmentation for the prefix of y up to the i-th character. We consider the last word in the optimal segmentation of this prefix. Let's assume it starts at index $j \le i$. Then according to our key observation above, the prefix containing only the first $j-1$ characters must also be optimal. But according to our induction hypothesis, $Opt(j)$ will yield us the value of the aforementioned optimal segmentation. Therefore the optimal cost $Opt(i)$ would be equal to $Opt(j)$ plus the cost of the last word.

But notice that our above recurrence exactly does this calculation for each possibility of the last word. Therefore our recurrence will correctly find the cost of the optimal segmentation.

As for the running time, a simple implementation (direct evaluation of the above formula starting at index 1 until n, where n is the number of characters in the input string) will yield a quadratic algorithm.

5.

If $w < w_i$ *then* $\text{OPT}(i, w) = \text{OPT}(i-1, w)$. *Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), \text{v}_\text{i} + \text{OPT}(i-1, w-w_i)).$$

Use the recurrence relation above in the pseudocode of the subset sum problem in class slides.