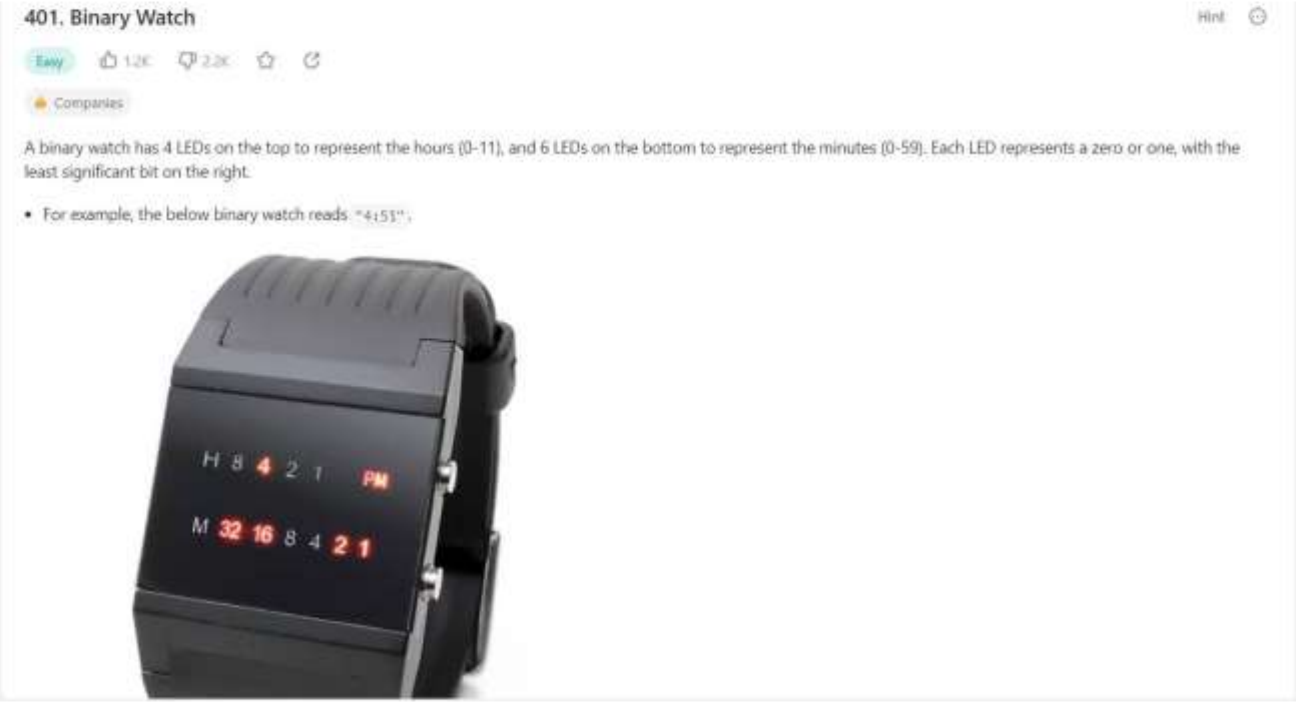
Experiment 3.2

Student Name: Harshit Raj **UID: 20BCS9266**
Branch: CSE **Section/Group: 608/A**
Semester: 6 **Date of Performance: 03/05/2023**
Subject Name: Competitive Coding-II **Subject Code:** 20CSP-351

Aim: To demonstrate the concept of Backtracking

Objective: **Binary Watch**: https://leetcode.com/problems/binary-watch/



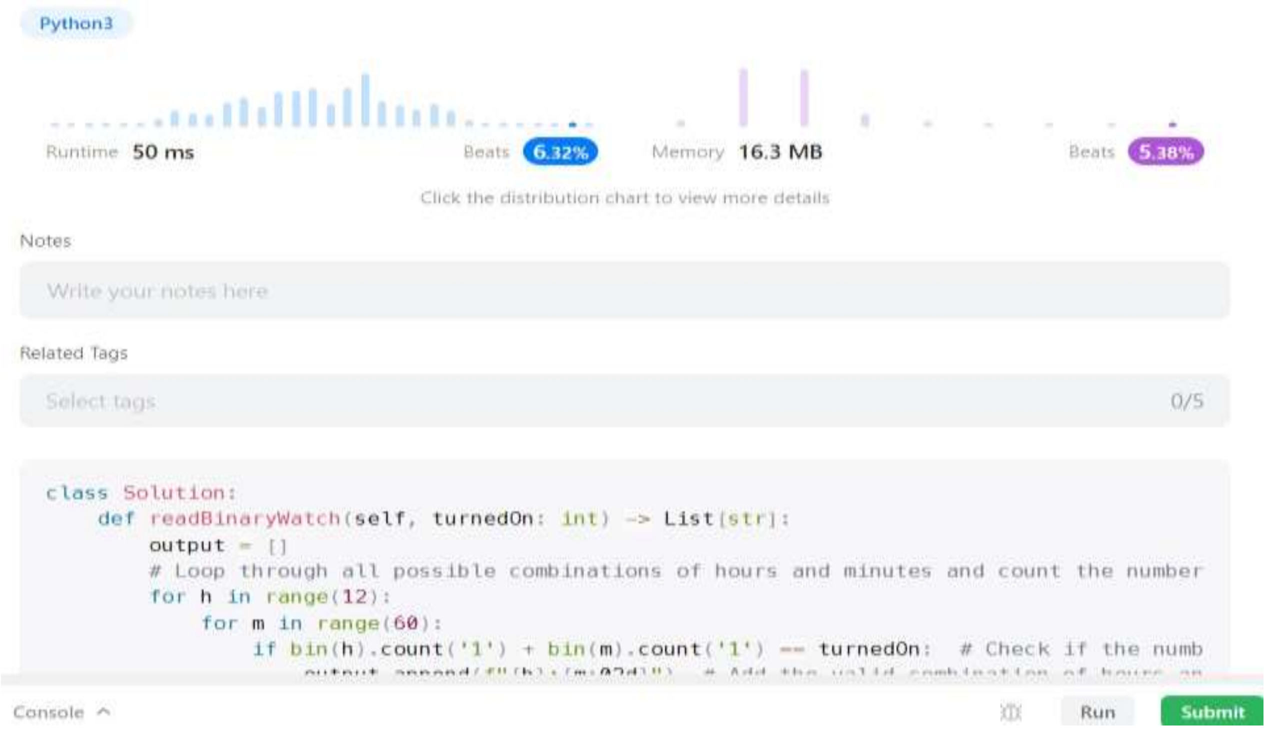
401. Binary Watch

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".

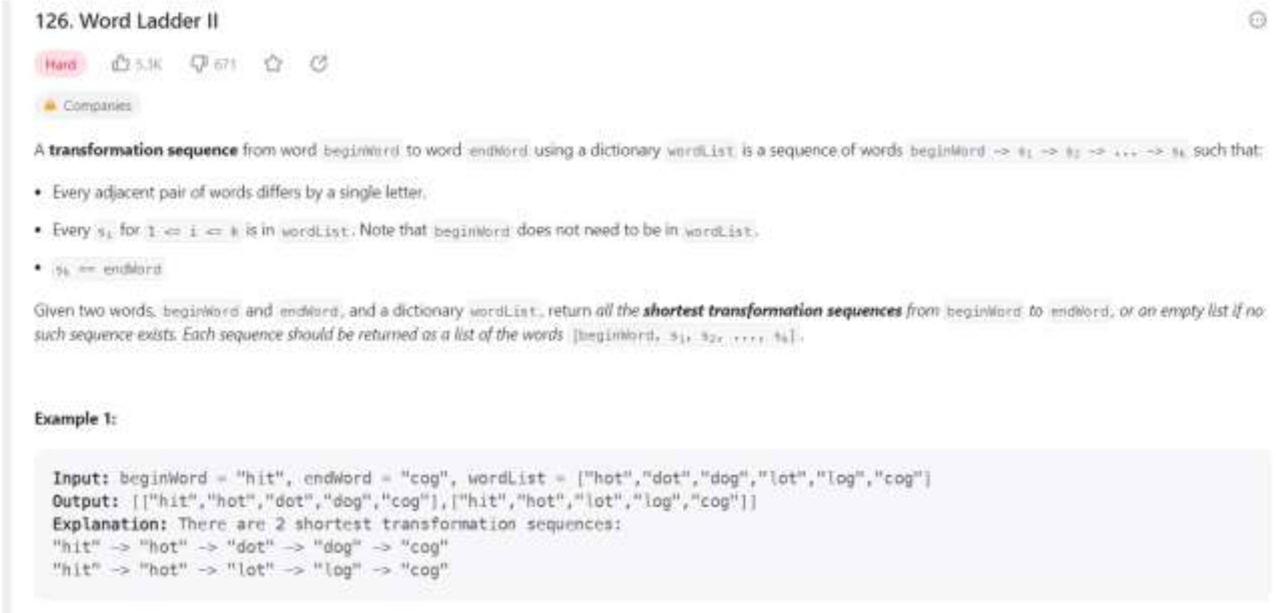
Code:

```
class Solution:       def readBinaryWatch(self, turnedOn:
int) -> List[str]:
        output = []
        # Loop through all possible combinations of hours and count the
number of set bits         for h in range(12):             for m in range(60):
                if bin(h).count('1') + bin(m).count('1') == turnedOn:  # Check if the number
of     set     bits     in     hours     and     minutes     equals     the     target     number
output.append(f"{h}:{m:02d}")   # Add the valid combination of hours and minutes to the
output list         return output
```



```
class Solution:
    def readBinaryWatch(self, turnedOn: int) -> List[str]:
        output = []
        # Loop through all possible combinations of hours and minutes and count the number
        for h in range(12):
            for m in range(60):
                if bin(h).count('1') + bin(m).count('1') == turnedOn:  # Check if the numb
                    output.append(f"{h}:{m:02d}")  # Add the valid combination of hours an
```

Word Letter II: https://leetcode.com/problems/word-ladder-ii/



126. Word Ladder II

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:

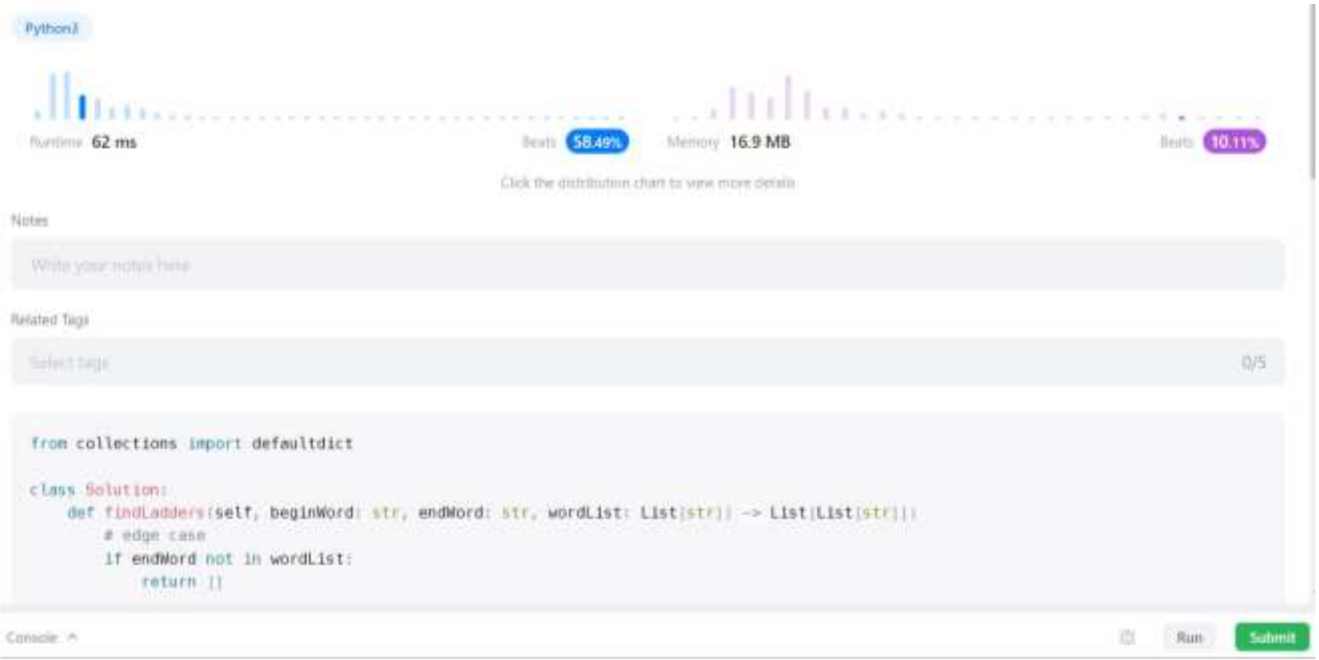
```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
```

Code:

```
from collections import defaultdict
```

```
 class
Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) ->
List[List[str]]:         # edge case         if endWord not in wordList:
            return []

        # 1) build neighbor list for first bfs
if      beginWord      not      in      wordList:
wordList.append(beginWord)           unseen  =
set(wordList)       word_size = len(beginWord)
neighbors = defaultdict(list)          for word
in  wordList:                          for  i  in
range(word_size):
                neighbors[f'{word[:i]}*{word[i+1:]}'].append(word)

        # 2) do first bfs and build reversed neighbors list for second bfs
reverse_neighbors  =  defaultdict(list)               n_t_h  =  [beginWord]
unseen.remove(beginWord)         while n_t_h:
            new_seen = set()           for
word in n_t_h:                         for i in
range(word_size):
                    for neighbor in neighbors[f'{word[:i]}*{word[i+1:]}']:
                        if neighbor in unseen:
                            reverse_neighbors[neighbor].append(word)
new_seen.add(neighbor)                         n_t_h  =  list(new_seen)
unseen -= new_seen              if reverse_neighbors[endWord]:
                break

        # if endWord does not have reversed neigbors it is not reachable so return
empty list         if not reverse_neighbors[endWord]:
            return []

        #  3)  do  second  bfs
paths     =     [[endWord]]
while True:
            new_paths = []
            for path in paths:
                last_node = path[-1]              for reverse_neighbor
in reverse_neighbors[last_node]:
                    new_paths.append(path + [reverse_neighbor])
paths = new_paths              if paths[0][-1] == beginWord:
                break

        #  4)  reverse  the  paths
result = []          for path in
paths:           path.reverse()
result.append(path)
return result
```

Output:



Runtime 62 ms Beats 58.49%

Memory 16.9 MB Beats 10.11%

```
from collections import defaultdict

class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        # edge case
        if endWord not in wordList:
            return []
```