P1

For the first problem, as given in the figure 3 of sample 2, it looks pretty much like a tree, so I check the question in this way. First, I would like to build a tree. Unlike ordinary tree, the tree contains the length of the “branch”, but we do not need to record all of that in the data structure since we only need to find the deepest one. Please noted the length of branches is calculated by the attack power minus the depth. Recall in the tree functions, we often use recursion. To improve time complexity, I began wounder how. It turned out that we can store the current total length of a level of height and then we can move to the next level. And we have to add the depth of a tree in the tree class, which is easy just by a recursion in the deepest function. We go through the length by max(max, current) pattern and update until we come to the leaves. When I was doing the program, I actually miss the requirement that the tree is undirected tree, so I then added the tree can go through the edges in both directions, otherwise, some possibilities may be ruled out. So the way to update the maximum length is to check the branch’s current neighbors. When all neighbors, or siblings of the leaves are checked, we can backtrack and move to siblings of the parent. So we first build the tree and calculate the results. To be specific, first, come to root

(destination, where HP = MP = 0), then find the furthest start point, by first go right, then go to the root of the children then go left, go through all of this and find maximum. Do not try to invent new elements to the tree, its far more complex. The logic is clear and actually don't take me much time.

P2

It's a really hard question, so I try to carry out it step by step. First, build the shelves. The question recommend us to build tuples with 3 elements, however, due to we will check the item taken from the shelf later, I change it to 2 elements, excluding the shelf number, and put the tuple in a list representing the shelf it belongs to. After building the shelves, we work on the first restriction: pick items simultaneously, its pretty like a sliding window problem, but our list-tuple data make it hard (actually I don't think it will work out). So I simply swap all the items of one shelf totally to the bag. Is the bag is full, compare the last and the first, drop the smaller one and continue until it is valid. Then, move to the next shelf. What about the circular behavior? I have 3 plans, first is go from 2 directions, second is traverse twice, third is linked list. Third one is ruled out since it's two complicated consider the empty shelf and I choose the second one since it is intuitive (first one is also OK and time efficient). Then the second

restrictions, same number. I simply set a set to record the number taken from the shelf respectively. It’s easy. Last, the empty one and I introduce the variable that test whether the former shelf is empty, if so, break the process and move to the next non-empty shelf (It looks like a linked list in this case). Finally, we start with a non-empty shelf, to avoid specially case that both forst amd the last shelf is empty. In fact, the key part the program is the ring structure. I come up with the example of deciding whether a circular linked list at lecture notes, so why don’t we go through the shelves twice? The problem is solves now (probably the most difficult question in the course up till now).