# Experiment – 3.2

**Student Name: Avinash Jena** **UID: 20BCS2690**

**Branch: CSE** **Section/Group: 707/B**

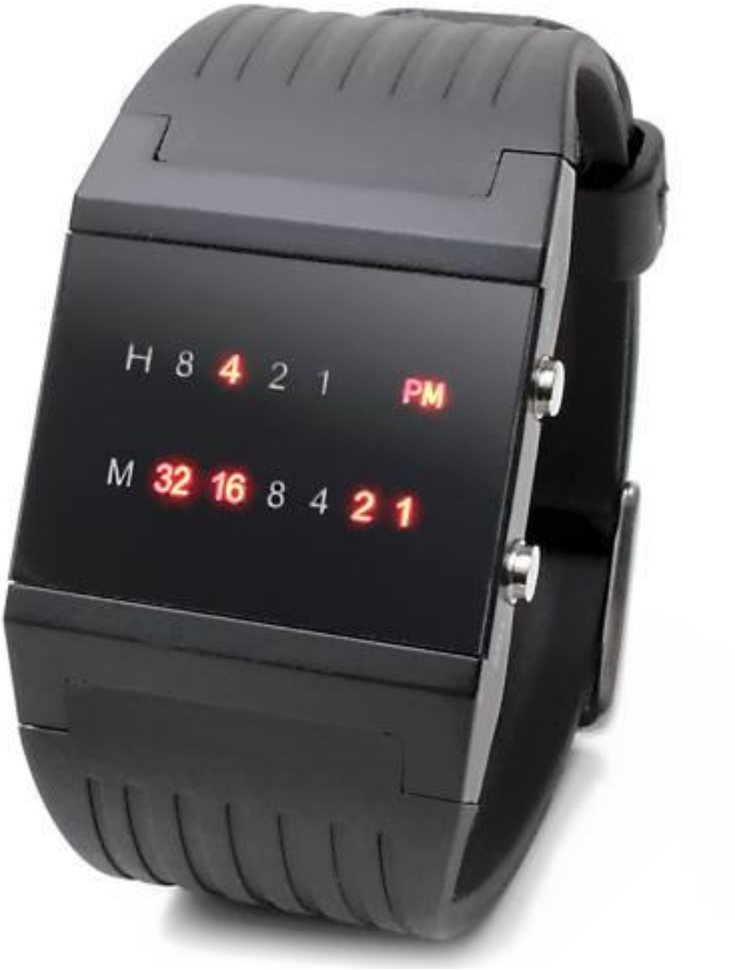
**Subject Name: Competitive Coding II** **Subject Code: 20CSP-351**

## Aim: Binary Watch

## Objective:

A binary watch has 4 LEDs on the top to represent the hours (0-11), and 6 LEDs on the bottom to represent the minutes (0-59). Each LED represents a zero or one, with the least significant bit on the right.

For example, the below binary watch reads "4:51".



Given an integer turnedOn which represents the number of LEDs that are currently on (ignoring the PM), return all possible times the watch could represent. You may return the answer in any order.

The hour must not contain a leading zero.

For example, "01:00" is not valid. It should be "1:00".

The minute must be consist of two digits and may contain a leading zero.

For example, "10:2" is not valid. It should be "10:02".

**Example 1:**

Input: turnedOn = 1

Output: ["0:01","0:02","0:04","0:08","0:16","0:32","1:00","2:00","4:00","8:00"]

**Example 2:**

Input: turnedOn = 9

Output: []

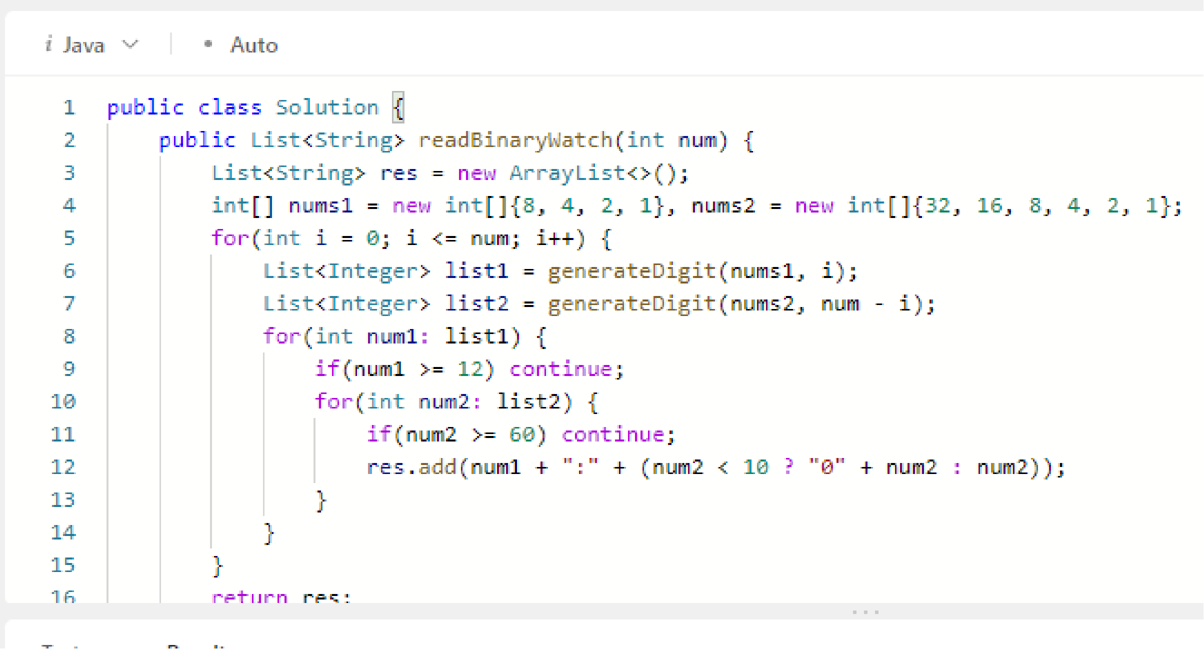
**Constraints:**

0 <= turnedOn <= 10

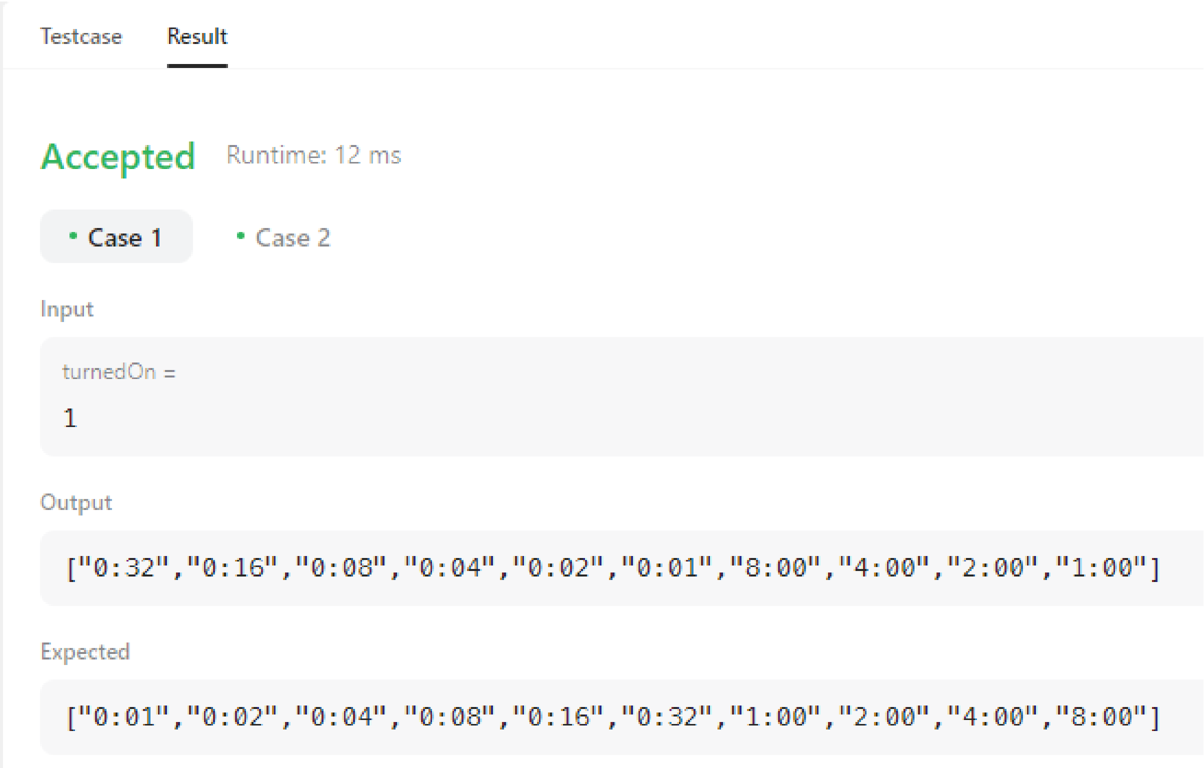
## Code:

```
public class Solution {
    public List<String> readBinaryWatch(int num) {
        List<String> res = new ArrayList<>();
        int[] nums1 = new int[]{8, 4, 2, 1}, nums2 = new int[]{32, 16, 8, 4, 2, 1};
        for(int i = 0; i <= num; i++) {
            List<Integer> list1 = generateDigit(nums1, i);
            List<Integer> list2 = generateDigit(nums2, num - i);
            for(int num1: list1) {
                if(num1 >= 12) continue;
                for(int num2: list2) {
                    if(num2 >= 60) continue;
                    res.add(num1 + ":" + (num2 < 10 ? "0" + num2 : num2));
                }
            }
        }
        return res;
    }

    private List<Integer> generateDigit(int[] nums, int count) {
        List<Integer> res = new ArrayList<>();
        generateDigitHelper(nums, count, 0, 0, res);
        return res;
    }

    private void generateDigitHelper(int[] nums, int count, int pos, int sum, List<Integer> res) {
        if(count == 0) {
            res.add(sum);
            return;
        }

        for(int i = pos; i < nums.length; i++) {
            generateDigitHelper(nums, count - 1, i + 1, sum + nums[i], res);
        }
    }
}
```

## Output:



```java
public class Solution {
    public List<String> readBinaryWatch(int num) {
        List<String> res = new ArrayList<>();
        int[] nums1 = new int[]{8, 4, 2, 1}, nums2 = new int[]{32, 16, 8, 4, 2, 1};
        for(int i = 0; i <= num; i++) {
            List<Integer> list1 = generateDigit(nums1, i);
            List<Integer> list2 = generateDigit(nums2, num - i);
            for(int num1: list1) {
                if(num1 >= 12) continue;
                for(int num2: list2) {
                    if(num2 >= 60) continue;
                    res.add(num1 + ":" + (num2 < 10 ? "0" + num2 : num2));
                }
            }
        }
        return res;
```



Accepted Runtime: 12 ms

Input

turnedOn =

1

Output

```
["0:32","0:16","0:08","0:04","0:02","0:01","8:00","4:00","2:00","1:00"]
```

Expected

```
["0:01","0:02","0:04","0:08","0:16","0:32","1:00","2:00","4:00","8:00"]
```

## Aim: Word Ladder II

## Objective:

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter. Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList. sk == endWord Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

### Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

Explanation: There are 2 shortest transformation sequences:

"hit" -> "hot" -> "dot" -> "dog" -> "cog"

"hit" -> "hot" -> "lot" -> "log" -> "cog"

### Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

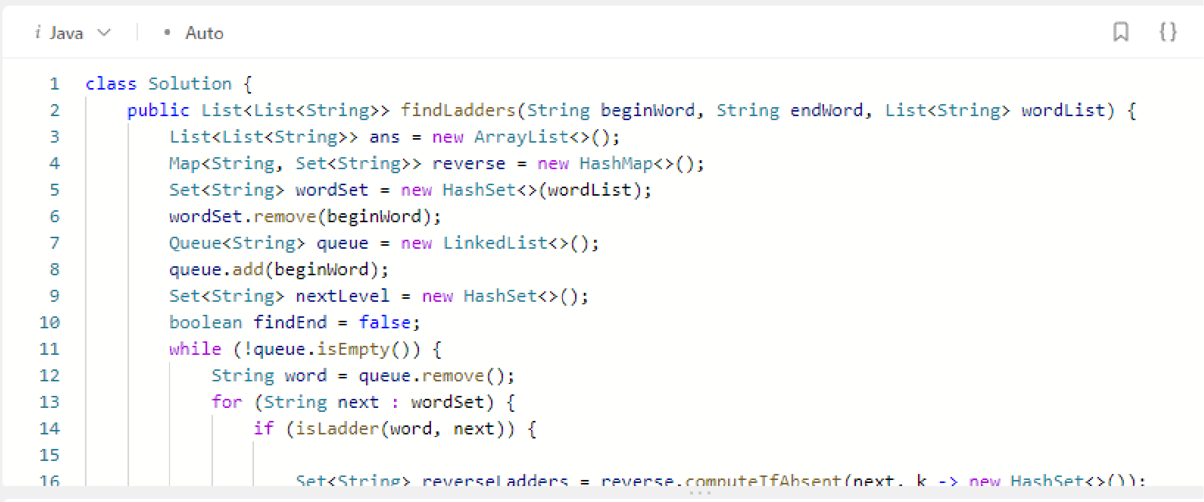
Output: []

Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

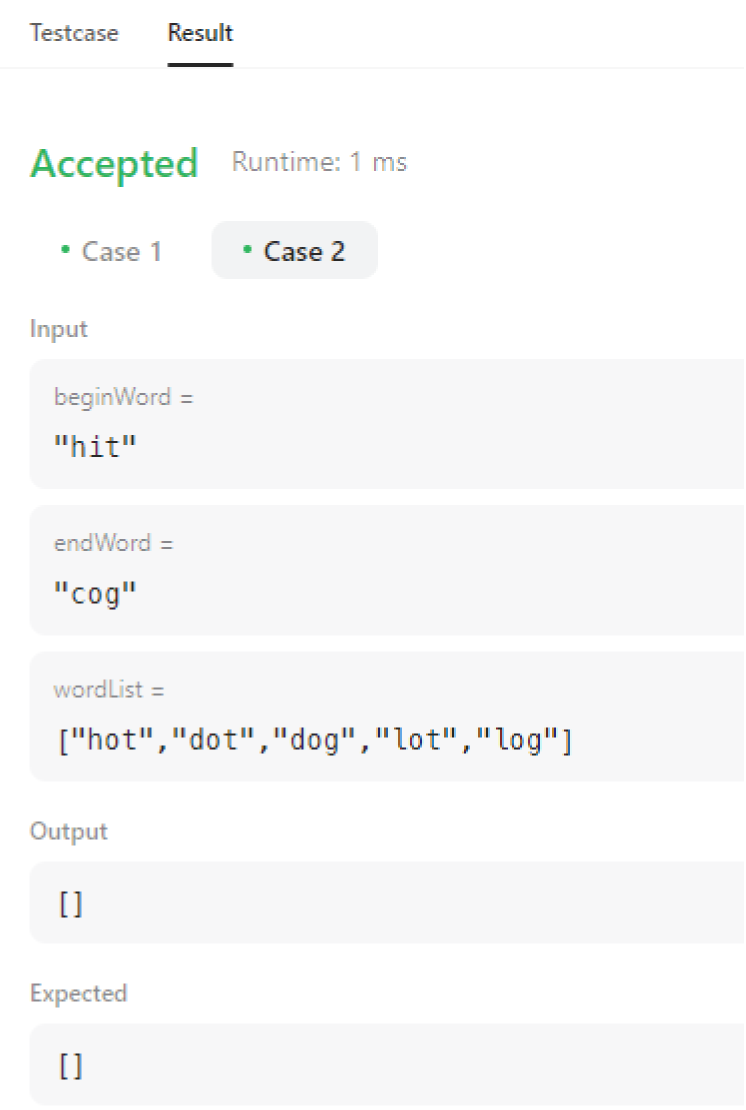
**Code:**

```
class Solution {
    public List<List<String>> findLadders(String beginWord, String
endWord, List<String> wordList) {
        List<List<String>> ans = new ArrayList<>();
        Map<String, Set<String>> reverse = new HashMap<>();
        Set<String> wordSet = new HashSet<>(wordList);
        wordSet.remove(beginWord);
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        Set<String> nextLevel = new HashSet<>();
        boolean findEnd = false;
        while (!queue.isEmpty()) {
            String word = queue.remove();
            for (String next : wordSet) {
                if (isLadder(word, next)) {

                    Set<String> reverseLadders = reverse.computeIfAbsent(next, k
-> new HashSet<>());
                    reverseLadders.add(word);
                    if (endWord.equals(next)) {
                        findEnd = true;
                    }
                    nextLevel.add(next);
                }
            }
            if (queue.isEmpty()) {
                if (findEnd) break;
                queue.addAll(nextLevel);
                wordSet.removeAll(nextLevel);
                nextLevel.clear();
            }
        }
        if (!findEnd) return ans;
        Set<String> path = new LinkedHashSet<>();
        path.add(endWord);
        findPath(endWord, beginWord, reverse, ans, path);
        return ans;
    }


    private void findPath(String endWord, String beginWord, Map<String,
Set<String>> graph,
                                 List<List<String>> ans, Set<String> path) {
        Set<String> next = graph.get(endWord);
        if (next == null) return;
        for (String word : next) {
            path.add(word);
            if (beginWord.equals(word)) {
                List<String> shortestPath = new ArrayList<>(path);
                Collections.reverse(shortestPath);
                ans.add(shortestPath);
            } else {
                findPath(word, beginWord, graph, ans, path);
            }
            path.remove(word);
        }
    }

    private boolean isLadder(String s, String t) {
        if (s.length() != t.length()) return false;
        int diffCount = 0;
        int n = s.length();
        for (int i = 0; i < n; i++) {
            if (s.charAt(i) != t.charAt(i)) diffCount++;
            if (diffCount > 1) return false;
        }
        return diffCount == 1;
    }
}
```

## Output:



```
class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
        List<List<String>> ans = new ArrayList<>();
        Map<String, Set<String>> reverse = new HashMap<>();
        Set<String> wordSet = new HashSet<>(wordList);
        wordSet.remove(beginWord);
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        Set<String> nextLevel = new HashSet<>();
        boolean findEnd = false;
        while (!queue.isEmpty()) {
            String word = queue.remove();
            for (String next : wordSet) {
                if (isLadder(word, next)) {

                    Set<String> reverseLadders = reverse.computeIfAbsent(next, k -> new HashSet<>());
```



Accepted Runtime: 1 ms

Input

beginWord =
"hit"

endWord =
"cog"

wordList =
["hot","dot","dog","lot","log"]

Output

[]

Expected

[]