NOTE: HashMap and HashSet are used in my program, and they are expected to have O(1) performance when we try to insert new element or get a element.

Part 1:

I wanted to represent the neighborhood as a directed graph, and in order to do so, I implemented a modified version of adjacency matrix, specifically :

```
Map<String, HashSet<String>> graph = new HashMap<>();
```

where the key is the String that represents a house, e.g. “House A”, value is a HashSet of Strings that represents the set of houses to which this house has the keys. For example, the following code creates a node “House A” and two edges that goes from “House A” to “House B” and House C”, respectively.

```
Set<String> haveKeysTo = new HashSet<>();
haveKeysTo.add("House B");
haveKeysTo.add("House C");
graph.put("House A, haveKeysTo);
```

I think this makes coding easier, compared to a regular adjacency list, because this makes the code of DFS easier to write. All I need to do to visit all neighbor nodes of a house is to iterate over its `haveKeysTo` HashSet. Furthermore, compared to an adjacency list, checking the existence of an edge takes only constant time.

Part2:

Algorithm:

```
//initialization
Initialize a list called HouseToRob to contain all houses
For each house, keep track of how many more keys I need to rob it.


//main
for every unlocked house H:
   DFS(H)
return true if there is no more house in HouseToRob; return false otherwise


//Depth First Search
DFS(H):
decrement number of keys needed to rob H   //I get to H from a (new) edge, meaning I have another key for it
if I have enough keys to enter H:
   remove H from HouseToRob
   for every house H' to which H has keys:
      DFS(H')
```

Time complexity (for this part of the program only) is O(V(V + E)), the same as the regular DFS.

My DFS is mostly the same as the regular DFS, except mine doesn't call DFS on a house, H', unless I have reached H' from every edge incident to it, i.e. have all keys to H'. Hence, DFS should take O(V + E). In the worst case, we need to call O(V) times, so O(V) * O(V + E) = O(V(V+E)).

Space complexity:

Like a normal adjacency matrix, mine takes O(V^2) space. More specifically, it's a HashMap that maps a String to a HashSet. There are V entries in the HashMap, and each entry's HashSet can have up to V elements, so O(V*V) = O(V^2)

My DFS does not use a stack. It looks up in the graph directly, so there will not be extra space allocated for a DFS stack.

There are also some things that I keep track of, such as

```
private Set<String> housesToRob = new HashSet<>();
private Map<String, Integer> keyGathered = new HashMap<>();
private Map<String, Integer> keyNeeded = new HashMap<>();
private Set<String> result = new LinkedHashSet<>();
```

But apparently none of them is larger than O(V^2), so the total space complexity is still O(V^2)

Part 3:

I used a variation of the greedy algorithm for this part.

```
//algorithm
sort all items by price per pound and put them in a list called itemList (descending order)
for all items in itemList:
    if current item can fit entirely in bag:
        put all of it into bag
    else:
        put as much as possible into bag
```

Sorting takes O(n*log n) time. Going through each item takes O(n) time. So total = O(n*log n) + O(n) = O(n*log n)

My algorithm will not work if items cannot be split into fractions. Counter Example:

Capacity = 10

TV: weight = 8, value = 800

Laptop: weight = 7, value = 699

iPod: weight = 3, value = 299

With my algorithm, I will put TV into the bag first, and then there is nothing else I can fit in the bag, so I can only get a total value of 800. But the optimal solution is to put Laptop and iPod into bag, and the total value will be 998 > 800.

Part 4:

The algorithm is still a variation of the greedy algorithm

```
//algorithm
sort all meetings by earliest finish time and put them in a list called meetingsList
Initialize an empty schedule
for each meeting, M, in meetingsList:
    if M does not have any conflict with the current schedule:
        put it in the schedule
```

Sorting takes O(n*log n) time. Going through each meeting takes O(n) time. So total = O(n*log n) + O(n) = O(n*log n)

Proof of optimality:

By way of contradiction, suppose my algorithm is not optimal, and there is another optimal algorithm that can fit more meetings into the schedule.

Let n be the maximum number such that meetings M1, M2, M3, …Mn are the same in both algorithms. Then when we come to Mn+1, my algorithm would choose a meeting such that it does not have a conflict with the current schedule and have an earlier finish time compared to other candidate meetings. Then no matter what other meeting the optimal solution chooses, mine will definitely leave the same amount or more space in the schedule (due to its earlier finish time) Therefore, no matter what the optimal solution have after Mn+1, my algorithm can cover them all, if not more. In other word, my algorithm is always ahead of the optimal solution.

Thus, my algorithm is optimal.