IE 343 Term Project Code Report

Ata Seyhan, Ahmet Özefe

December 31, 2022

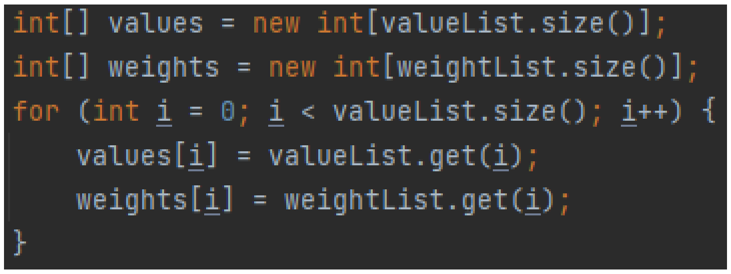
1 Introduction

In this term project, we are given 2 problems. The first problem expects an exact algorithm and optimal objective function value with the solution as an output of the code. The second problem however, cannot be solved with an exact algorithm due to time inefficiency. We are asked to come up with a heuristic algorithm. In the following sections we will discuss the algorithms, heuristic approach and the numerical results.

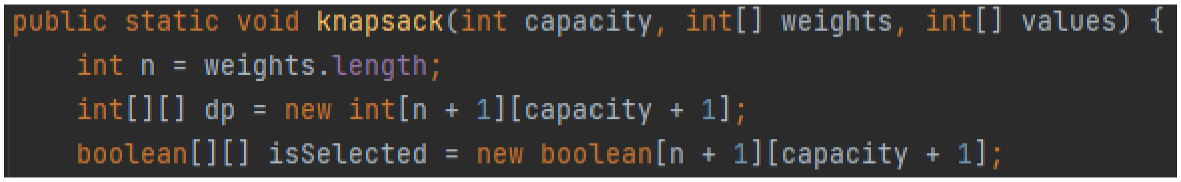
2 The First Problem

We first need to understand what kind of problem we are given in order to solve it. We have song popularities, durations and album length. We are asked to choose which songs to include in our album. This sounds very much like the 0-1 Knapsack Problem. 0-1 Knapsack Problem can be solved with various algorithms. In this project, we used The Bottom-Up approach.

First, we converted valueList and weightList ArrayLists into standard arrays. This was done to make them compatible with the Bottom-Up algorithm that we used. Also ArrayLists cannot be multidimensional, which is a constraint. This step was done in the main function.

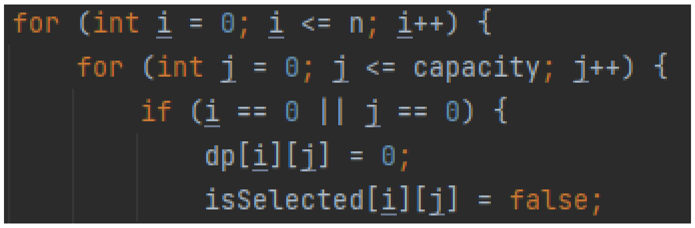

```java
int[] values = new int[valueList.size()];
int[] weights = new int[weightList.size()];
for (int i = 0; i < valueList.size(); i++) {
    values[i] = valueList.get(i);
    weights[i] = weightList.get(i);
}
```

Then, outside of the main function, we defined a knapsack method that takes 3 arguments: capacity number, weights array and values array. Here, weights are durations and values are popularities.

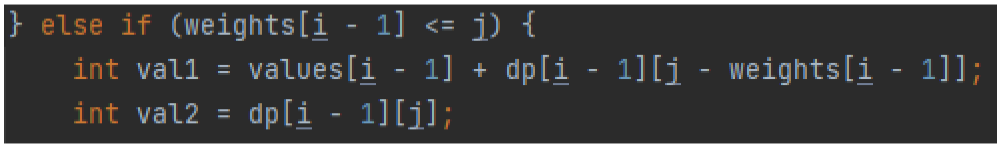

```java
public static void knapsack(int capacity, int[] weights, int[] values) {
    int n = weights.length;
    int[][] dp = new int[n + 1][capacity + 1];
    boolean[][] isSelected = new boolean[n + 1][capacity + 1];
```

We needed the number of items to choose from so we assigned **weights.length** to **n.** After that, we initialized the Bottom-Up table. The Bottom-Up table is an **n+1 x W+1** matrix that holds all possible sub-solutions to the original problem. We called this 2 dimensional matrix ***dp***, for Dynamic Programming. Bottom-Up is Dynamic Programming after all. ***isSelected*** boolean matrix is used to tell whether we included an item in the knapsack or not.

After the initialization, we start building the table in Bottom-Up manner.

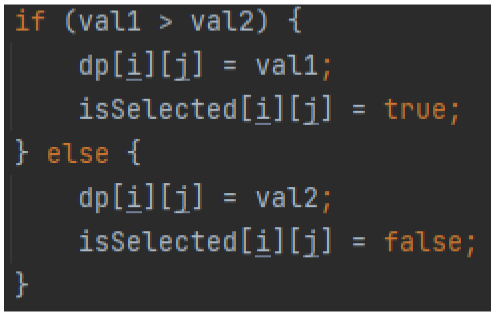

```java
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= capacity; j++) {
        if (i == 0 || j == 0) {
            dp[i][j] = 0;
            isSelected[i][j] = false;
```

We must go through all possible n and W values and that's what the 2 for loops are for. If the number of items = 0 or the capacity left = 0, we do not include any items and we return 0. We also set the boolean False.



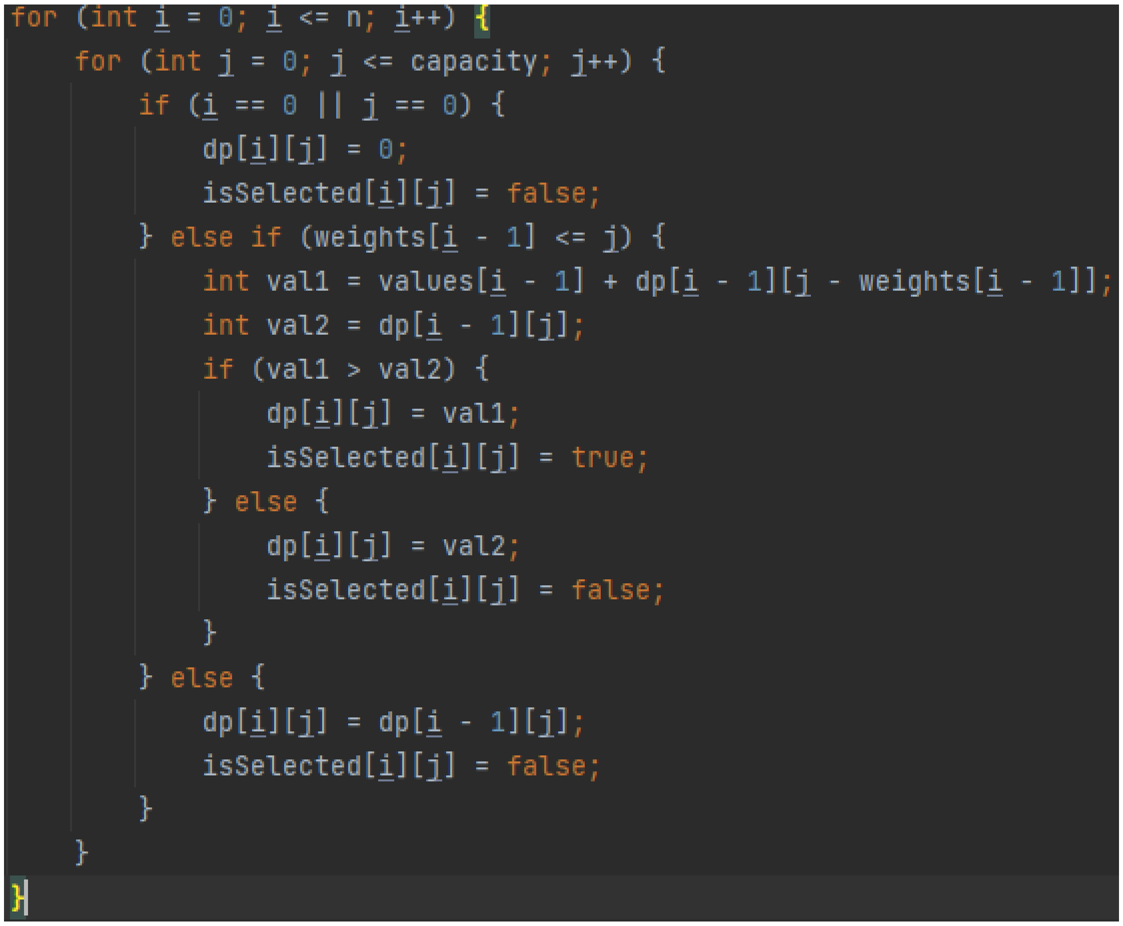
```
} else if (weights[i - 1] <= j) {
    int val1 = values[i - 1] + dp[i - 1][j - weights[i - 1]];
    int val2 = dp[i - 1][j];
```

In this snippet of code, we check for condition: if the weight of the item is less than or equal to the capacity. After that we see a familiar function: The Value Function that we saw throughout the IE343 course. Here, **val1** represents *include* and **val2** represents *do not include.*



```
if (val1 > val2) {
    dp[i][j] = val1;
    isSelected[i][j] = true;
} else {
    dp[i][j] = val2;
    isSelected[i][j] = false;
}
```

Here we check for maximum of the two. If the included value is greater than not included value, then we must obviously include. If not, we simply do not include the item in the knapsack.

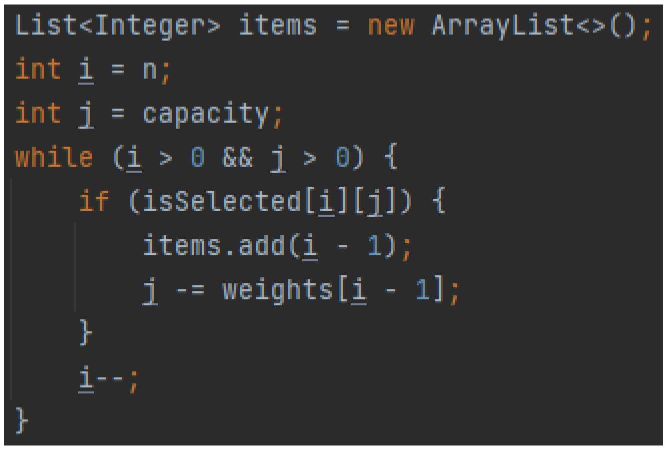


```java
for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= capacity; j++) {
        if (i == 0 || j == 0) {
            dp[i][j] = 0;
            isSelected[i][j] = false;
        } else if (weights[i - 1] <= j) {
            int val1 = values[i - 1] + dp[i - 1][j - weights[i - 1]];
            int val2 = dp[i - 1][j];
            if (val1 > val2) {
                dp[i][j] = val1;
                isSelected[i][j] = true;
            } else {
                dp[i][j] = val2;
                isSelected[i][j] = false;
            }
        } else {
            dp[i][j] = dp[i - 1][j];
            isSelected[i][j] = false;
        }
    }
}
```

Here is the full loop that fills out the table in Bottom-Up approach. The last **else** statement is simply there to catch any other possible situations.

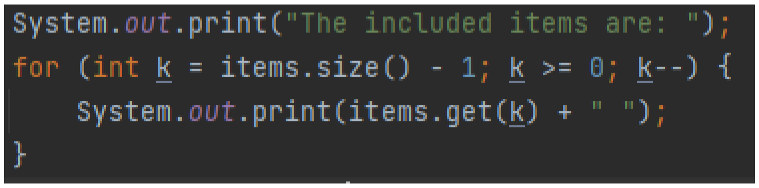
if n = 0 or capacity = 0, do not include.
else if weight of item n < capacity, check the value function.
else, do not include.

Now we have the full table and the information as to which items are included at each step. We can use this information to figure out the optimal solution.



```java
List<Integer> items = new ArrayList<>();
int i = n;
int j = capacity;
while (i > 0 && j > 0) {
    if (isSelected[i][j]) {
        items.add(i - 1);
        j -= weights[i - 1];
    }
    i--;
}
```

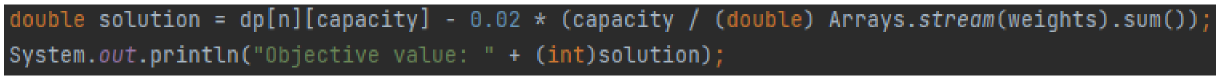
An ArrayList is initialized to store the indices of the included songs. This will be important later. We loop through all the possible values of **isSelected** array and if an item is selected, we put the index of that item in the **items** list. This code outputs the indices in a descending order. To reverse this order, we use the following snippet of code:



```java
System.out.print("The included items are: ");
for (int k = items.size() - 1; k >= 0; k--) {
    System.out.print(items.get(k) + " ");
}
```

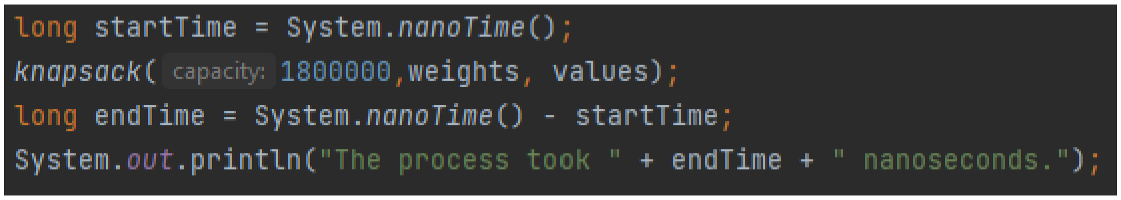
So far, we have filled out the Bottom-Up table, found which items to include in the optimal solution and figured out their indices.
All that's left is to find the optimal objective function value which is given by the equation:

solution = last entry in the dp table - 0.02*(Remaining time in the album)



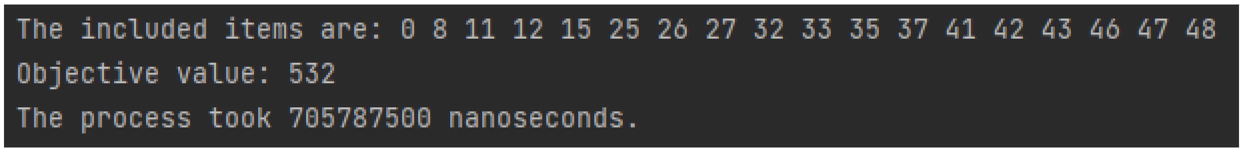
```java
double solution = dp[n][capacity] - 0.02 * (capacity / (double) Arrays.stream(weights).sum());
System.out.println("Objective value: " + (int)solution);
```

In the main function, we call knapsack(1800000,weights,values) to find the solution. Right before calling knapsack, we call System.nanoTime() to keep track of the time. After knapsack ends, we call System.nanoTime() once more and calculate the difference. This number is the time that knapsack method took to finish calculation in nanoseconds.



```java
long startTime = System.nanoTime();
knapsack( capacity: 1800000,weights, values);
long endTime = System.nanoTime() - startTime;
System.out.println("The process took " + endTime + " nanoseconds.");
```

The optimal solution and the optimal objective function value is as follows:



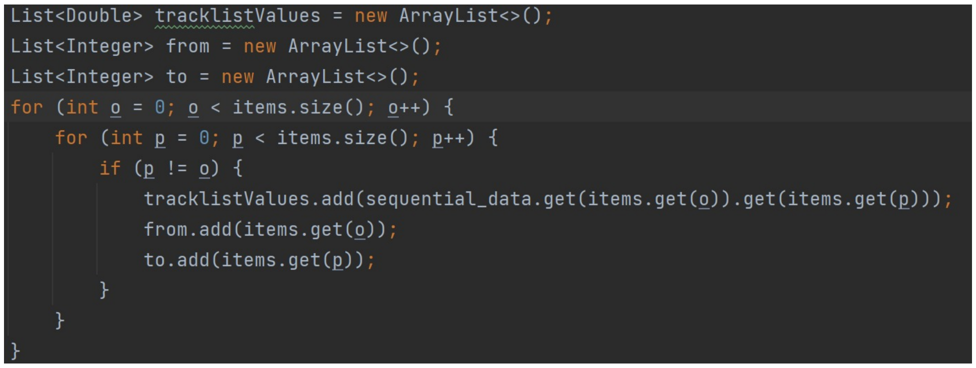
```
The included items are: 0 8 11 12 15 25 26 27 32 33 35 37 41 42 43 46 47 48
Objective value: 532
The process took 705787500 nanoseconds.
```

Our knapsack method takes approximately 0.7 seconds to find the optimal solution. Compared to the provided exact method, which takes 0.02 seconds, our approach requires 35 times more time to reach the solution.

3 The Second Problem

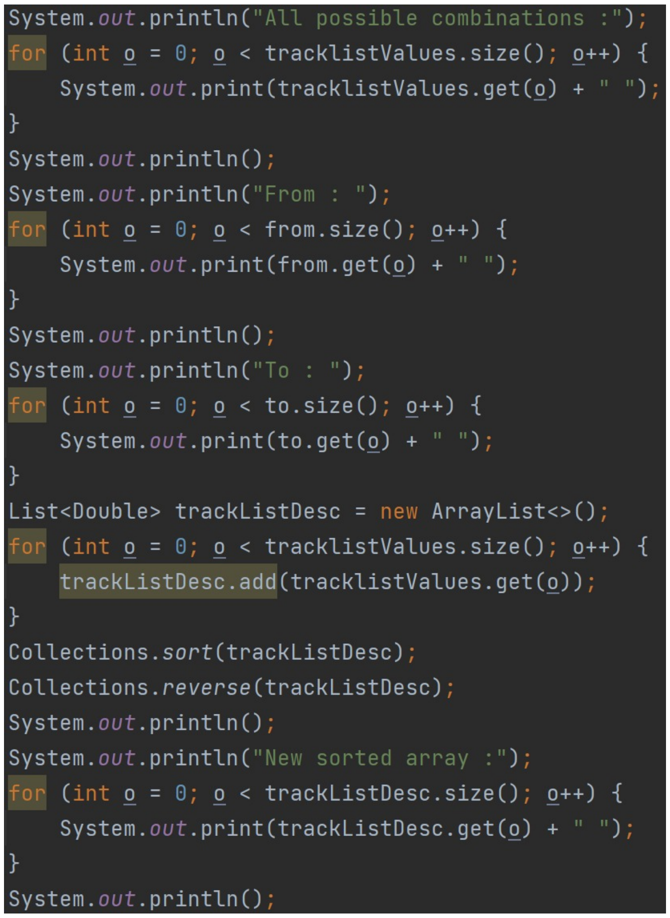
In this part we created 3 arraylists called **tracklistValues**, **from** and **to**. The tracklistValues arraylist keeps all the sequential values between 2 items and we did it for all the possible 2-pairs of selected items.

The from and to arraylist contains from which item pair we got this sequential value in the tracklistValues arraylist. For example if tracklistValue index 9 equals to 5.93 we can look at from index 9 and to index 9 to see which two items give us this value.



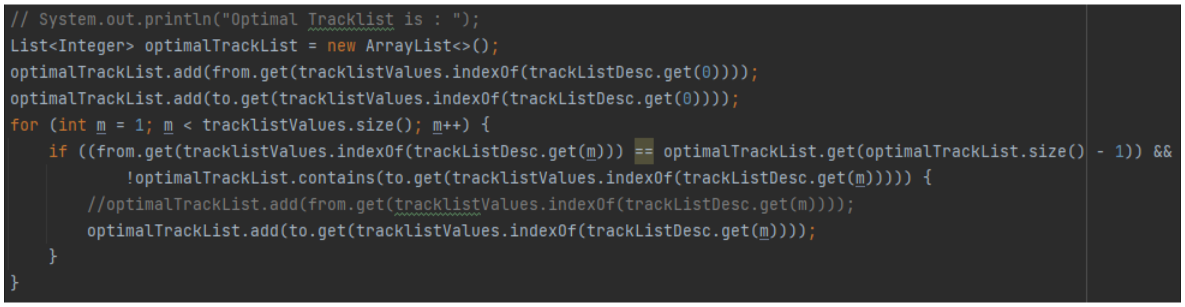
```java
List<Double> tracklistValues = new ArrayList<>();
List<Integer> from = new ArrayList<>();
List<Integer> to = new ArrayList<>();
for (int o = 0; o < items.size(); o++) {
    for (int p = 0; p < items.size(); p++) {
        if (p != o) {
            tracklistValues.add(sequential_data.get(items.get(o)).get(items.get(p)));
            from.add(items.get(o));
            to.add(items.get(p));
        }
    }
}
```

In the code below, we have printed out all the possible 2-paired combinations and from which item to which item that we got that value. Then we sorted the value array in decreasing order to see what are the biggest values so that we can find the item pairs using our to and from arraylist.



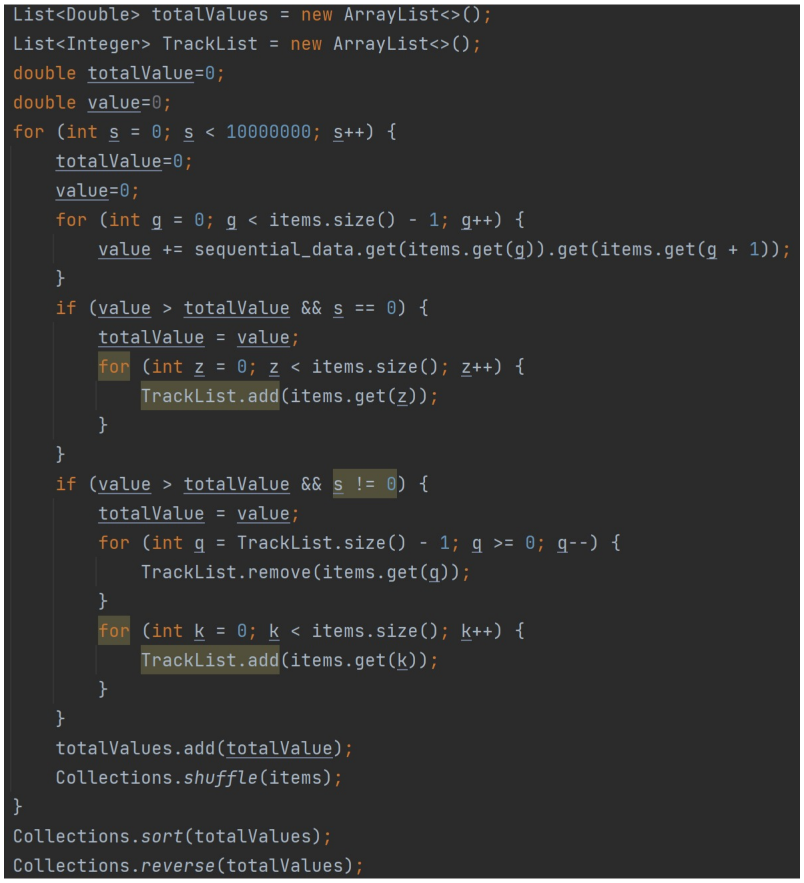
```
System.out.println("All possible combinations :");
for (int o = 0; o < tracklistValues.size(); o++) {
    System.out.print(tracklistValues.get(o) + " ");
}
System.out.println();
System.out.println("From : ");
for (int o = 0; o < from.size(); o++) {
    System.out.print(from.get(o) + " ");
}
System.out.println();
System.out.println("To : ");
for (int o = 0; o < to.size(); o++) {
    System.out.print(to.get(o) + " ");
}
List<Double> trackListDesc = new ArrayList<>();
for (int o = 0; o < tracklistValues.size(); o++) {
    trackListDesc.add(tracklistValues.get(o));
}
Collections.sort(trackListDesc);
Collections.reverse(trackListDesc);
System.out.println();
System.out.println("New sorted array :");
for (int o = 0; o < trackListDesc.size(); o++) {
    System.out.print(trackListDesc.get(o) + " ");
}
System.out.println();
```

In the code below, we have created an optimal tracklist array where we will put the optimal tracklist. We first added the most valuable pair in our descending list. Then we iterate through the descending value list and if that values' from part is the same with the last element in our arraylist and if our arraylist does not contain this item we added that item because they are linked. We added the to part to our array to make the sequential connection.

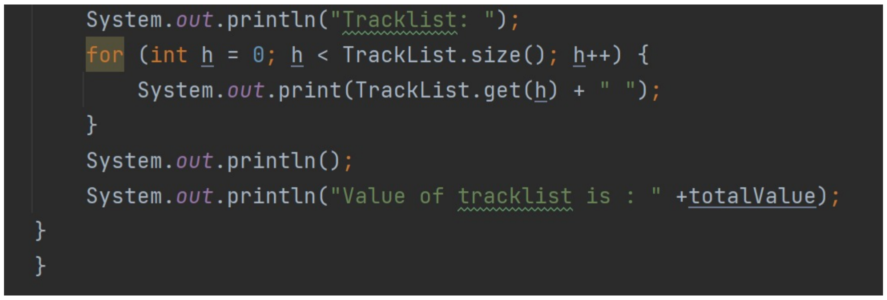


```java
// System.out.println("Optimal Tracklist is : ");
List<Integer> optimalTrackList = new ArrayList<>();
optimalTrackList.add(from.get(tracklistValues.indexOf(trackListDesc.get(0))));
optimalTrackList.add(to.get(tracklistValues.indexOf(trackListDesc.get(0))));
for (int m = 1; m < tracklistValues.size(); m++) {
    if ((from.get(tracklistValues.indexOf(trackListDesc.get(m))) == optimalTrackList.get(optimalTrackList.size() - 1)) &&
            !optimalTrackList.contains(to.get(tracklistValues.indexOf(trackListDesc.get(m))))) {
        //optimalTrackList.add(from.get(tracklistValues.indexOf(trackListDesc.get(m))));
        optimalTrackList.add(to.get(tracklistValues.indexOf(trackListDesc.get(m))));
    }
}
```

In the part below, we have created two arraylists **tracklist** and **totalvalues**. We are giving value the value of sequential 2 items' value and if this value is greater than the totalvalue, which is max value got from these 2-paired items, we add the items to the tracklist until we get a better totalvalue(max value). For the first iteration we directly add the first item list because there is no old list. But for the rest of the iterations we check if the list is full. If it is full but we have a higher value, we delete the elements in the list and add our new element order to the list. In the last part of the for loop we shuffle the item list every time and we do this for *1000000* times to cover most possibilities. And we also save all these total values in the totalValues arraylist.



```
List<Double> totalValues = new ArrayList<>();
List<Integer> TrackList = new ArrayList<>();
double totalValue=0;
double value=0;
for (int s = 0; s < 10000000; s++) {
    totalValue=0;
    value=0;
    for (int g = 0; g < items.size() - 1; g++) {
        value += sequential_data.get(items.get(g)).get(items.get(g + 1));
    }
    if (value > totalValue && s == 0) {
        totalValue = value;
        for (int z = 0; z < items.size(); z++) {
            TrackList.add(items.get(z));
        }
    }
    if (value > totalValue && s != 0) {
        totalValue = value;
        for (int g = TrackList.size() - 1; g >= 0; g--) {
            TrackList.remove(items.get(g));
        }
        for (int k = 0; k < items.size(); k++) {
            TrackList.add(items.get(k));
        }
    }
    totalValues.add(totalValue);
    Collections.shuffle(items);
}
Collections.sort(totalValues);
Collections.reverse(totalValues);
```



```
        System.out.println("Tracklist: ");
        for (int h = 0; h < TrackList.size(); h++) {
            System.out.print(TrackList.get(h) + " ");
        }
        System.out.println();
        System.out.println("Value of tracklist is : " +totalValue);
    }
}
```

In the last part we are printing the optimal tracklist and our objective tracklist value.