Q1/ OUR APPROACH TO THE FIRST PART OF THE PROBLEM

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. Taking the Knapsack problem as an example, we accepted the durations of the songs as weight. We considered their popularity as value. In our approach to the first part of the problem, the first thing we do is divide the popularity by the duration and get a ratio. We do Value divided by Weight. This gives us the ratio. We assume that the higher the ratio, the more valuable the music. We are trying to increase the popularity and decrease the duration. In the ratioList, we prioritize the one with the highest ratio and place it. We place the valuable song as the first index. The song with the highest ratio is thus placed first. We have a list called FlagList, we adjusted this list according to ratioList. We should not see the song we placed first while selecting the second song. For example, in our solution result, the song number 25 with a large ratio was chosen first. The song number 27 was chosen as the second song. Since we do not want song 25 to be seen while song 27 is being selected, in the code flagList[loc]=true; we are accepting. In this way, when true, we can add new songs one by one without including the previously selected songs. However, even if the song we added at some point finds the maximum ratio, since it exceeds the total weight, that is, the total duration, we ensure that it does not add it by saying if(total\_weight-weightList.get(loc)<0) { so as not to exceed the total weight. We say continue instead of break because there may be ratioLists that we don't see afterwards and we want to be able to see and place them all. When we look at our solution with the exact solution, we found that the objective of the exact solution is 532, and the objective value in our solution is 644.30342. So together with our algorithm approach, we found an even more advanced objective result.

Q2/ OUR APPROACH TO THE SECOND PART OF THE PROBLEM

For the solution of the second part of our problem, while going to the nearest neighbor in the normal Tsp problem, we ran our algorithm based on the values that are the farthest from the other values, that is, the largest values of our values, not the closest values to the other values in our algorithm. we choose the list we prepared in the first code. Then we open a new list so that we can throw the numbers we have chosen into it. Our index should be the first thing we look at in the sequential list, and it is again 25 because it is at the beginning. Then we write the initial values and enter the while loop and say that the size and length of our list must be one less than the length of the list we send. The while loop loops until there is one missing length. Since there is a rule to put the second most valuable song last, we create a list with the song removed at the beginning, and therefore we want the size of the list to be one less. In the other while loop we have location and max. From the 2nd index, we are trying to find the maximum in the song intervals. As they are found, the maximum and location are placed. After the for loop is finished, the flagList is made true and the index is changed. My index is whichever is bigger. The index I found is added, so the sequence in the sortedList is clear.