Prac 6 Documentation by Pavin Wu (26916592)

Task1

We are solving this problem in two stages. First, the step to find each of a solution by memoizing past solutions. Then in the next stage, we use the memoized past solutions to find the actual sequence of the total solution by backtracking.

Find solution stage: general idea is to look at all the total sale from previous houses with more than length k apart to the current house; find which is maximum; add the maximum total sale with current house's value; then store that sum together with other past total sales in a list of past total sales.

For example, if k=3, and current house we are looking at is i=7 (8th entry in the list of houses) with value 12. We look at all the past total sales from house 0, 1, 2, 3 (1st, 2nd, 3rd and 4th entry in the list of houses) and find which is maximum. Say 65. Then we add 65 with 12 to get 77. Finally, we append 77 into the list of past total sales, which also contains 65 in either position 0, 1, 2 or 3. Note how we don't take the total sales from house 4, 5, 6 since these houses are with length k=3 from the current i=7 house.

Be aware that this will not work with the first k + 1 houses since there will be no houses before this range of neighbourhood. But because these houses have no neighbour, each total sale we get from this stage will be only the value we get from each single house in the sequence. So all we have to do in this stage is just append the house value into the list of past total sales, one by one in order.

One caveat with looking at "all" valid houses (i.e. "more than length k apart …") is that the look up time will increase by 1 each time we go through each houses (after the first k+1 houses). This will lead to O(N^2) time complexity. To solve this, we can see that there is only one maximum amongst all the past valid houses sales. This means when we have a new total sale from a new house, all we need to do is compare this new total sale with the previous maximum, and assign the maximum amongst these two as the new maximum of total sales from past valid houses. This will result in O(N) time complexity since we only have to do O(1) comparison in each iteration.

Backtracking stage: general idea is to get the maximum in the list of all past total sales (which is the true maximum sales); find the entry with this maximum in the list of past total sales; subtract the house value corresponding to this entry; then treat this difference as a new sub-maximum; then find the entry with this maximum in the list of past total sales, and so on. Note that we also have to keep the house entries we found (after searching the list of past total sales) into a separate list of solution houses so we can print the solution to the sales man.

The process repeats until the difference is 0, or in our case, until we reach the end of the list of past total sales. Our implementation works the same since we were told that everybody in the town are willing to buy something i.e. each house has > 0 value. This means when difference reaches 0, no new entry from the list of past total sales will be recorded.

<u>Worst case space</u>: We need to create a list of length N for storing house values and storing past total sales after visiting each house. These take O(N) space. We also need to create a list to store the sequence of houses to visit. This takes at worst O(N). Since all the other variables take O(1) space, the overall space complexity is O(N).

<u>Worst case time</u>: Find solution stage takes O(N) since we only do O(1) operations (such as assignments, additions, comparisons) for each house in the list. We also treat lists as arrays, so accessing an item in a list is O(1). Finding the maximum past total sales with built-in function takes O(N). Backtracking takes O(N) since we go through all entries in the list of past total sales once and the number of entries in this list is N. All other operations take O(N) or O(1). Therefore, time complexity is O(N).

Task2

Similar to task1, we are solving this problem in two stages: find solution stage, and backtracking stage.

Find solution stage: Instead of making a list of past solutions of size N like task 1, we use a matrix of size O(N*T) instead, where N is the number of songs, and T is the duration Alice takes to travel. In the matrix, the rows of T will correspond to each total partial duration after listening to each song. Each column of N corresponds to each song, with N=1 corresponding to song with id 1 and so on. Note that we still need a list of size N to contain the duration of each song in Alice's library.

First, we create a zero matrix (contains only 0) of size T*(N+1). General idea is we treat the matrix as a record for whether or not the duration after listening to each song is possible. If it is, we denote the entry in matrix corresponding with the duration *t* and the song *n* as 1 where *t* <= T and *n* <= N, otherwise the entry will be left as 0. We find whether a duration is possible by summing a duration of the song *n* with every entry in the previous column. This previous column contains all the possible total partial duration we have found so far. When we sum the duration of the song *n* with an entry in the previous column, we will get a new duration. If this new duration can exist (i.e. <= T), we will set the corresponding (t, n) entry of the matrix to 1. If it cannot exist, we discard it.

Note that by this implementation, to be able to add the possibility of first song's duration into a matrix, we need to create a dummy column for the first song's duration to sum to. This is why we need to make N+1 columns and leave the first column (with index n=0) as 0.

We repeat this process until we have gone through all items in the previous column, and have denoted all possible duration after listening to the current song *n* (into the column of n) in the matrix.

After that, in order to be able to access previous possible durations which come before the last one, we need to copy all entries marked with 1 in previous column to the current column *n* we are considering.

Repeat this process until we have gone through every song in the songs' durations list.

As a result, if there's a sequence of songs which have the total duration summed up to T, there will be at least an entry within a row of T (last row of matrix) that is marked with 1. So to find whether or not we should tell Alice bad luck, we simply search if the row of T has the condition satisfied.

Backtracking stage: This is also similar to task 1 backtracking, but instead of simply subtracting and look up values, we need to take into account the fact that we have copied the entries marked with 1 after we finished with each *n*.

First note that although both the row of T in the matrix, and the T itself is incremented by 1 each passing row, our implementation has the row of T starts from 0. Since we know T always starts from 1 (song with duration t=0 is not a song), we will need to subtract total duration T with 1 when we want to refer to a row in the matrix associated with the duration t. Same reasoning goes for N and the index in the list of duration of songs.

So to backtrack, we first initiate a list which we use to put the IDs of the songs with total duration T. Then we find where in the row of T does 1 appear. We take that entry and append it to the list of IDs.

After this, similar to task 1, we also subtract T by the duration of the song corresponding to the column *n* to get a duration difference. Next, we go backward to previous song by going into a previous column *n-1* and the row corresponding to the duration difference. We do a similar process to see if this particular entry is marked with a 1. Now, unlike task1 where we take the solution immediately, we also have to check whether the entry in the same row but with column *n-2* is an also an entry with a 1. If it is, it means we copied this 1 from previous duration solution, and therefore the song which contributes to the solution is actually the song from the entry *n-2*. We repeat this until we find the first entry with 1 in the row. Only then do we append the song's ID (corresponding to the column of the entry) into the list of songs' IDs.

Same as task 1, repeat this process until the difference is 0.

<u>Worst case space</u>: We need to create a matrix of size O(NT) to store possibility of the duration. We also need a list of size O(N) to store songs' durations and the solution IDs. We also need some other variables for the algorithm to function which takes O(1). Therefore, overall space complexity is O(NT)

<u>Worst case time</u>: When we are marking O(NT) matrix with 1's for each song *n*, we need to go through all the entries in previous column, and add the duration corresponding to the entry to the song's duration. As we can assume the matrix works like an array i.e. constant access time, the time complexity for finding solution is O(NT). During backtracking, since we always start from the last column and go towards the first one by one in each iteration, backtracking takes O(N). Since all other operations take either O(N) or O(1), the overall time complexity is bounded by O(NT).