ÖZYEĞİN ÜNİVERSİTESİ

2022

CS 333

Project 2

Dynamic Programming

by: Selin Köleş (S021541)

Table of Content

Introduction

This project aims to implement a dynamic programming solution for the n electric outlets and n electric bulbs problem. This report gives a detailed insight into the given problem statement, a brief introduction to Dynamic Programming and the steps I followed, implementation of code, dynamic problem solution to the given problem, and time complexity analysis of the solution.

Problem Statement

There are n electrical outlets and each one can turn on just one electric lamp. Each outlet has a unique code that can only be used to turn on the lamp. A wall has outlets mounted on it, and the lamps are mounted in some order in front of the outlets on another wall.

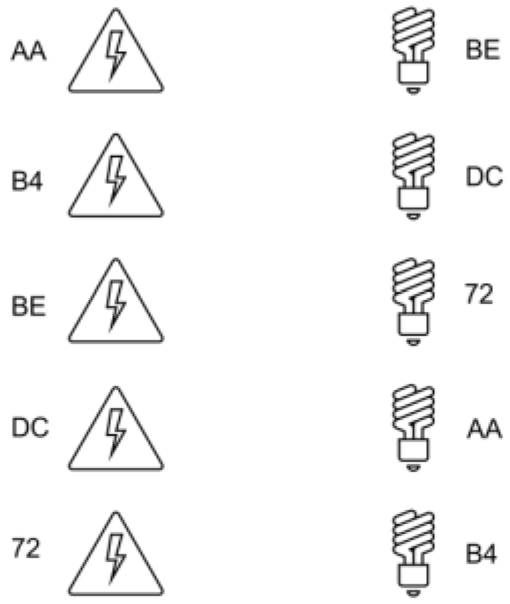


Fig 1. The figure shows the arrangement of electric outlets and bulbs mounted on a wall.

We want to use wires to connect electric outlets to lamps so that the lamps can be turned on. However, the connection wires cannot cross each other.

For example, the wiring in the following figure is not acceptable.

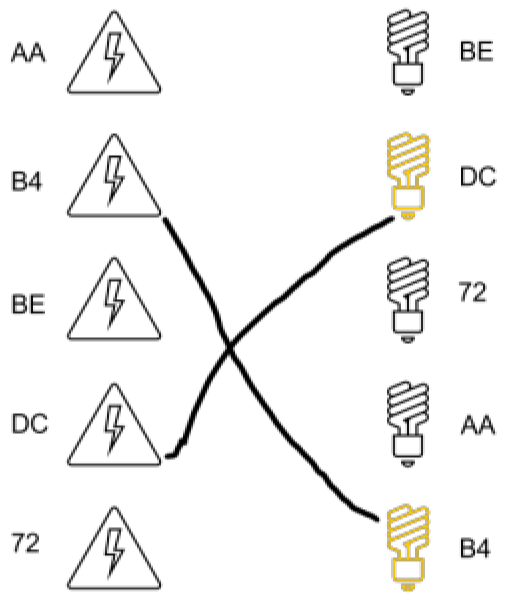


Fig 1. The figure shows the invalid wiring arrangement.

Dynamic Programming

Dynamic programming is one of the most powerful algorithmic techniques in computer science. This is an idea of solving a problem by first identifying and solving subproblems and then bringing the subproblems together in some manner to solve the larger problem.

In order to tackle this dynamic problem efficiently, I used the following five steps:

Visualization of the problem

We should find a way to visualize the given problem, which is a great way to see the connections and underlying patterns. Generally, we use a directed acyclic graph to visualize such dynamic problems.

Find an appropriate subproblem

Dividing the problem into smaller “subproblems”, which are later used to find out the solution for the original problem, simplifies the process.

Find relationships among the subproblems

We should find an appropriate relationship between the subproblems, which can help us to find out the solution to a particular subproblem.

Generalization of the relationship

After finding a reasonable relationship among the subproblems, we should generalize that.

Implementation of solving subproblems in order

Implementing a dynamic programming solution is just a matter of solving the subproblems in the appropriate order. Most important thing is that before solving any subproblem all the necessary prerequisite subproblems have been solved.

Implementation of code

The solution to this given problem statement is implemented in Java programming language, in OpenJDK-19 environment using IntelliJ IDEA Community Edition 2022.3.

In order to implement this solution, only one class is used, having only the void `main(String[] args)` method.

Initialization of Scanner and variables

First of all `new Scanner(System.in).useDelimiter("\n")` is initialized in order to read input from the keyboard. We read three inputs from the keyboard, by prompting messages for the user using `System.out.print()` method.

Three inputs include the number of electric outlets which are stored in the `num` variable and an array which contains the 2-digit hex codes for the electric outlets and electric bulbs which are stored in `inputOutlets` and `inputBulbs` variables respectively.

The hex codes for the electric outlets and electric bulbs are read as a String from the keyboard, which is processed later and hex values are stored in `Integer[] outlets` and `Integer[] bulbs` arrays respectively. Alongside, the given input is also checked if it's valid or not i.e. either the user didn't give the n hex values or the given hex value is not a 2-digit hex value, if not the user is then asked to give a valid input. The following code snippet shows the implementation:

```
Integer[] outlets = new Integer[num];
Integer[] bulbs = new Integer[num];

System.out.println("Now give the list of " + num + " 2-digit hex codes that
represent the order from top to bottom")

do {
      flag = false;
      System.out.print("Enter the codes of " + num + " electric outlets:
");
      String inputOutlets = sc.next();
      outletsInput = inputOutlets.split(" ");
      for (int i = 0; i < outletsInput.length; i++) {
            outlets[i] = Integer.parseInt(outletsInput[i], 16);
            if (outlets[i] > 255) {
                    flag = true;
                }
      }
} while ((outletsInput.length != num) || flag);

do {
      flag = false;
        System.out.print("Enter the codes of " + num + " electric bulbs:
");
        String inputBulbs = sc.next();
        bulbsInput = inputBulbs.split(" ");
        for (int i = 0; i < bulbsInput.length; i++) {
            bulbs[i] = Integer.parseInt(bulbsInput[i], 16);
                if (bulbs[i] > 255) {
                    flag = true;
            }
        }
} while ((bulbsInput.length != num) || flag);
```

Creation of `outletsToBulbs` sequence

Later a sequence is created, in which indices represent electric outlets and the values represent the corresponding electric bulbs i.e. a particular outlet can turn ON one and only one bulb containing the same hex code. The implementation is shown in the following code snippet:

```
Integer[] outletsToBulbs = new Integer[num];
```

```
for (int i = 0; i < num; i++) {
      for (int j = 0; j < num; j++) {
            if (Objects.equals(outlets[i], bulbs[j])) {
                    outletsToBulbs[i] = j;
                }
        }
}
```

Implementation of Dynamic Programming solution

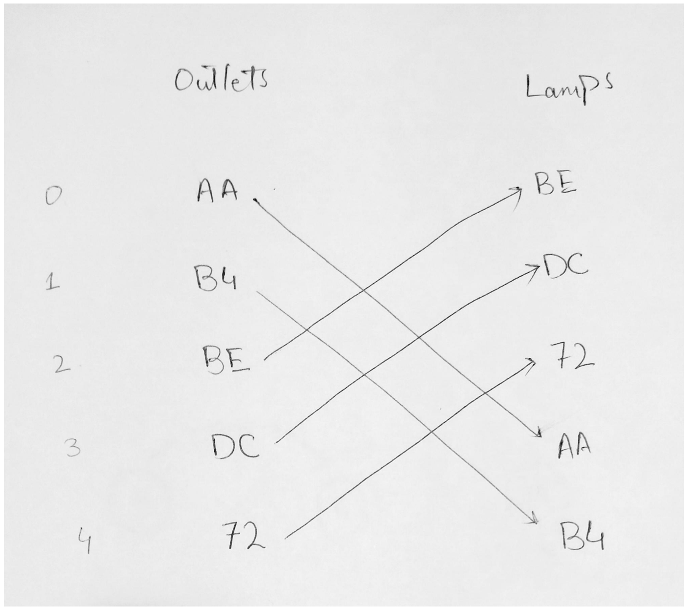
Then the dynamic programming solution is implemented to the given problem statement, which will be discussed in the next section in detail.

Printing the result on screen

In the end, using the data stored in the memoization variables declared during the dynamic programming solution we get the highest increasing subsequence in an array list `ArrayList<String> result`, which is our result, and this result is prompted on the output screen.

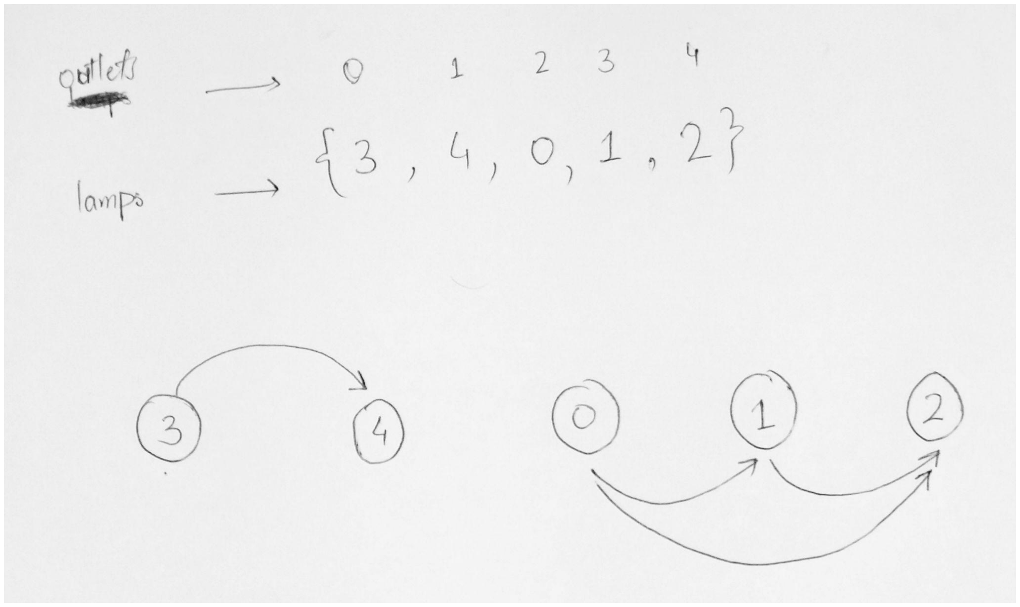
Dynamic Programming Solution

In the first step the problem is visualized using a graphic representation for the creation of the `outletsToBulbs` sequence.



This representation resulted in a sequence where the indices of the sequence represent the lamps while the values represent the corresponding lamps which can be turned ON by the

lamp. Now to figure out the desired result we are only supposed to get the longest increasing subsequence (LIS).
The sequence is then visualized in a directed acyclic graph where we construct a directed edge from one node to the other if the mode on right contains the larger value. This shows that each increasing subsequence is just another path in the graph.



As the next step, we define our subproblem for each node present in the graph. Hence, the subproblem at an index k will be the `LIS[k]` which is the `LIS`, where LIS is the length of the longest increasing subsequence, ending at index k. In our case, the `LIS[1] = 2`.

Then we establish a generalized relationship between the subproblems as follows:

$$\mathrm{LIS}[n] = 1 + \max\{\mathrm{LIS}[k] \mid k < n, A[k] < A[n]\}$$

There are two arrays created for memoization for each step, one is `Integer[] seqList` which stores the length of the longest increasing subsequence `LIS[n]` for each node in the original sequence and the other one is `Integer[] hashIndices` which contains the index of the previous node in the subsequence for each node. The following snippet shows the implementation:

```
for (int i = 0; i < seqList.length; i++) {
          hashIndices[i] = i;
          ArrayList<Integer> subprograms = new ArrayList<>();
            for (int k = 0; k < i; k++) {
            if (outletsToBulbs[k] < outletsToBulbs[i]) {
```

```
                        subprograms.add(seqList[k]);
                        if (1+seqList[k] > seqList[i]) {
                            hashIndices[i] = k;
                        }
                    }
            }
             SeqList[i] = 1 + Collections.max(subprograms);
}
```

The implementation of the above block of code is enclosed in the **`try-catch`** block because during the first iteration `subproblem` ArrayList is empty, hence it raises `NoSuchElementException`.

Also, we keep track of the index of the longest increasing subsequence in variable `lastIndex`. For each iteration, we check if the `LIS` on the current node is maximum or not, if yes then we update the variable `lastIndex`.

```
if (seqList[i] > maxSubSeqLen) {
      maxSubSeqLen = seqList[i];
      lastIndex = i;
}
```

Time Complexity

The two code snippets in lines 17-28 and 30-41, have the same logic, where n 2-digit hexadecimal numbers are converted into decimal which takes O(n) since we are iterating for loop n times.

For the code snippet in lines 43-51, initializing variables and arrays takes O(1) for each while filling the array takes O(n).

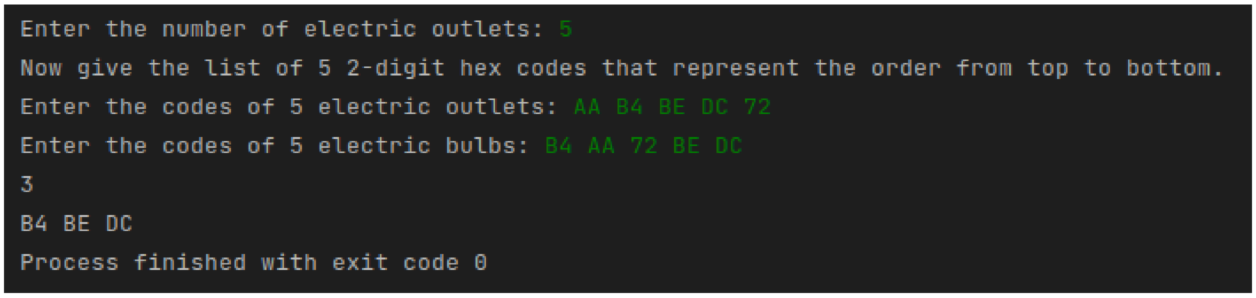
For the code snippet in lines 61-81, the total number of times this code executes is 1 + 2 + 3 + 4 + …. + n. If we sum till n, by mathematical formula it equals n(n+1)/2 which results in O(n^2).

To print the list, we are iterating for loop n times which is O(n).

Overall, the time complexity for the provided solution is O(n^2) since it is the dominant one.

Results

The proposed implementation for the given problem statement is very efficient and gives the correct output for all use cases. As an example following figure shows the output for a program run:



```
Enter the number of electric outlets: 5
Now give the list of 5 2-digit hex codes that represent the order from top to bottom.
Enter the codes of 5 electric outlets: AA B4 BE DC 72
Enter the codes of 5 electric bulbs: B4 AA 72 BE DC
3
B4 BE DC
Process finished with exit code 0
```

References

- https://www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296