1. (a) A small example :

$L_1$ : 4
$L_2$ : 5
$L_3$ : 6
cars = [4, 5, 6]   n=3

Prof. I.M. Rong's greedy algorithm :
For the first car, which length is 4, park it in Lane 3 because it has the most remaining space, which is 6
For the second car, which length is 5, park it in Lane 2 because it has the most remaining space, which is 5
For the third car, which length is 6, there is no enough space to park this car in any of the three lanes
Parked 2 cars in total

A better solution :
Park the first car, which length is 4, in lane 1
Park the second car, which length is 5, in lane 2
Park the third car, which length is 6, in Lane 3
Parked 3 cars in total, which is better

(b) My DP table would require 4 dimensions :

$L_1$ : the length of Lane 1
$L_2$ : the length of Lane 2
$L_3$ : the length of Lane 3
$n$ : the number of cars

DP[$L_1$][$L_2$][$L_3$][$n$] :
the value of DP[$i$][$j$][$k$][$l$] means the most number of cars in the first $l$ cars that can be parked using length $i$ in Lane 1, length $j$ in Lane 2, length $k$ in Lane 3
our result is stored in DP[$L_1-1$][$L_2-1$][$L_3-1$][$n-1$].
my use-it or lose-it approach :
try one car at a time in order of the array.

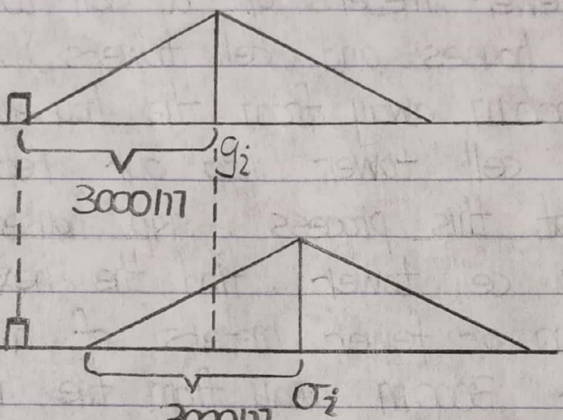
```
CarPark ($L_1$, $L_2$, $L_3$, ~~[illegible]~~ cars[n]) :
    find cars[0] , if no cars[0], we have tried all cars, return 0
    if cars[0] $\leq L_1$ :
        try CarPark($L_1$ - cars[0], $L_2$, $L_3$, cars[1:n]) + 1
    if cars[0] $\leq L_2$ :
        try CarPark($L_1$, $L_2$ - cars[0], $L_3$, cars[1:n]) + 1
    if cars[0] $\leq L_3$ :
        try CarPark($L_1$, $L_2$, $L_3$ - cars[0]. cars[1:n]) + 1

    if cars[0] > $L_1$ && cars[0] > $L_2$ && cars[0] > $L_3$
        try CarPark($L_1$, $L_2$, $L_3$, cars[1:n])
    find the most number of cars could be parked from all
    cases above, return it
```

2.(a) First we have an empty sorted array of positions of cell towers.
Start from the first element of the sorted array of positions of houses, which represents the first house's position from the left starting point of the road, and visit each element of the array in order. Find the first house that is not within 3000 or fewer meters of a cell tower (by comparing the position of houses and cell towers), and then set a cell tower 3000m away from the house on the right, save the position of cell tower into our returned array at back.
Then repeat this process, skip houses that are covered by at least a cell tower, find the next first house that is not within 3000 or fewer meters of a cell tower, set a new cell tower 3000m away from the house on the right, save its position into our returned array at back, until we have visited all houses in the array.
Since we visit houses in order (from left to right), the positions of cell towers in our returned array is naturally sorted (from left to right) because every time we add the position of a cell tower at the back of the array.

(b) In my greedy algorithm, I need to visit every house in the array in order to determine whether the house is within 3000 or fewer meters of a cell tower, which would help us to decide whether to set a new cell tower.
Therefore, the worst-case asymptotic runtime is O(n), n is the number of houses.

(c) consider an optimal solution $\sigma$ that differs from our greedy solution $g$
look at the first cell tower where they differ, the position of the cell tower is at $\sigma_i$ in the optimal solution and at $g_i$ in our greedy solution

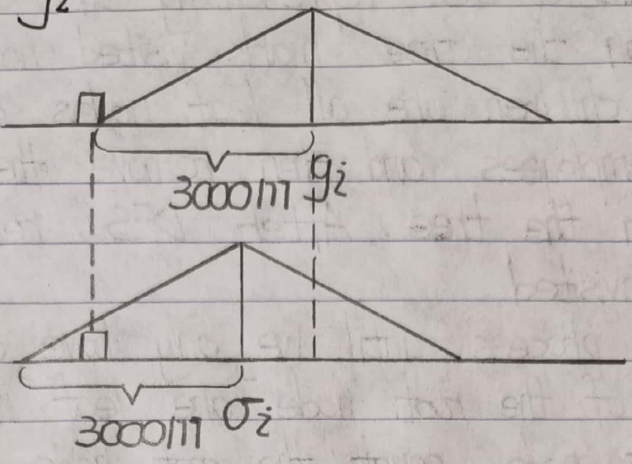
1° $\sigma_i > g_i$



This situation is invalid. Remember there is a house 3000m away from $g_i$ on the left, and it is only covered by the cell tower at $g_i$. If we choose to set the cell tower at $\sigma_i > g_i$, then the house is not within 3000 or fewer meters of any cell tower, which gives a wrong answer.

2° $\sigma_i$ ~~does not exist~~ does not exist
In this case, $g_i$ is our last cell tower and the optimal solution only has $i-1$ cell towers, one cell tower less than our greedy solution.
Same as Case 1°, this situation is invalid because the house 3000m away from $g_i$ on the left could not be covered by any cell towers in the optimal solution, which gives a wrong answer.

3° $\sigma_i < g_i$

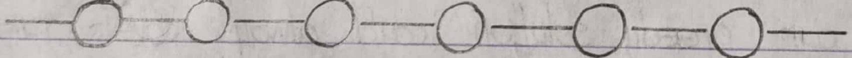


In this case, since $g_i > \sigma_i$, our greedy solution covers further distance than the optimal solution, which means that compared to the optimal solution, there are less or equal numbers of houses uncovered by a cell tower in our greedy solution. Therefore, it is impossible for the optimal solution to set fewer cell towers to cover those remaining uncovered houses, because our greedy solution could always cover further or equal distance.

In general, we have ~~pro~~ proved that our greedy algorithm correctly gives an optimal solution.

3.(a) First we have a tree representing the company hierarchy. Do DFS on the tree, mark visited nodes, find every node whose children are all leaf nodes, count all leaf nodes as invited employees, and then remove the node and its children from the tree. After DFS, remark all remaining nodes as unvisited.
Repeat this process until we only have a root node or all children of the root node are leaf nodes. If we only have a root node, count the root node as invited employees. Otherwise, count the children of the root node as invited employees.

(b) In the worst case, every node has at most one child, which is just like a linked list.



In our greedy algorithm, we would find nodes whose children are all leaf nodes in each DFS, count leaf nodes and remove those nodes with their children, and then do the next DFS. As we seen, we need to visit every node during each DFS to find target nodes.
In the worst case, we can only remove two nodes during each DFS, the only leaf node and its parent. Assume we have n nodes in total, we need to visit n nodes in the first DFS, n-2 nodes in the second DFS, reduce by 2 nodes after each DFS, until there only remains the root node and its possible existing child. In total, we need to do $\frac{n}{2}$ times of DFS.
So the worst-case runtime: $\frac{(n+1) \times \frac{n}{2}}{2} \in O(n^2)$

(c) Strong induction :

Base case :

If we only have one node in the tree, which is the root node, our greedy algorithm just count it as invited employees. Our greedy algorithm selects one node from one node, which gives an optimal solution.

Inductive Hypothesis :

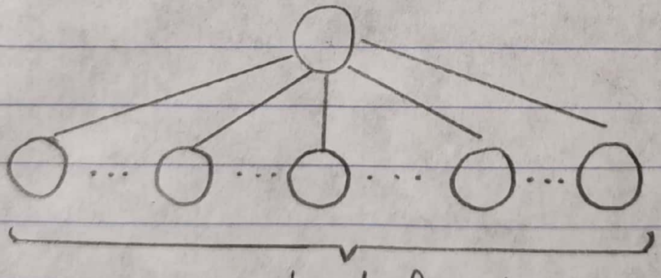
Assume that our greedy algorithm correctly selects an optimal solution for a tree of size $k$ or smaller.

Inductive Step :

If we have a tree of size $k+1$ :

1° a special case

If the tree has one root node with $k$ children that are all leaf nodes :



In this case, our greedy algorithm would count those $k$ leaf nodes as invited employees, which gives an optimal solution that selects $k$ nodes from $k+1$ nodes.

2° general case

Except from the special case, based on our greedy algorithm, during each DFS, we would find nodes whose children are all leaf nodes. Here, every target node with its children could be considered as a smaller tree, whose size is at most $k-1$.

According to the Inductive Hypothesis, when we have a tree of size k or smaller, our greedy algorithm would correctly select an optimal solution. Here, all the smaller trees come from target nodes have a size smaller than k, which means that our greedy algorithm would give an optimal solution in each of them. Since these trees would be removed from the original tree and are independent to each other, by summing up the optimal solutions of each smaller tree found during DFS, our greedy algorithm gives an ~~algorithm~~ optimal solution of the whole tree.

In other words, our greedy algorithm would break the original tree into several smaller trees and gives an optimal solution in each of them. Since these smaller trees are independent to each other, adding their optimal solutions together would give an optimal solution to the original tree of size k+1.