Q1) This is a greedy problem similar to job scheduling based only on time. Here we minimize the function which is based on both time to submit and penalty score.

Solving the problem in increasing order of t/p is the optimal solution
Proof: Let's say we have solved all problems and only two problems i and j are left. The total time taken till the last two problems is T and the penalty score is P. Let ti/pi < tj/pj.

If we solve problem i first then total penalty score,

$$P1 = P + (T+ti)*pi + (T + ti + tj)*pj$$
$$= P + T*(pi + pj) + ti*pi + tj*pj + ti*pj$$

If we solve problem j first hen total penalty score,

$$P2 = P + (T+tj)*pj + (T + ti + tj)*pi$$
$$= P + T*(pi + pj) + tj*pj + ti*pi + tj*pi$$

Now,

$$P1 - P2 = (P + T*(pi + pj) + ti*pi + tj*pj + ti*pj) - (P + T*(pi + pj) + tj*pj + ti*pi + tj*pi)$$
$$= ti*pj - tj*pi$$
$$= pi*pj(ti/pi - tj/pj)$$
$$< 0 \text{ (as } ti/pi < tj/pj)$$

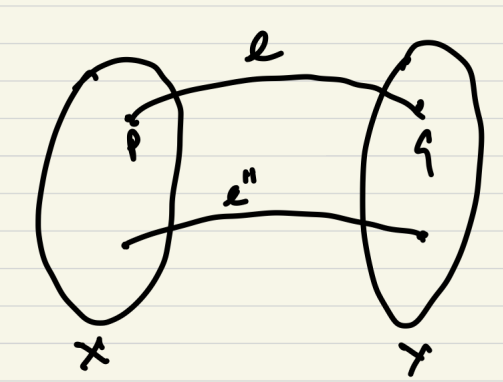
Hence, we should problem i first then solve problem j. Similarly to solve the N problems, solve the problem with the lowest t/p first as we can always swap the problem with one that has a lower t/p value to decrease the total penalty score.

Now, you can use any sorting algorithm with **O(NlogN)** time complexity to solve this problem.

Q2)

(a) In this part, we have to select roads such that each city is connected and we have to reduce the maximum value of F among these roads. Basically, we have to find a **minimum bottleneck spanning tree (MBST**) i.e a spanning tree in which the most expensive edge is as small as possible.

There is a property of MST that every MST is also MBST.
Proof: Let e be the maximum weighted edge in MST with weight w and e' be the maximum weighted edge in MBST with weight w'. Now w >= w' as e' is edge in MBST. If w > w' and edge e connects vertices p and q, then we can create cut b/w p and q.



Now since the graph is MST, then e is the only minimum weighted edge b/w p and q. However, we have MBST with a lesser weight than w. Now this is a contradiction as MBST is also a spanning tree and it must have an edge between these cuts let's say e'' with weight w'' <= w' which is less than w. So the assumption that w > w' is wrong and w = w'. Hence proved, that every MST is MBST.

Now we will use Kruskal's algorithm with dsu to find MST in O(MlogM + MlogN) time complexity.

Pseudo Code

```
E <- edges sorted in increasing order of F
Fmax = -Inf
For e in E:
        If parent(e.u) != parent(e.v):
                join(e.u, e.v)
                Fmax = max(Fmax,e.F)

Return Fmax
```

Time complexity: Sorting edges will take O(MlogM) time and the time complexity for dsu operations is O(logN) and we will do it for every edge hence it will take O(MlogN) time, Total time complexity is O(MlogM + MlogN)

(b) For this part, we have to find a spanning tree such that the sum of the maximum value of waiting time and fuel is minimum. For this problem, we transverse edges in increasing order of F and then find MBST on waiting time such that all edges have fuel value less than F. We will keep track of edges only in previous MBST as this graph includes all smallest edges that do not form cycle for given F and other remaining edges will never be part of MBST for next value of F. We will add upcoming edge to previous MBST and then find new MBST.

Pseudo Code

```
E <- edges sorted in increasing order of F
E' = E[0:N-2] (smallest N-2 edges)

ans = Inf
For i = N-2 to M:
        E'.push(E[i])
        E'' <- sort E' on increasing order of W
        T <- MBST on edges E'' using W as weight with maximum weight as Wmax
        If T has N-1 edges:
                ans = min(ans, E[i].F + Wmax)
        E' <- edges in T

return ans
```

Time Complexity: Sorting edges will take O(MlogM) time. We will run for loop on all edges and for each edge will sort the edges in E' and find MBST. Both MBST and sorting will take O(NlogN) time as E' will always have a size less than equal to N-1. So the total time complexity of for loop is O(M*NlogN). Hence, the total time complexity of the algorithm is O(MNlogN + MlogM)

Note: We can also solve this problem in O(MN + MlogM) time, basically we have to add only more edge to the previous MBST and find the new MBST. Add a new edge to the previous MBST, and if there is a cycle in the graph remove the maximum weighted edge based on W that forms the cycle to find a new MBST. We can use dfs to check for the cycle and remove the edge. Now time complexity inside the loop is reduced from O(NlogN) to O(N). Total time complexity will be now O(MN + MlogM).

Q3)

(a) This question is similar to multi-increment problem and report updated value stored at any index efficiently.
For each employee, increment values in the interval (entryTime, exitTime) by 1 using the multi_increment function. This takes O(MlogN)
Use report(i) to answer the Q queries which takes time Qlog(N)

Multi-Increment(i, j, Δ) efficiently

Sketch:

1. Let **u** and **v** be the leaf nodes corresponding to x_i and x_j.
2. Increment the value stored at **u** and **v**.
3. Keep repeating the following **step** as long as parent(**u**) <> parent(**v**)
   Move up by one step simultaneously from **u** and **v**
   - If **u** is **left child** of its parent, increment value stored in sibling of **u**.
   - If **v** is **right child** of its parent, increment value stored in sibling of **v**.

Executing Report(i) efficiently

Sketch:

1. Let **u** be the leaf nodes corresponding to x_i.
2. **val** ← **0**;
3. Keep moving up from **u** and keep adding the value of all the nodes on the path to the root to **val**.
4. Return **val**.

(b) Create an array A of size N+1 initialized with 0. For every employee add 1 to the time index at which he enters the office and subtract 1 at time index+1 he leaves the office.
Then we calculate the prefix sum using A[i] = A[i] + A[i-1].
Now A[Query[i]] is the answer for the ith query.

```
For i in 1 to M  #O(M)
        A[entryTime[i]] += 1
        A[exitTime[i]+1] -= 1
For i in 1 to N  #O(N)
        A[i] += A[i-1]
For i in 1 to Q  #O(Q)
        print(A[Query[i]])
```

There are three independent loops. takes time complexity O(M+N+Q)

Q4)

(a) Sort tuples (EmpTalents, EmpSkills) in descending order with the key as EmpTalent

```
For t,s in sorted_tuples: # O(M)
        Insert s in list so that ascending order is maintained #O(M)
        Sum += s
        If len(list) > k:
                Sum -= list.top() and pop it # This is the minimum element
        Ans = max(ans, t * Sum)
Return Ans
```

Time Complexity -> We loop through all employees and we are inserting elements in a list which again takes O(M). Hence $O(M^2)$.

(b) Here, the bottleneck is the insertion step that could be improved using a data structure like a minimum heap.
Instead of insertion, we push into a heap (Takes O(logK)) and pop out (Takes O(logK)).
Sorting step takes O(MlogM).
Time complexity is O(MlogM + MlogK) which is O(MlogM)