DSC 190 - Homework 06

Due: Monday, November 13

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 PM.

Programming Problem 1.

In a file named `make_change.py`, create a function `make_change(t)` which computes the **number** of ways of combining American quarters (worth 25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent) to make t cents.

It may be possible that there is no way to make change for the target, t, in which case your function should return 0. You should assume that your are very rich and have an unlimited number of each type of coin.

For example: `make_change(11)` should be 4, since we can either use:

- 11 pennies,
- 1 nickel and 6 pennies, or
- 1 dime and 1 penny, or
- 2 nickels and 1 penny.

There is starter code on the course page, and you should submit your solution to the autograder.

Note: there is no target time complexity for this problem, but the autograder will time out if your solution takes more than 60 seconds to run. However, as long as you're not implementing a brute force solution, you should be fine.

Solution:

```
# the value of a quarter, dime, nickel, penny
VALUES = (25, 10, 5, 1)


# BEGIN REMOVE
def _make_change_helper(t: int, coin_ix=0):
    """Computes the number of ways to make change using only VALUES[coin_ix:]"""
    if t == 0:
        return 1

    if coin_ix >= len(VALUES):
        return 0

    if t < 0:
        return 0

    # either we use coin coin_ix at least once
    n_with_coin_ix = _make_change_helper(t - VALUES[coin_ix], coin_ix)

    # or we don't
    n_without_coin_ix = _make_change_helper(t, coin_ix + 1)
```

```
    return n_with_coin_ix + n_without_coin_ix


# END REMOVE
def make_change(t: int):
    """Count the number of ways to make change.

    Assumes an unlimited number of quarters, dimes, nickels, pennies.

    Parameters
    ----------
    t: int
        The total number of cents.

    Example
    -------
    >>> make_change(5) # 5 pennies or 1 nickel
    2
    >>> # (1 dime + 1 penny) or (2 nickels + 1 penny) or (11 pennies)
    >>> # or (6 pennies + 1 nickel)
    >>> make_change(11)
    4

    """
    # BEGIN PROMPT
    return _make_change_helper(t, 0)
    # END PROMPT
```

Programming Problem 2.

Consider the same problem of computing the number of ways to make change, but now we are not rich – we only have a certain number of quarters, dimes, nickels, and pennies.

In `constrained_make_change.py`, write a function named `constrained_make_change(t, coins)`, where t is the target sum that we are trying to reach, and `coins` is a list of four elements containing the number of quarters, dimes, nickels, and pennies we have. The function should return then number of ways of adding up to t using only the coins we have.

For example, `constrained_make_change(25, [2, 0, 0, 25])` should return 2, since there are only two ways with the coins we have:

- 1 quarter, or
- 25 pennies.

There is starter code on the course page, and you should submit your solution to the autograder.

Note: there is no target time complexity for this problem, but the autograder will time out if your solution takes more than 60 seconds to run. However, as long as you're not implementing a brute force solution, you should be fine.

Solution:

```
VALUES = (25, 10, 5, 1)
```

```
# BEGIN REMOVE
def _get_a_coin(coins):
    """Returns the first coin we have available, or None if we're out of coins."""
    for i in range(4):
        if coins[i] > 0:
            return i

    return None

# END REMOVE
def constrained_make_change(t, coins):
    """Compute the number of ways to make change, given a set number of each coin.

    Examples
    --------
    >>> constrained_make_change(11, [1, 1, 2, 5])
    2
    >>> constrained_make_change(11, [99, 99, 99, 99])
    4
    >>> constrained_make_change(25, [2, 0, 0, 25])
    2

    """
    # BEGIN PROMPT
    if t == 0:
        return 1

    if t < 0:
        return 0

    coin = _get_a_coin(coins)

    if coin is None:
        return 0

    original_number = coins[coin]

    # either we use this coin at least once
    coins[coin] -= 1
    n_with_coin = constrained_make_change(t - VALUES[coin], coins)

    # or we don't use it at all
    coins[coin] = 0
    n_without_coin = constrained_make_change(t, coins)

    coins[coin] = original_number

    return n_with_coin + n_without_coin
    # END PROMPT
```

Problem 1.

Consider the following set of activities.

Activity #	Start	Finish	Weight
0	0	3	12
1	1	2.5	10
2	2	3	22
3	2.75	4.25	14
4	4	8	33
5	4.5	11.5	11
6	5	6	12
7	5.5	6.5	7
8	8	10	5
9	9	12	19

a) Suppose the dynamic programming solution to the weighted activity selection problem is run on this data, and let `cache` be an array storing the memoized solution to each subproblem. In particular, `cache[i]` is the weight of the max weight schedule considering only the activities $i, i+1, \ldots, 9$. Note that we will consider two activities `x` and `y` to be compatible if `y.start` $\geq$ `x.finish` or `x.start` $\geq$ `y.finish`.

What are the entries of `cache`? You do not need to include the "dummy" entry of zero at the end of the array.

Hint: when checking your work, you should ask yourself: is it possible that `cache[i] < cache[i+1]`?

Solution:

```
[74, 74, 74, 52, 52, 31, 31, 26, 19, 19]
```

This can be obtained by essentially running the bottom-up algorithm by hand.

b) What is the weight of the optimal solution to the weighted activity selection problem for this data?

Hint: the right answer is greater than 70, less than 80.

Solution: 74

Programming Problem 3.

In lecture, we designed a dynamic programming solution for the weighted activity scheduling problem. Our solution computed the *weight* of the best possible schedule, but did not return the schedule itself. Luckily, an optimal schedule can be computed using the `cache` array as long as we know for each event i, the *next* event j which begins on or after i's finish time.

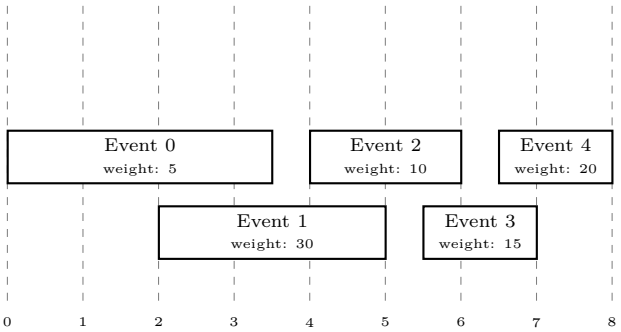
In a file named `recover_schedule.py`, write a function `recover_schedule(cache, next_activity)` which accepts two arguments:

- `cache`: The cache array as computed by the dynamic programming solution. It will be a Python list of numbers, where `cache[i]` is the weight of the best schedule that can be made from activities $i, i+1, \ldots, n-1$. `cache` *will* have one dummy entry of zero at the end, so that `len(cache)` is $n+1$, where n is the number of activities.
- `next_activity`: A list of n integers where `next_activity[i]` is the first activity starting after activity i finishes. We assume that the first activity is numbered zero, and that activities are sorted in order of start time. If there is no next activity possible, this entry should equal n.

Your function should return a list of integers representing the events that are in the optimal schedule. For instance, the list `[0, 2, 3]` represents the schedule consisting of the events 0, 2, and 3. There may be more than one optimal schedule; you need only return one of them.

Example: Consider the instance of the weighted scheduling problem shown below:

Activity #	Start	Finish	Weight
0	0	3.5	5
1	2	5	30
2	4	6	10
3	5.5	7	15
4	6.5	8	20



The best solution is to do activities 1 and 4, for a total weight of 50 (check for yourself!)

The dynamic programming `cache` will contain `[50, 50, 30, 20, 20, 0]`, where `cache[i]` represents the best solution using only activities from the set $\{i, i+1, \ldots, n-1\}$. The last entry of zero is the dummy entry mentioned above.

The `next_activity` list contains `[2, 3, 4, 5, 5]`. The 5s at the end denote that there is no possible next activity for Events 3 and 4.

Therefore, your code should behave as follows:

```
>>> cache = [50, 50, 30, 20, 20, 0]
>>> next_activity = [2, 3, 4, 5, 5]
>>> recover_schedule(cache, next_activity)
[1, 4]
```

Note that the code does not need to know anything about the actual activities, such as their weight – it only needs to know what the *next* activity is.

Hint: the hard part of this problem isn't writing the code – it is determining how to go from the `cache` to a schedule. Before writing any code, try to figure out the right strategy using a small example.

Solution: Let's say `cache[i] == cache[i + 1]`. What this means is that the weight of the best schedule possible using activities $\{i+1, \ldots, n-1\}$ is the same as the weight of the best schedule possible using events $\{i, \ldots, n-1\}$. In other words, including event i makes no difference. If event i were in the optimal schedule, there would be a difference.

Therefore, we loop through the cache. If we see `cache[i] > cache[i+1]`, then event `i` is in our optimal schedule. But we don't need to continue looping to `i + 1`, `i + 1`, etc. We know that the next event that can possibly be in our optimal schedule is `next_activity[i]`, so we skip ahead to it. In fact, if we don't skip ahead, we're likely to add another event to the schedule that should not be added.

```
def recover_schedule(cache, next_activity):
    """Return the indices of the activities that should be selected.

    You may assume that the activities are labeled in increasing order of their
    start time. In the case that more than one schedule is optimal, return an
    arbitrary one.

    Parameters
    ----------
    cache : List[float]
        A list such that cache[i] is the largest total weight of any valid
        schedule that consists of elements from the set {i, i+1, i+2, ..., n-1}.
```

```
    If there are n activities, `cache` should have n + 1 elements, with the last
    element being a dummy element equal to zero.
next_activity : List[int]
    A list of integers such that next_activity[i] is the index of the next
    activity whose start time is >= activity i's end time.

Returns
-------
List[int]
    A list of the activities containing in an optimal schedule.

Example
-------
>>> cache = [50, 50, 30, 20, 20, 0]
>>> next_activity = [2, 3, 4, 5, 5]
>>> recover_schedule(cache, next_activity)
[1, 4]

"""
# BEGIN PROMPT
n = len(next_activity)
schedule = []
i = 0
while i < n:
    if cache[i] > cache[i + 1]:
        schedule.append(i)
        i = next_activity[i]
    else:
        i += 1
return schedule
# END PROMPT
```