HW7

1. Greedy Ferries

a) Counterexample: say we have 3 cars with length a=1, b=4, c=10, and l1=3, l2=4, l3=12
According to Lai's algorithm, car a will be parked in lane 3, car b will be parked in lane 3, and car c will not be able to get onto the ferry. In contrast we can park a-l1, b-l2 and c-l3, or other ways as a-l3, b-l2, c-l3, etc. to ensure every car is on the ferry.

b) At first place, I guess it refers to the scheduling problem introduced in class. But it turns out I wrongly understood the problem, that I thought we only know the length of current car. In this case it's an '***online scheduling***' problem, which we can only deal with the current car we are facing and is a much more complex topic.
Another *greedy* idea that came to my mind is to park in lane with least possible space. But for l1=3, l2=4, l3=5 and a=3, b=3, c=4, d=2, this greed cannot effectively solve it.
Still, we can somehow tackle it with **recursive** calls. For example, we can put one car into each lane, which gives us 3 possibilities. And for each possibility, we have another 3 possibilities. Finally we find the one solution which holds the most cars and rewind to yield the result. But it'll soon lead to really deep recursion when there are more cars. The recursion above describe as follow:

```
1.  def max_car_recursive(x1, x2, x3, cars):
2.      """
3.      The recursive solution. Not optimal in asymptotic runtime
4.
5.      :param x1: remaining space for lane 1
6.      :param x2: remaining space for lane 2
7.      :param x3: remaining space for lane 3
8.      :param cars: the list of length of cars
9.      :return: the optimal choice for each step
10.     """
11.
12.     # base cases
13.     if x1 < 0 or x2 < 0 or x3 < 0:
14.         return 0  # no capacity
15.     if not cars:
16.         return 0  # cars all loaded
17.     if cars[0] > max(x1, x2, x3):
18.         return 0  # cannot hold the current car
19.
20.     next_cars = cars[1:]
21.     lane_1 = 1 + max_car_recursive(x1 - cars[0], x2, x3, next_cars)
22.     lane_2 = 1 + max_car_recursive(x1, x2 - cars[0], x3, next_cars)
23.     lane_3 = 1 + max_car_recursive(x1, x2, x3 - cars[0], next_cars)
24.     return max(lane_1, lane_2, lane_3)
```

In fact, this recursive solution will lead to a worst asymptotic runtime of $O\left(3^n\right)$, which is definitely bad!

With some inspection though, I found that it could be solve more efficiently with **Depth First Search**. The DFS could slightly improve the runtime, and is still easy to implement. The DFS describe as follow:

```
1.  def max_car(x1, x2, x3, cars):
2.      """
3.      The DFS solution with non-recursive stack implementation
4.
5.      :param x1: length of lane 1
6.      :param x2: length of lane 2
7.      :param x3: length of lane 3
8.      :param cars: the list of length of cars
9.      :return: the maximum number of cars that can be parked
10.     """
11.     stack = list()
12.     stack.append((0, 0, 0, 0))
13.     visited = set()
14.     res = 0
15.     while stack:
16.         cur = stack.pop()
17.         x = cur[0]
18.         y = cur[1]
19.         z = cur[2]
20.         num = cur[3]
21.
22.         res = max(res, num)
23.         if num > len(cars) - 1:
24.             return res
25.         if cur in visited:
26.             continue
27.         visited.add(cur)
28.         length = cars[num]
29.
30.         if x + length <= x1:
31.             stack.append((x + length, y, z, num+1))
32.         if y + length <= x2:
33.             stack.append((x, y + length, z, num+1))
34.         if z + length <= x3:
35.             stack.append((x, y, z + length, num+1))
36.     return res
```

We use a stack to keep track of the 'states' that we've been through to avoid recursion costs. Still in the worst scenario it shares a similar complexity with the recursive approach, but overall it better than that.

After some discussion, it turns out that **Dynamic Programming** can efficiently solve it, but with some really

complex coding. We could tackle it with a 4-D DP-table. The runtime of DP is $O\left(l_1 \times l_2 \times l_3 \times n\right)$, which is not a typical polynomial runtime. The key idea is to keep track of all the 4 parameters in the previous recursive approach within a DP table.

For example, when the first car comes in, we add 1 to all of the 3: *dp_table[l1-cars[0]][l2][l3]*, *dp_table[l1][l2-cars[0]][l3], dp_table[l1][l2][l3-cars[0]]*. Then we move to next car in the cars dimension, carry on the information from last step and add 1 to corresponding coordinates and so forth. Since it's impossible to depict it here clearly I'll just keep short - we can do it with DP but it needs delicate design and some clear programming.

Final words on this problem:

- At first I didn't quite get the idea of it. After several hours of searching I found the 'online scheduling' problem, which I thought to be similar to this one. But after clarification I understood that we know all the length of cars in advance, so that we can recursively/exponentially solve it at least.
- The 4-D DP table is really hard to imagine, and I couldn't confidently say that I can implement it with no effort. **Maybe Prof. Eric could consider to design a project homework for next year to deal with this problem!** ☺
- After some discussion with peers, we found that this problem might also be described as follow
  Consider a 3-D maze where we start from [0, 0, 0] and the exit is at [l1, l2, l3]. The step size is fixed and known. What is the number of the most steps that we can take to approach the exit without getting out or just reach the exit?
  This sounds more instructive and could make people think of shortest-path, graph and DP related algorithms.
- Hopefully I won't see such a problem in the finals! 😂

2. Rural Cell Phone Towers

A basic idea is to set the tower as far as possible to the previous house on the left, so that it can potentially cover more houses on the right.

a) First tower is as far as possible down to the right of the first house, i.e. t[0]-h[0]=r
Take record of the distance of the tower, then scan through the houses list until a house that is not covered by last tower was found.
Build next tower to the right furthermost of this house and record the distance of this tower and so on.
The code for this algorithm as follow:

```
1.  def tower_dist(houses, r=3):
2.      towers = []
3.      if len(houses) == 0:
4.          return []
5.      towers.append(houses[0] + r)  # tower 0 is to the right most of house 0
6.      i = 0  # Index of latest tower.
7.      for k in range(2, len(houses)):
8.          if abs(houses[k] - towers[i]) > r:  # house k doesn't have service
9.              i = i + 1
10.             towers.append(houses[k] + r)  # next tower to the right most
11.     return towers
```

b) The worst case asymptotic runtime of this algorithm is O(n), which is the time to loop through all the houses. It's possible to add an 'if statement' to judge if the last house is covered already, but it does not improve the asymptotic runtime.

c) First, the empty case, i.e. there are no houses in the list. Both the optimal and our result do not build towers. Also we should exclude the case such that two or more towers have the same distance. In that case it's not optimal since we can remove replicates.

- Consider an optimal solution that differs from the greedy solution.
- Look at the first tower where they differ, i.e. the optimal solution picked tower[i]=sigma and greed picked tower[i]=g.
- Clearly greedy solution covers further to the right, i.e. g > sigma (because it was the greedy choice).
- That means the optimal solution covers less to the right.
- Suppose that there is one house that is outside the coverage of optimal but inside the coverage of greedy.
- That means we need to add one more tower between tower[i] and tower[i+1] to cover that house for optimal solution, which will make optimal have more towers than greedy.
- That could not happen, otherwise optimal solution will not be optimal! So there are no towers like the statement above, which means the optimal has exact same number as the greedy.

Proved.

3. Greedy Party Planning

We'll try to use the 'take leaves' method.

a) As we can observe from the tree, there is a very important underlying rule: for each parent node with only children, it's always better to take its children! Consider a really small subtree with only one parent. If the parent node has 1 child, either taking its child or itself will be optimal; if the parent node has more than 1 child, then taking its children is optimal. For any part of the tree, we can do the same operations.

- Start with a tree, use DFS to reach its leaves
- While haven't reached its root yet:
  - Take all the leaves of a parent who only have leaf nodes, add them to the result set, cross off all taken leaves and the parent itself
- Returns the result set.

b) The worst asymptotic runtime is $O(n^2)$, since we need to visit each node to decide either to take it or not. The worst case happens when the tree becomes a twig, and for each step we can only cross off 2 nodes. That gives us $n+n-2+n-4+\ldots+1 \sim O\left(n^2\right)$. The best should be O(n), where the tree only have 1 or 2 levels and all the nodes only need to be traveled once.

c) With similar approach as Problem2, we can briefly prove it. First consider the base case, where there is only a root node. It is also a leaf node so we take it.

- Then consider an optimal solution that is better than this greedy solution.
- Look at the first node that the optimal solution differ from greedy solution. If the optimal is different from our greedy solution, then that node must be a parent node, since our greed is to take the leaf node.
- However, if the optimal takes the parent node, we can always cross off that parent node and take its leaves. If it only has one leaf, then greedy is the same as optimal; if it has more than one leaves, then greedy will do better than optimal by crossing off this parent node and take its leaves.
- That could not happen for the optimal! So the optimal will not take any parent nodes who have not crossed off leaves in any step of the tree.
- Therefore, greedy has same or better numbers of nodes than optimal, which makes the greedy solution optimal.

Proved.

<u>In order to give an even more strict proof for this problem, we want to use the strong inductive steps to prove it.</u>

- First, the base case. When no nodes, apparently correct; when 1 node, optimal solution and greedy gives the same result; when 2 nodes, optimal solution and greedy solution both give 1, which is still the same.
- Second, the inductive hypothesis: **suppose that greed gives the optimal solution when number of nodes is less than or equal to k, n<=k.**
- Think about the case when n=k+1. When we cross off 1 or more leaf nodes from the tree, we already know that taking leaves is better than taking their parent (proved above). The other thing that we need to explain is that by removing this subtree, it won't affect the other part of optimality.
- That is to say, when we cross off a subtree, both its leaves and the parent of this small subtree are crossed off. So will the parent node to this subtree be affected? No, because according to our greed,

we never take the parent and cross off the leaves! Therefore, with removing the subtree, we can still either take the parent to that subtree or don't take it - for this particular step our greed approach does not affect the optimality.

- And after crossing off the subtree, n becomes less than k. As our hypothesis states that, when n<=k, the greed gives the optimal result.
- Therefore, by strong induction, the optimality of greed is proved.

After discussing with professor, I decided to use the 'take leaves' greedy method instead. However, I'll just leave my original approach below and hopefully you could help me to know if it's correct.

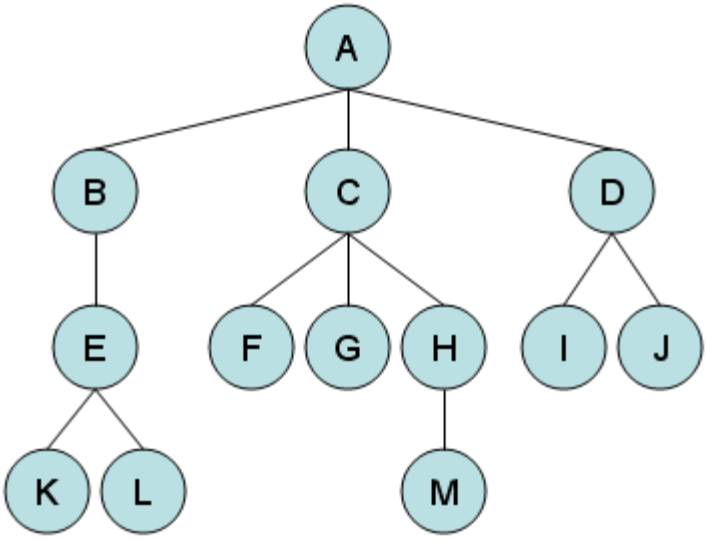
This problem reminds me of the red-black tree introduced by Prof. Hilton. Still, I'm a bit confused that if the head boss, or the root of the tree should be included. Think about a case that there are only two layers of the tree, and the leaves are more than 1. We should take all the leaves instead of the root.

a) The first method comes to my mind is to compare the number of leaf nodes of current node's parent, and the leaf nodes of parent's parent. Here is the description:

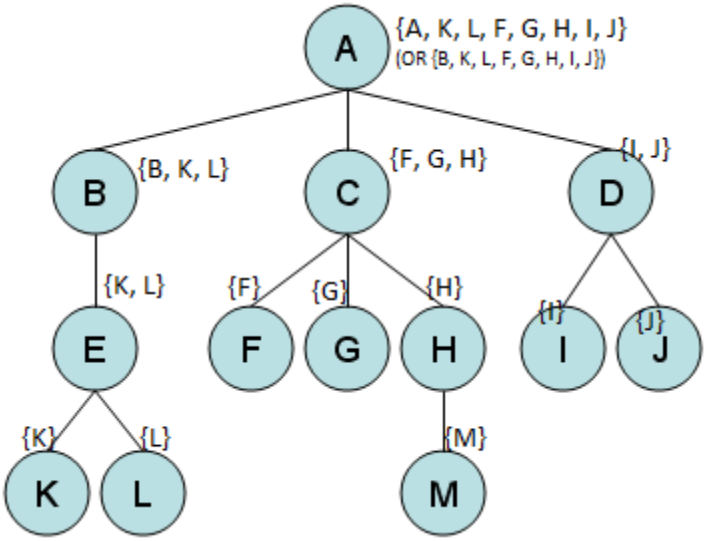
For each iteration, we visit the unvisited leaf nodes, which means it does not have children nodes, or all of its children are visited. That is to say, we loop the tree in a bottom-up fashion.

- For each unvisited leaf node,
  - Let $Set(n) = \max\left(n + \bigcup_g Set(g), \bigcup_c Set(c)\right)$, where g stands for all the grandchildren of current node n and c stands for all the children nodes of n. If the node does not have grandchild or child, then the corresponding sets are empty. In this expression, I used plus symbol to union n and all sets of grandchildren in order to avoid confusion. The max function take the larger set of those two.
  - Keep track of the set for each node and mark them as visited.
- Loop until we reach the root node. Return the set for root node.

It works as follow:



Given the tree above, we start from the bottom leaf nodes (K, L, F, G, M, I, J).



The figure above shows all the sets for each node. When ties in length of set are reached, we just pick the self+grandchildren set.

b) The asymptotic runtime of this algorithm should be $O(n^2)$.

c) To prove that these algorithms are correct, we need to make an observation about the tree.

If node n is a leaf node, there exist a largest independent set that contains node n.

Proof to this observation:

Consider a largest set S.

If n belongs to S, proved.

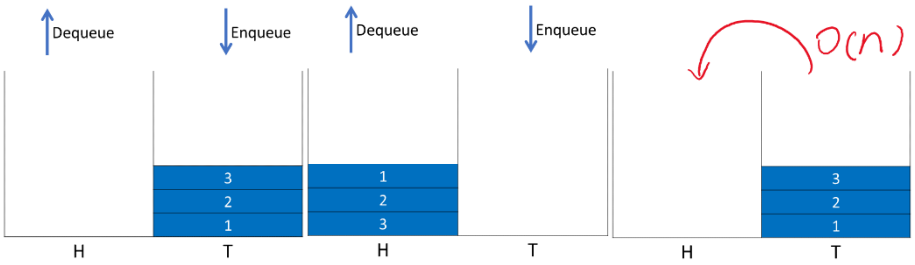
If n does not belong to S:

- n's parent m belong to S: delete m, add n, S is still largest independent set.
- n's parent m does not belong to S: S is not largest set.

Therefore, when we finish looping each level of the tree, we can always ensure our temporary sets are largest.

4. Two Stacks can make a Queue

a) I happened to see one of the discussions of this problem before (6 stacks make a O(1) queue), so I just borrow the figures in that discussion.



Here we can see the whole process of this queue with two stacks.

- For Enqueue we just push into stack T.
- For Dequeue,
  - When stack H is not empty, we just pop from stack H;
  - When stack H is empty, we pop all the elements from stack T and push them into stack H, then pop stack H.

b) Give each element 4 rubles when it is pushed into the stack T.

1 ruble will pay for the first push operation into stack T.

2 rubles will pay for the pop from stack T and push into stack H

1 ruble will pay for the pop from stack H.

This covers all of the necessary work of each element, meaning that we have $4 \in O(1)$ amortized cost.