Some conceptual ideas :
Using dynamic programming, we can first sort all the delivery D[1......n] by their due dates such that the first element of the array has the earliest due dates, any item with a ti > di is regarded as impossible to deliver.

For example, if the original delivery queue (by due date) is D[2, 6, 3, 4], with a corresponding ti of [2, 2, 2, 2].

And after we sorted the array D, it becomes [2, 3, 4, 6], and for a particular delivery i in the sorted array, it will take ti days to deliver, for example the second delivery, with a due date of 3, and a ti of 2, if we are to deliver this particular order, then any other item to be delivered within (di - ti +1, di) can not happen, for example, if we are to deliver second item which will take 2 days to deliver, then no delivery can be made on day 2 or day 3 to guarantee the delivery of the second item. So we need to find a date k such that dk < di - ti +1

Overall picture: we need to compute the max profit over the sorted array [1…….n]

Question1:

Pseudocode/algorithm:

Sort D by the due dates;
-Let dd represent the due dates in the sorted D, from 1 to T
-Let i represent the deliveries in the sorted D, from 1 to n
-Set ComputeProfit(n+1,dd) = 0; // set the last entry of the graph to be 0 for convenience

// first idea → use this

```
ComputeProfit(i, dd)
        -For i = 1 to n
           -For dd = 1 to T
              // the the time to deliver surpassed the due dates
              -if dd + ti > di
                 // move on to the next one
                 -return ComputeProfit( i+1, dd);
              -else
               // now we need to choose whether to include delivery i or not
                -let include = Pi + ComputeProfit (i+1, dd+ti); // include the delivery i
                - let exclude = ComputeProfit (i +1, dd); // exclude i
                -let max = max(include, exclude);
        -return ComputeProfit(1,1);
```

Recursive equation:

1. ComputeProfit(i, dd) = Max(Pi + ComputeProfit (i+1, dd+ti) → if dd+ti < di, ComputeProfit (i+1, dd));

Proof of correctness:

Using induction on i going from 1 to n+1

1) Base case: i = n+1, this is the case where the profit is 0 by the definition, therefore correct. Or if |D| = 1, there only exists a single delivery, and if d > t, we will correctly calculate the profit.
2) Induction hypothesis: the profit we compute for (i+1, dd) (in the delivery array D works correctly for any i
3) Inductive step:
   a) consider instance (i, dd) and let OPT be the optimal solution, that is, OPT is the max profit of the sorted array D with all the deliveries
   b) 1st possibility: OPT contains delivery i, then OPT\ { i } is also an OPT for the subarray D[i+1 ….. n], otherwise, OPT would not be an optimal solution for instance (i, dd), so ComputeProfit (i+1, dd+ti) = OPT\ { i } by I.H. and OPT = Pi + ComputeProfit (i+1, dd+ti).
   c) 2nd possibility: OPT doesn't contain delivery i, then OPT is the max profit of the subarray D[i+1 …..n], otherwise OPT would not be optimal for instance (i, dd), so OPT = ComputeProfit (i+1, dd)
   d) Therefore OPT is optimal.

Running Time analysis:

Running time = #subproblems * time per subprogram

RT = O(n*T) * O(1) = O(nT)

Question2:

If we put less emphasis on the profit and more emphasis on the market share

Algorithm:

-sort the deliveries by the ti, such that the last delivery in the array has the least delivery time

-Let k represents the number of days

-Let array D[i] represent the max number of deliveries made in the first i days

-Base case: At the first day, if d1 > t1, then D[1] = 1, else D[1] = 0
-pre-compute the largest k that exist in delivery poll, for every i such that dk < di - ti +1;
-if a delivery is made, then D++
-Set ComputeNumber(n+1) = 0; // set the last entry of the graph to be 0 for convenience

```
// working backwards → start from n+1 and work back to 1
computeNumber( i )
-for i = n+1 to 1
   -if  dd + ti > di
        -Return computeNumber( i + 1);
   -else
        // knowing k will help us leave plenty time for that particular delivery
        // and know where to start in the next recursive call
       -calculate the largest value k in the delivery poll such that dk < di - ti +1
       -let include = 1 + computeNumber( k );
       -let exclude = computeNumber( i - 1 );
       -let maxDelivery = max(include, exclude);
-return computerNumber (n);
```

Recursive Equation:
ComputerNumber (i) = Max (1 + computeNumber(k) , computeNumber(i -1))

Proof of correctness:
Using induction on i going from n+1 to 1

1) Base case: i = n+1, this is the case where the profit is 0 by the definition, therefore correct. Or if |D| = 1, there only exists a single delivery, and if d > t, we will correctly calculate the max number of deliveries = 1.
2) Induction hypothesis: the max number of deliveries we compute for (n) works correctly for any i
3) Inductive step:
    a) consider instance (i) and let OPT be the optimal solution, that is, OPT is the max number of deliveries of the sorted array with all the deliveries
    b) 1st possibility: OPT contains delivery i, then OPT \ { i } is also an OPT for the subarray Delivery[1…...k], otherwise, OPT would not be an optimal solution for instance (i, dd), so ComputeNumber (k) = OPT\ { i } by I.H. and OPT = 1 + ComputeNumber (k).
    c) 2nd possibility: OPT doesn't contain delivery i, then OPT is the max profit of the subarray Delivery[1…... i-1], otherwise OPT would not be optimal for instance (i, dd), so OPT = ComputeNumber (i-1)
    d) Therefore OPT is optimal.

Running Time:

RT = size of the delivery array * time per subproblem = O(n)