# CS333 Project 2

**Hasan Can Sevindir**

In this project it was expected to find the greatest number of possible ways to connect n amount of lamp posts and outlets, that has matching symbols, while making wires not cross with each other using a dynamic programming approach.

## Solution approach:

I realized the problem can be simplified. If I get the indexes of the outlets (first list) then assign the numbers to lamps (second list) with same symbols, I just needed to find the longest non-decreasing list of numbers in the second list. Let me explain with the following example:

Outlets: AA B4 BE DC 72 96

Lamps: BE DC 72 AA 96 B4

If I get the indexes of symbols from first list (AA: 0, B4: 1, BE: 2 …) and assign the indexes with the same symbols on the second list, this list of indexes will look like this:

| | | | | | | |
|---|---|---|---|---|---|---|
| Lamps: | BE | DC | 72 | AA | 96 | B4 |
| Indexes: | 2 | 3 | 4 | 0 | 5 | 1 |

As seen here, the longest non-decreasing list is 2, 3, 4, 5 this list shows the symbols that should be matched. So, in this case it is:

BE DC 72 96

with length of 4.

And it is indeed correct as it can be shown as:

```
Outlets: AA B4 BE DC 72 96
               |   |   |    \
Lamps:         BE DC 72 AA 96 B4
```

Same solution also works if we take the first list as lamps and second list as outlets. This solution is easily applied to n number of outlets and lamps.

## The code:

As seen in the algorithm above, I didn't use any nested loops. It is linear. Only parts in the code that can affect the calculation time are as follows:

- Splitting the inputs to arrays
- Checking if all symbols are valid hexadecimal symbols
- Checking if the symbols in two inputs match
- Initializing the indexes array
- Calculating the longest non-decreasing index list
- HashMap insert and look up operations

Other than these, I did not use any other data structures, nor did I write a class myself for no special operations.

HashMap is initialized from lamps list as follows:

```java
HashMap<String, Integer> lampDict = new HashMap<>();
for (int i = 0; i < lampArray.length; i++)
   lampDict.put(lampArray[i], i);
```

Here, the lampDict is the HashMap. I put lamp list values with its indexes for later use. Then I form the index array as follows:

```java
int[] indexArray = new int[outletArray.length];
for (int i = 0; i < indexArray.length; i++)
   indexArray[i] = lampDict.get(outletArray[i]);
```

Here I iterate through the outlet list. I get the value of element each element from the outlet list, look up the element in HashMap and get its assigned index from the HashMap. By doing this I get the index of element of outlet list, and what index it has in the lamp list.

Then to calculate the longest non-decreasing number list, I iterate through the index list as follows:

```java
ArrayList<Integer> longestSequence = new ArrayList<>();
ArrayList<Integer> currentSequence = new ArrayList<>();
currentSequence.add(indexArray[0]);

for (int i = 1; i < indexArray.length; i++) {
  int num = indexArray[i];

  if (num > currentSequence.get(currentSequence.size() - 1))
    currentSequence.add(num);
  else if (currentSequence.size() > longestSequence.size()) {
    longestSequence = new ArrayList<>(currentSequence);
    currentSequence = new ArrayList<>();
    currentSequence.add(num);
  }
}

if (currentSequence.size() > longestSequence.size())
  longestSequence = currentSequence;
```

Longest sequence is the longest non-decreasing number list and the current sequence is the current non-decreasing number list, since there can be multiple lists of non-decreasing numbers. After finding a non-decreasing list, it is compared with previously assigned list to see which list is longer. Upon finding a longer list, longest sequence is updated. After the iteration ends, one final check is done to see if the final current sequence is the longest or not.

## Time Complexity Analysis:

Here, I will talk about the time-consuming operations that I listed at the section above. I included HashMap operations because even though the average time complexity is O(1), the worst case scenario is O(n). Or so I thought. As of Java 8, HashMap operations can be calculated at O(logn) complexity if the keys can be compared for ordering, in the worst case of course. Explanations for other points are as follows:

**Splitting the input** is done by using Java's String.split method. This function works by:

1) Iterating over the String.
2) Taking a regex input and calling Matcher.find method underneath to find the next word boundary.
3) Calling String.substring method to extract the matched word.
4) Adding each word to a list of Strings.
5) Converting to the list to an array type.

There aren't other loops nested within the algorithm, as seen in Java docs, thus String.split function called with a single character (space) takes O(n) time.

**Checking if all symbols are valid hexadecimals** is done by using String.matches method which takes constant time. Repeating this procedure for each element in the array takes O(n) linear time.

**Checking if the symbols in two inputs match** is done by the help of contains function from previously initialized HashMap instance. HashMap contains takes constant time and checking that the all elements of outlet is present inside the HashMap and checking if the length of lamp array and outlet array once is enough for the conclusion and takes O(n) linear time.

**Initializing the indexes array** is done by the help of HashMap.get and array element access operator (subscript operator) inside a linear loop. Get method takes constant time, subscript operator takes constant time and thus the entire initialization takes O(n) linear time.

**Calculating the longest non-decreasing index list** is done by the way I explained at the section above. It uses a linear loop and calls HashMap.get, ArrayList.size and ArrayList.add methods inside the loop, and all of these functions take constant time. Therefore, it takes O(n) linear time.

**HashMap insert and look up operations** take constant time and O(logn) in a worst-case scenario. But since it is very rare occasion that hashing 2 strings will result in the same output, and this collision is probably impossible in our case considering all of the valid inputs are very limited 2 digit hexadecimal inputs, it is safe to say all operations of HashMap are executed in O(1) constant time.

So, this algorithm successfully finds the way that turns on the maximum number of lamps without making the cables cross. It takes O(n) linear time to complete for n many outlets and lamps.