**Name:** Debnath Kundu (MT22026) & Manvendra Kumar Nema (MT22038)

**DP** for solving the homework scheduling question **P:**

## 1. Specify a problem P:

Given three global arrays **TT, MM & DL** (each of size say **n**) which denotes Time Taken to complete an assignment, Marks obtains on completing that assignment, Deadline of that assignment respectively, the function **Scheduler( x, t )** gives the maximum score that can be ***possibly*** obtained by completing **1…x** homework on time, where **t** represents the timestamp initialized with the highest possible deadline.

Along with that, there are **Memoization** table and **Pointer** table of size **(highest deadline * n)**. [***Highest deadline*** = maximum among **DL[i]** ; ***1 <= i <= n*** ]

## 2. Give a recurrence expression/formula or recursive algorithm for solving P :

**Scheduler( x, t )** gives the maximum score that can be ***possibly*** obtained by completing **1…x** homework on time, where **t** represents the timestamp initialized with the highest possible deadline.

**Scheduler**( i , t ) = -1 if i < 0
**Scheduler**( i , t ) = argmax{
**Scheduler**(i–1, t),
**Scheduler**(i–1 , **argmin(t,DL[i]) - TT[i]** ) +**MM[ i ]**; if **( min(t,DL[i])** – **TT**[ i ] **)** >= 0
}

## 3. Prove correctness of recurrence relation.

In the given problem we have to find the maximum score possible while considering the deadline and time taken to solve each homework.
Assuming **Scheduler(x)** has returned a maximum score for the homeworks **1…x.**
Now let **k = x+1**.
Now **k**$^{th}$ Homework has two choices: it can be completed if there is sufficient time or it can be skipped.

- The **k**$^{th}$ Homework can be completed if there is sufficient time to complete it, which further lead to two cases:
  1. **t** is reduced by the time taken to do that homework ( **TT[k] ),** if deadline has not been crossed i.e. **t** will be updated as **t - TT[k]**
  2. If the deadline has been crossed, then we will try to fit homework into the last possible deadline, that is **DL[ k ]** is reduced by **TT[ k ]** i.e. **t** will be updated as **DL[ k ] - TT[ k ].**

  Now the time has been reduced to either of the above cases. The function **Scheduler** will consider the previously completed homeworks & fetch the maximum marks that

can be obtained in that reduced time. Specifically, it will fetch the maximum marks possible among **0…(k-1)** homeworks, say **PreviousMaximum**.
So, we add **MM[k]** to **PreviousMaximum** to get the maximum marks including this homework.

**Scheduler( k, t )** = **Scheduler**( k- 1 ,argmin (t , DL[ k ])- TT[ k ] ) + **MM[ i ]**
{ if min(t,D[ i ]) - TT[ i ] >= 0 }

- The Homework can be skipped in this case the score is equal to the score of the previously completed HomeWork.

**Scheduler**( k, t )=**Scheduler**( k - 1, t )

//OVERLAPPING SUBPROBLEMS

Building on the above intuition, an exhaustive search will be done on whether to complete or skip the Homeworks in order to find the maximum score if thought in terms of recursion.

The maximum of both cases will give the maximum score that can be obtained with or without completing that homework i.e. maximum score that can be ***possibly*** obtained by completing HomeWork **1…x**.

### 4. Describe a memoization data structure:

The **Memoization** table **M** is a **2-D** array of size **(n+1) * t**, and any index of **M** say **(i,j)** gives a maximum score obtained on a given timestamp **j**, when **i** numbers of homework are available.

The memoization table **M** is initialized as **M[0][1…t ]=0** and **M[0…n][ 0 ]=0.** The index **[n] [t]** will give the maximum score that can be obtained.

### 5. Give an algorithm/ordering for solving P for all values.

**Initialize.**
**Given**
**M[0][1…t ]=0 and M[0…n][ 0 ]=0**

```
for i in 1…t:
        for j in 1…n:
                a = M[ i - 1][t]
                b = -1
                if argmin( t, DL[ i ] ) - T[ i ] >= 0 :
                        b = M[ i - 1][ argmin( t, D[ i ] ) - TT[ i ] ] + MM[ j ]          //  completing
        that homework
        if a > b:
                M[ i ][ j ] = a      // filling Memoization array if element is excluded
        else:
                M[ i ][ j ]  = b
```

## 6. How to solve the original problem from memo

**M[n][t]** contains the maximum score that can be obtained in the given time.
The sequence that led to this score can also be retrieved through the below given algorithm. We should start retrieval from that index only, as it is the last filled cell in the memoization table.

## 7. What is space and time complexity for solving problems?

Since the above problem usage **2-D** array is used for **Memoization** of size
Space complexity - **O( (n+1)** * **t** ).

As filling the Memoization table requires traversing over all **(n+1) * t** elements of **M**.
So, the time complexity of filling the table would be **O( (n+1) * t)**.

To retrieve the sequence traversal over elements of the memoization table is required i.e traversing each row => **n+1** elements.

So Time complexity of retrieving sequence is **O(n)**
Total Time complexity = Time complexity of filling memoization and Pointer tables + To retrieve the sequence = **O((n+1) x t). + O(n)**
Total Time complexity = **O(n x t).**

## 8. How to obtain the optimal structure?

To compute the sequence with the maximum score, the Memoization table is used.
Since Index **[n] [t]** will give the maximum score that can be obtained, so in order to retrieve the sequence we will start from this index itself as:

```
i = n
j = t
print( M[ i ][ j ] )                          // Printing Maximum Marks
while  i !=0 :
        if M[ i ][ j ]  == M[ i - 1 ][ j ] :  // ith HomeWork is considered
                i=i-1
        else:
                print( “HW” , i)
                i =  i -1
                j = argmin( j , DL[ i ] ) - TT[ i ]
```

The remaining homeworks will be considered as late submissions, hence, no marks will be awarded for them.