

Solution Review: Weighted Scheduling Problem

In this lesson we will solve the weighted scheduling problem with different techniques of dynamic programming.

We'll cover the following

- Solution 1: Simple recursion
 - Explanation
 - Time complexity
- Solution 2: Top-down dynamic programming
 - Optimal substructure
 - Overlapping subproblems
 - Explanation
 - Time and space complexity
- Solution 3: Bottom-up dynamic programming
 - Explanation
 - Time and space complexity

Solution 1: Simple recursion

```
# Given the index of the class and the list of schedule, this function returns the last class that
def lastNonConflict(index, schedule, isSorted = False):
    if not isSorted:
        schedule = sorted(schedule, key=lambda tup: tup[1])
    for i in range(index, -1, -1):
        if schedule[index][0] >= schedule[i][1]:
            return i
    return None

def WSrecursive(schedule, n):
    if n == None or n < 0: # base case of conflict with the first event
        return 0
    if n == 0: # base case of no conflict with the first event
        return schedule[n][2]

    # find max of keeping the n-th event or not keeping it
    return max(schedule[n][2] + WSrecursive(schedule, lastNonConflict(n, schedule, isSorted= True)),
               WSrecursive(schedule, n-1))

def WeightedSchedule(schedule):
```

```
# sort the schedule by end time of events
schedule = sorted(schedule, key=lambda tup: tup[1])
return WSrecursive(schedule, len(schedule)-1)

print(WeightedSchedule([(0, 2, 25), (1, 6, 40), (6, 9, 170), (3, 8, 220)]))
```



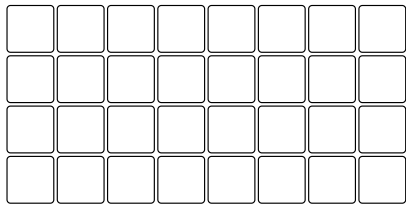
Explanation

To find the profit-maximizing schedule, we need to check all possible schedules. How do we find all possible schedules? This is pretty simple given the fact that you have a helper function, `lastNonConflict`, that gives us the last event that did not clash with the event provided to it. We sort our schedule with the end times of all the events (*line 21*). Next, we start from the last event in our sorted schedule, i.e., the event that ended the last. Now if this event is part of the optimal schedule, all the other events that clash with it cannot be part of the optimal schedule. Thus, we find the last non-conflicting event and make a recursive call with it (*line 17*). Another option could be that this event is not part of the optimal event, so we recursively make a call to the event before it (*line 18*). In the end, we take the max of these two calls and return it (*lines 17-18*).

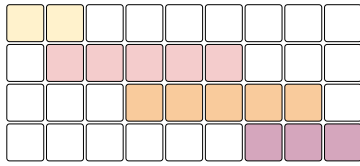
As always, let's take a look at a visualization of the dry run of this algorithm.

WeightedSchedule((0, 2, 25), (1, 6, 40), (6, 9, 170), (3, 8, 220))

Let's build a Gantt chart out of this schedule sorted by the end time

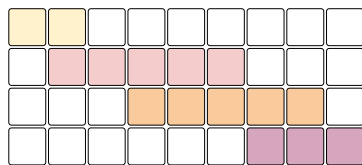


Events are listed along the vertical axis, while time runs along horizontal axis



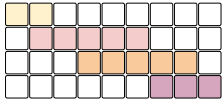
Yellow	25
Pink	40
Orange	220
Purple	170

Each square on Gantt chart represent 1 unit of time, right table denotes the total reward for each event

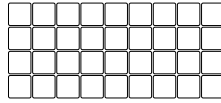


25
40
220
170

A schedule is valid if it does not have any event with overlapping time, this schedule for example is not valid



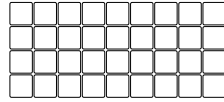
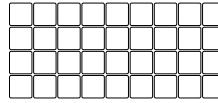
25
40
220
170



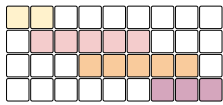
Let's start off from the last event i.e. the 4th one. Arrow denotes the event we are currently on



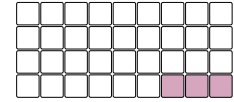
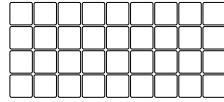
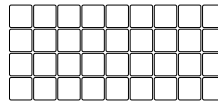
25
40
220
170



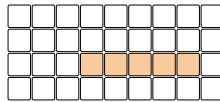
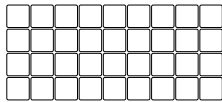
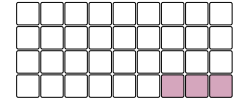
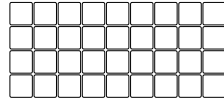
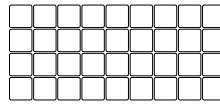
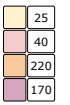
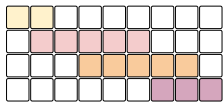
Now either the 4th event can not be a part of the optimal schedule, in which case we simply move to 3rd event



	25
	40
	220
	170



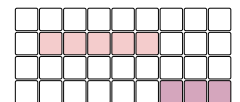
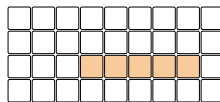
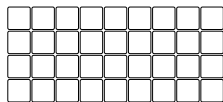
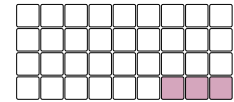
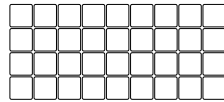
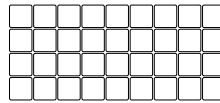
Or it can be a part of the schedule, in which case we will move to 2nd event because 3rd event clashes with the 4th one so it cannot be in same schedule



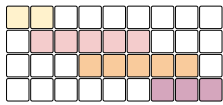
Similarly solving further, we can either have 3rd event in optimal schedule or we cannot



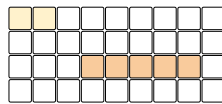
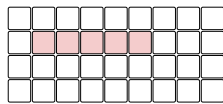
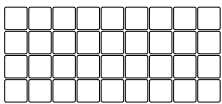
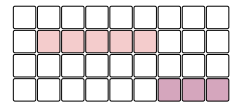
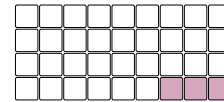
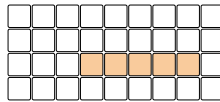
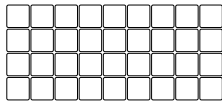
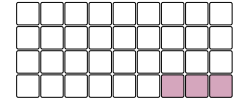
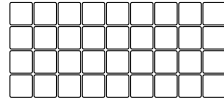
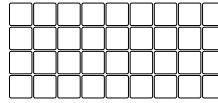
25
40
220
170



Solving the other recursive call, we can either have 2nd event in schedule or we cannot



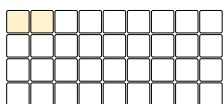
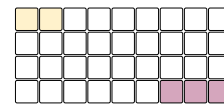
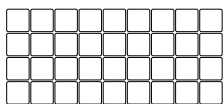
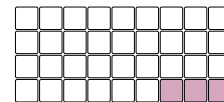
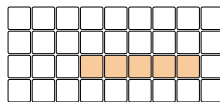
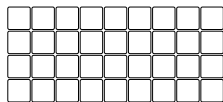
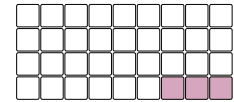
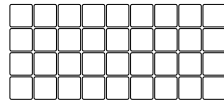
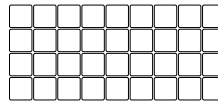
	25
	40
	220
	170



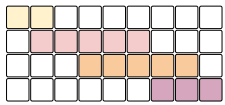
Going down one level deeper, we get following recursion tree, In a few cases we have no further option to proceed either because we have placed all events on schedule, or the other events clash with current ones



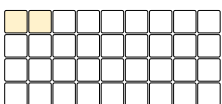
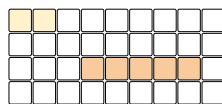
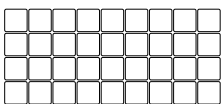
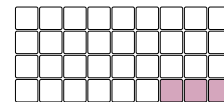
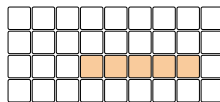
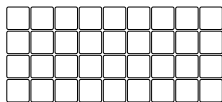
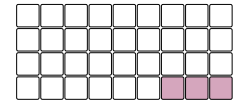
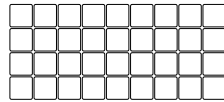
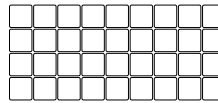
25
40
220
170



Evaluating the last call, we get following tree. Let's see the reward of each leaf i.e. a schedule



Yellow	25
Pink	40
Orange	220
Purple	170



Evaluating the last call, we get following tree. Let's see the reward of each leaf i.e. a schedule

Legend:

Yellow	25
Pink	40
Orange	220
Purple	170

Thus the maximum reward is 245

14 of 14



Time complexity

At each step, we have two possible options to explore. One is to include the event in the schedule, while the other is to exclude it. Thus, as we have already seen multiple times in this course, having n steps translates to a time complexity bounded by $O(2^n)$. Now, multiply with this the time complexity of finding the last non-conflicting event, i.e., $O(n)$. We have an overall time complexity of $O(n2^n)$.

Solution 2: Top-down dynamic programming

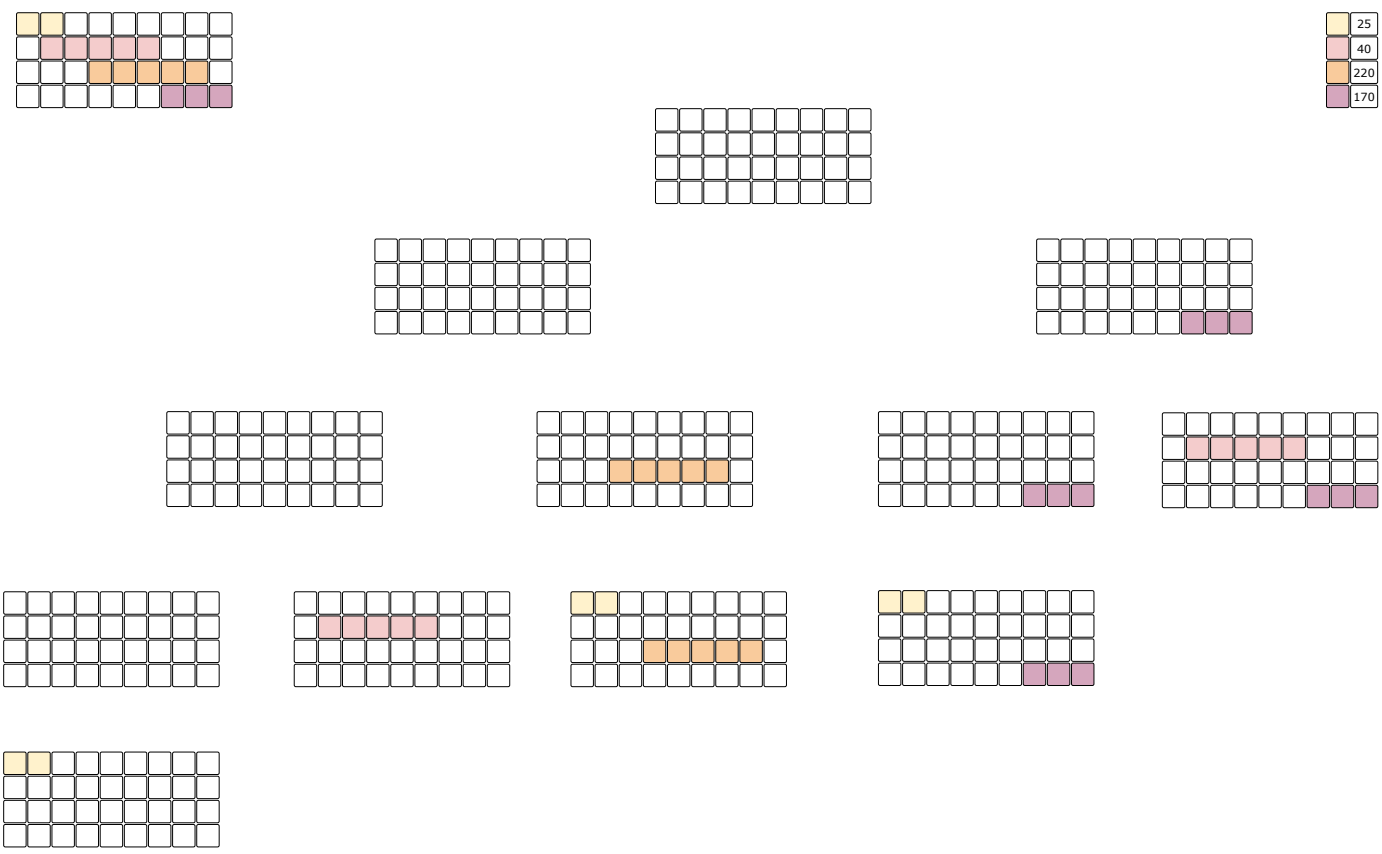
Let's look at how this problem satisfies both the properties required to apply dynamic programming to it.

Optimal substructure

Suppose we are solving the problem with n events and have solved all the subproblems less than n . We can see that the n^{th} subproblem requires the evaluation of at most two subproblems. One of these would most definitely be the $n-1^{th}$ subproblem. While depending on the clash of the n^{th} event, the other could be anywhere between the 0^{th} to $n-1^{th}$ event. So, since n^{th} problem's evaluation only depends on the smaller subproblems and nothing else; this problem satisfies the optimal substructure condition.

Overlapping subproblems

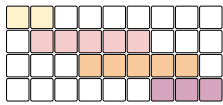
You could already see a number of overlapping subproblems in the above visualization. Below, we have highlighted them for you.



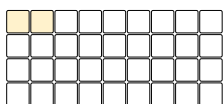
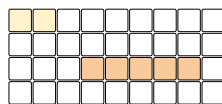
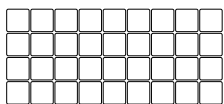
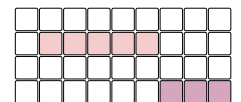
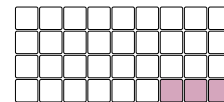
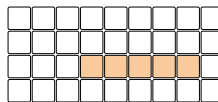
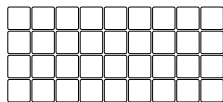
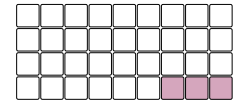
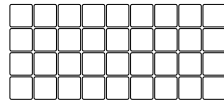
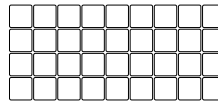
25
40
220
170

Let's try to find some overlapping subproblems

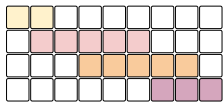
1 of 4



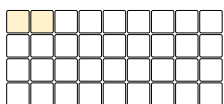
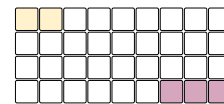
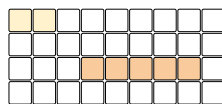
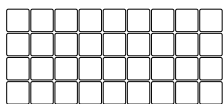
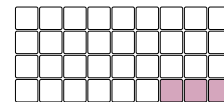
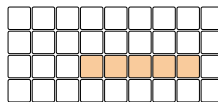
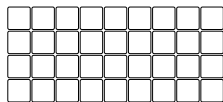
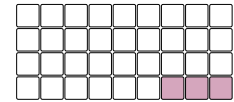
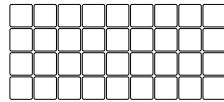
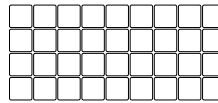
	25
	40
	220
	170



case for 2nd event has been reevaluated twice



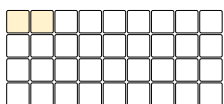
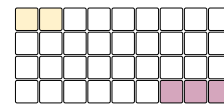
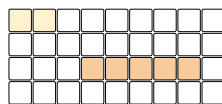
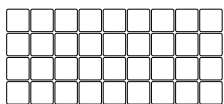
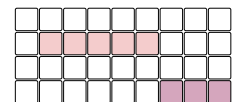
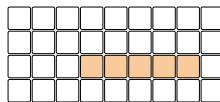
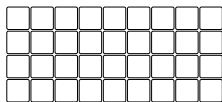
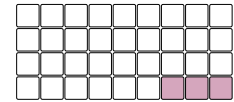
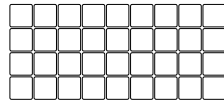
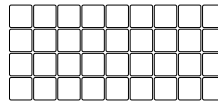
	25
	40
	220
	170



similarly, case for 1st event has been reevaluated multiple times



	25
	40
	220
	170



Have a look at redundant recalculations because of these calls

4 of 4

—

[]

```
# Given the index of the class and the list of schedule, this function returns the last class that
def lastConflict(index, schedule, isSorted = False):
    if not isSorted:
        schedule = sorted(schedule, key=lambda tup: tup[1])
    for i in range(index, -1, -1):
        if schedule[index][0] >= schedule[i][1]:
            return i
    return None

def WSrecursive(schedule, n, memo):
    if n == None or n < 0:
        return 0
    if n == 0:
        return schedule[n][2]
    if n in memo:
        return memo[n]
    memo[n] = max(schedule[n][2] + WSrecursive(schedule, lastConflict(n, schedule, isSorted= True),
        WSrecursive(schedule, n-1, memo))
    return memo[n]
```

```
def WeightedSchedule(schedule):
    schedule = sorted(schedule, key=lambda tup: tup[1])

    memo = {}
    return WSrecursive(schedule, len(schedule)-1, memo)

# make sure start and end of any event is not the same
print(WeightedSchedule([(0, 2, 25), (1, 5, 40), (6, 8, 170), (3, 7, 220)]))
schedule = [(i,i+2,10) for i in range(100)]
print(WeightedSchedule(schedule))
```



Explanation

Just like any other memoization based dynamic programming algorithm, the idea here is exactly the same. Look it up in the **memo** table before evaluating a subproblem and store the result in the **memo** table after evaluation to avoid recomputations.

Time and space complexity

Due to $O(1)$ lookups in the **memo** dictionary, we are not recomputing the same problems again and again. Thus, the time complexity to make n recursive calls would be $O(n)$. Since at every recursive call we also make an $O(n)$ lookup for the last non-conflicting event, our overall complexity is bounded by $O(n^2)$.

The space complexity would be $O(n)$ since we have n unique subproblems.

Solution 3: Bottom-up dynamic programming

```
# Given the index of the class and the list of schedule, this function returns the last class that
def lastConflict(index, schedule, isSorted = False):
    if not isSorted:
        schedule = sorted(schedule, key=lambda tup: tup[1])
    for i in range(index, -1, -1):
        if schedule[index][0] >= schedule[i][1]:
            return i
    return None

def WeightedSchedule(schedule):
    # sort the schedule by end times of events
    schedule = sorted(schedule, key=lambda tup: tup[1])
    dp = [0 for _ in range(len(schedule)+1)]

    for i in range(1, len(schedule)+1):
        # find the last conflicting event
        index_LC = lastConflict(i-1, schedule, isSorted=True)
        if index_LC == None:
            index_LC = -1
        # find the max of either keeping this event or not keeping it
```

```

        dp[i] = max(dp[i-1], dp[index_LC+1]+schedule[i-1][2])
    return dp[len(schedule)]

print(WeightedSchedule([(0, 2, 25), (1, 5, 40), (6, 8, 170), (3, 7, 220)]))
schedule = [(i,i+2,10) for i in range(100)]
print(WeightedSchedule(schedule))

```



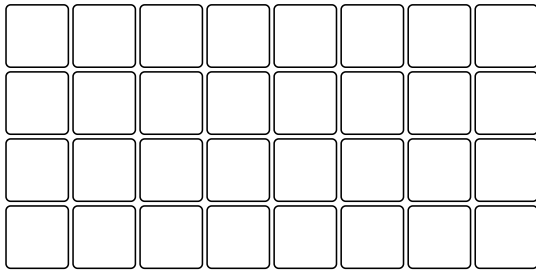
Explanation

We start filling our `dp` table from the first index. At each index, `i`, we require the answers to two problems we have already solved. The first one is simple, the answer to the previous subproblem stored in `dp[i-1]`. The other one is not complex either, it is the answer to the subproblem given by the last event with no conflict with the current problem's event. This way when we have completely filled our `dp` table, the answer will be at the last index.

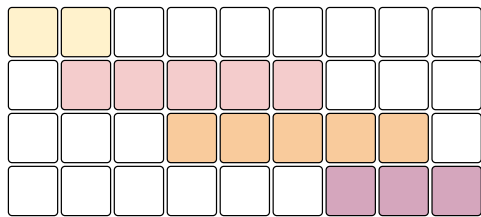
Here is a visualization of this algorithm's dry run.

WeightedSchedule([(0, 2, 25), (1, 5, 40), (6, 8, 170), (3, 7, 220)])

Let's build a Gantt chart out of this schedule sorted by the end time

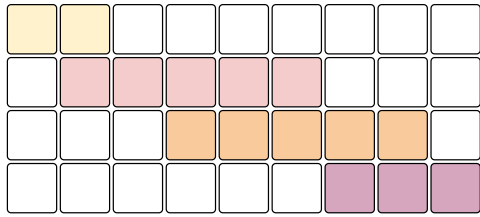


Events are lied along the vertical axis, while time runs along horizontal axis



Yellow	25
Pink	40
Orange	220
Purple	170

Each square on Gantt chart represent 1 unit of time, right table denotes the total reward for each event



	25
	40
	220
	170

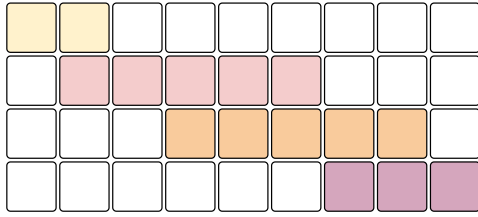
0	1	2	3	4
0				

Let's build a dp table of size 5, with index 0 set to 0

	25
	40
	220
	170

0	1	2	3	4
0				

Start filling the table starting from index 1



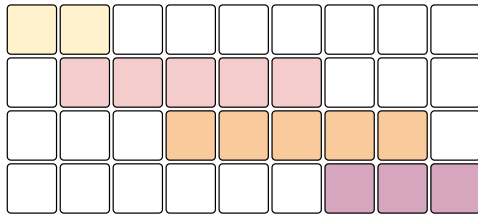
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0				

$$dp[1] = \max(dp[0], \text{profit}[0] + dp[0])$$

$$dp[1] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} , \right)$$

At each index we will choose maximum of these two options: 1- Schedule containing last event given here by $dp[0]$



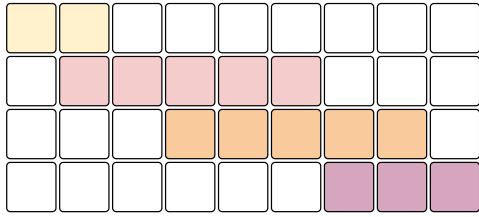
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0				

$$dp[1] = \max(dp[0], \text{profit}[0] + dp[0])$$

$$dp[1] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array}, \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} \right)$$

2- The schedule containing this event and the last non conflicting event, given here by profit[0] + dp[0]



Yellow	25
Red	40
Orange	220
Purple	170

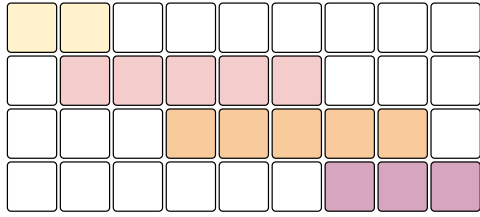
0	1	2	3	4
0	25			

$$dp[1] = \max(dp[0], \text{profit}[0] + dp[0])$$

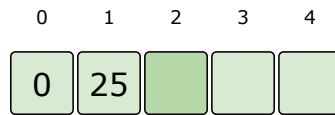
$$dp[1] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array}, \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} \right)$$

$$dp[1] = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array}$$

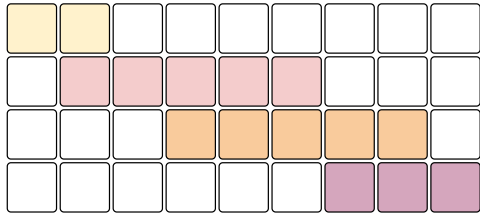
Since maximum profit is from the second term, $dp[1] = 25$



Yellow	25
Red	40
Orange	220
Purple	170



moving on to 2



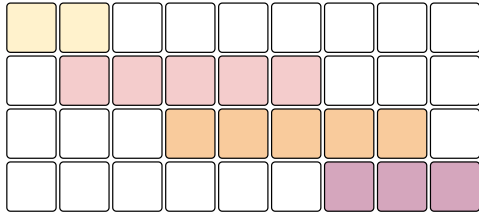
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0	25			

$$dp[2] = \max(dp[1], \text{profit}[1] + dp[0])$$

$$dp[1] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & \text{White} & \text{White} & \text{White} & \text{White} & \text{White} & \text{White} & \text{White} \\ \hline \text{White} & \text{Red} & \text{Red} & \text{Red} & \text{Red} & \text{Red} & \text{White} & \text{White} & \text{White} \\ \hline \text{White} & \text{White} & \text{White} & \text{Orange} & \text{Orange} & \text{Orange} & \text{Orange} & \text{Orange} & \text{White} \\ \hline \text{White} & \text{White} & \text{White} & \text{White} & \text{White} & \text{White} & \text{Purple} & \text{Purple} & \text{Purple} \\ \hline \end{array} , \right)$$

We will either have the last schedule given by $dp[1]$



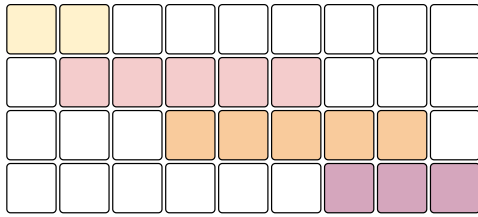
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0	25			

$$dp[2] = \max(dp[1], \text{profit}[1] + dp[0])$$

$$dp[2] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} , \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & & & & \\ \hline & \text{Red} & \text{Red} & \text{Red} & \text{Red} & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} \right)$$

or by keeping current event and the last non-conflicting event, which becomes $\text{profit}[1] + dp[0]$



Yellow	25
Red	40
Orange	220
Purple	170

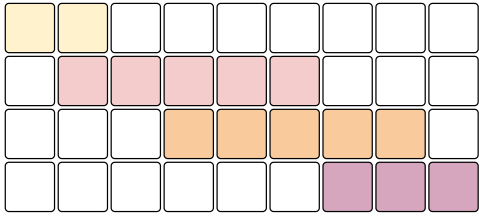
0	1	2	3	4
0	25	40		

$$dp[2] = \max(dp[1], \text{profit}[1] + dp[0])$$

$$dp[2] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array}, \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & & & & \\ \hline & \text{Red} & \text{Red} & \text{Red} & \text{Red} & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} \right)$$

$$dp[2] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & & & & \\ \hline & \text{Red} & \text{Red} & \text{Red} & \text{Red} & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array}$$

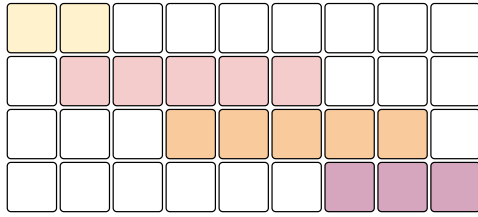
The max would be the second term, so $dp[2] = 40$



	25
	40
	220
	170

0	1	2	3	4
0	25	40		

moving on to 3



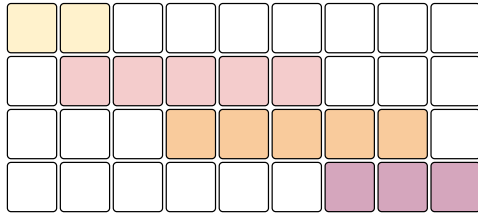
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0	25	40		

$$dp[3] = \max(dp[2], \text{profit}[2] + dp[1])$$

$$dp[3] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \text{Red} & \text{Red} & \text{Red} & \text{Red} & \text{Red} & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} , \begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \text{Orange} & \text{Orange} & \text{Orange} & \text{Orange} & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} \right)$$

We will either have the last schedule i.e $dp[2]$ or the schedule given by current event and last non conflicting schedule, which if you look at Gantt chart is $dp[1]$



Yellow	25
Red	40
Orange	220
Purple	170

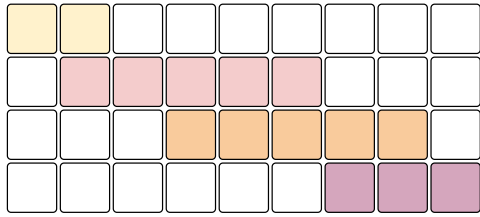
0	1	2	3	4
0	25	40	245	

$$dp[3] = \max(dp[2], \text{profit}[2] + dp[1])$$

$$dp[3] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \text{red} & \text{red} & \text{red} & \text{red} & \text{red} & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array}, \begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{yellow} & \text{yellow} & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \text{orange} & \text{orange} & \text{orange} & \text{orange} & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array} \right)$$

$$dp[3] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{yellow} & \text{yellow} & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \square & \square & \square & \text{orange} & \text{orange} & \text{orange} & \text{orange} & \square \\ \hline \square & \square & \square & \square & \square & \square & \square & \square \\ \hline \end{array}$$

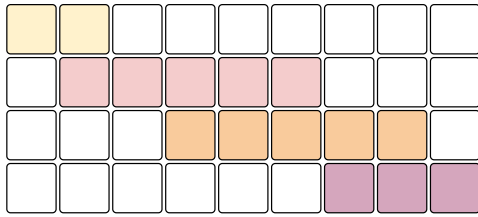
The max here would be second term so $dp[3] = 245$



Yellow	25
Pink	40
Orange	220
Purple	170

0	1	2	3	4
0	25	40	245	

moving on to 4



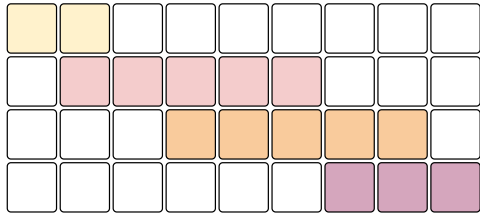
Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0	25	40	245	

$$dp[4] = \max(dp[3], \text{profit}[3] + dp[2])$$

$$dp[4] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} , \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & & & & \\ \hline & \text{Red} & \text{Red} & \text{Red} & \text{Red} & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & \text{Purple} & \text{Purple} & \text{Purple} \\ \hline \end{array} \right)$$

We will either have the last schedule i.e $dp[3]$ or the schedule given by current event and last non conflicting schedule, which if you look at Gantt chart is given by $dp[2]$



Yellow	25
Red	40
Orange	220
Purple	170

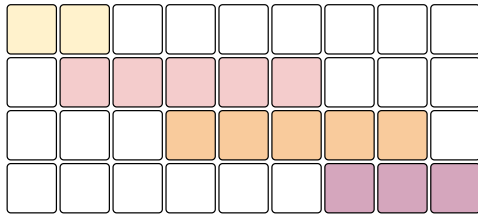
0	1	2	3	4
0	25	40	245	245

$$dp[4] = \max(dp[3], \text{profit}[3] + dp[2])$$

$$dp[4] = \max \left(\begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array}, \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} \right)$$

$$dp[4] = \begin{array}{|c|c|c|c|c|c|c|c|} \hline \text{Yellow} & \text{Yellow} & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array}$$

The max here would be the first term so $dp[4] = 245$



Yellow	25
Red	40
Orange	220
Purple	170

0	1	2	3	4
0	25	40	245	245

$$dp[4] = \max(dp[3], \text{profit}[3] + dp[2])$$

$$dp[4] = \max \left(\begin{array}{c} \text{Grid 1} \\ \text{Grid 2} \end{array} \right)$$

$$dp[4] = \begin{array}{c} \text{Grid 1} \\ \text{Grid 2} \end{array}$$

Thus the optimal schedule has a profit of 245

20 of 20

Time and space complexity

Again, the time complexity would be $O(n^2)$ because we need to evaluate all the n subproblems, in which the evaluation for calculating the last conflict would be $O(n)$.

Similarly, as the size of `dp` is n , space complexity would be $O(n)$. We can't do better than this because any problem can require a subproblem anywhere between 0 and n .

In the next lesson, we will go over another coding challenge on dynamic programming

programming.