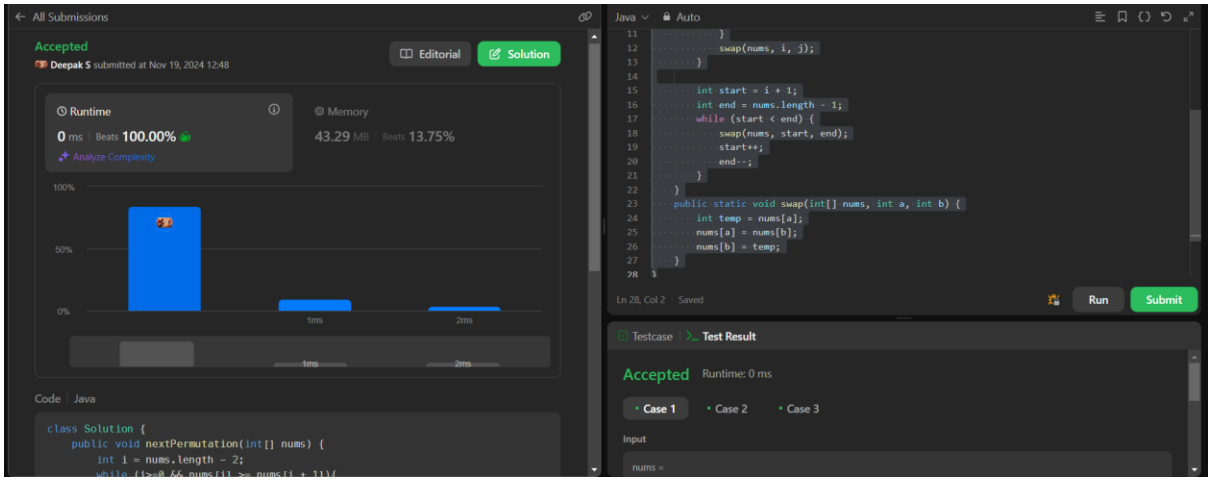
Next Permutation

Code:

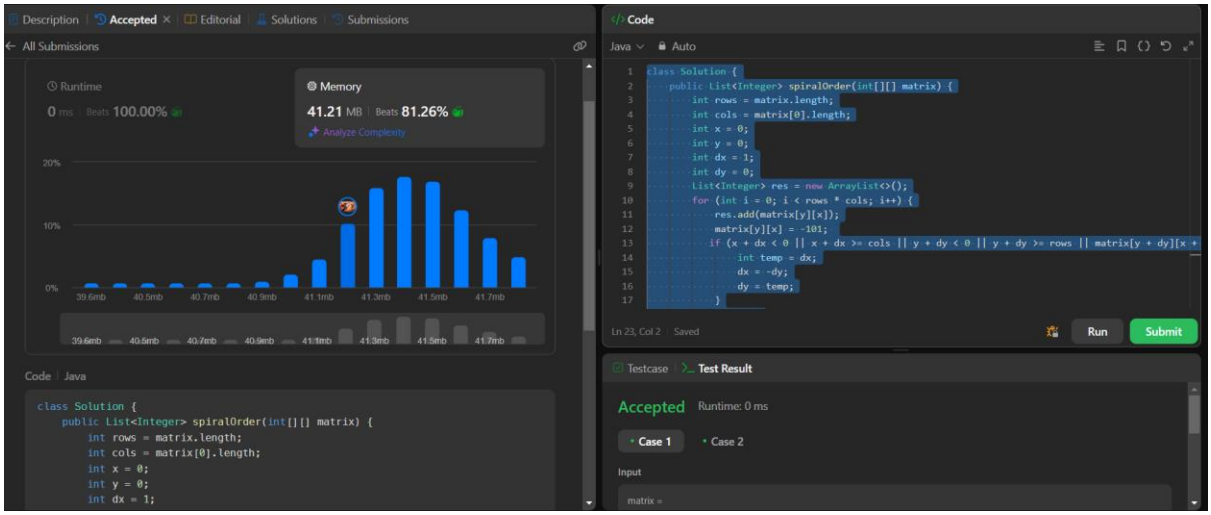
```
class Solution {

    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i>=0 && nums[i] >= nums[i + 1]){
            i--;
        }
        if (i != -1) {
            int j= nums.length-1;
            while (j>=0 && nums[i] >= nums[j]) {
                j--;
            }
            swap(nums, i, j);
        }

        int start = i + 1;
        int end = nums.length - 1;
        while (start < end) {
            swap(nums, start, end);
            start++;
            end--;
        }
    }
    public static void swap(int[] nums, int a, int b) {
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```



Spiral matrix

Code:

```
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        int x = 0;
        int y = 0;
        int dx = 1;
        int dy = 0;
        List<Integer> res = new ArrayList<>();
        for (int i = 0; i < rows * cols; i++) {
            res.add(matrix[y][x]);
            matrix[y][x] = -101;
           if (x + dx < 0 || x + dx >= cols || y + dy < 0 || y + dy >= rows ||
matrix[y + dy][x + dx] == -101) {
                int temp = dx;
                dx = -dy;
                dy = temp;
            }
            x += dx;
            y += dy;
        }
        return res;
    }
}
```

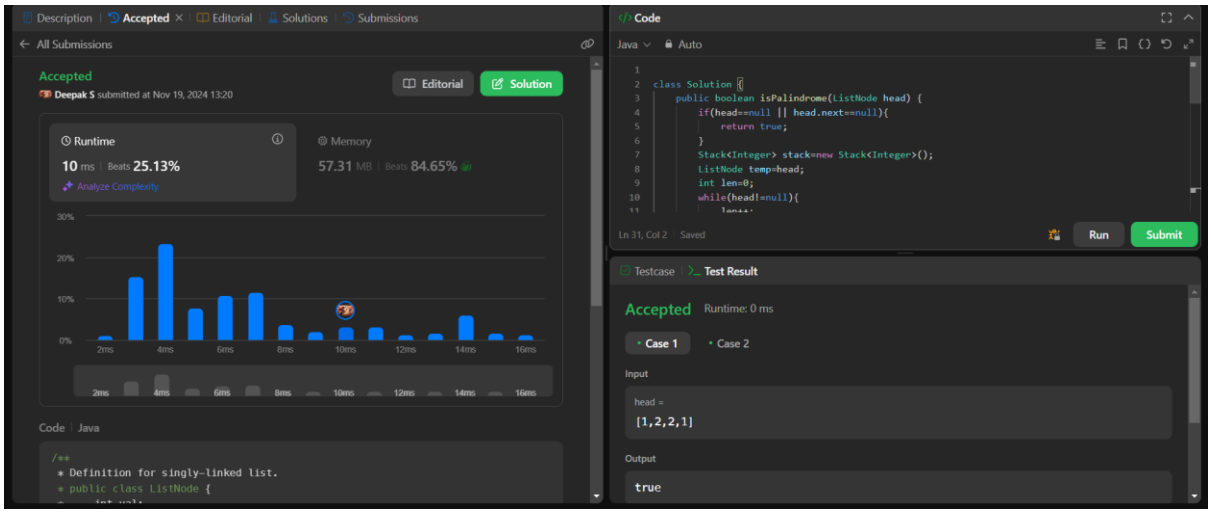


Palindrome linked list

Code:

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        if(head==null || head.next==null){
            return true;
        }
        Stack<Integer> stack=new Stack<Integer>();
        ListNode temp=head;
        int len=0;
        while(head!=null){
            len++;
            head=head.next;
        }
        int i=0;
        while(temp!=null && i<len/2){
            stack.push(temp.val);
            temp=temp.next;
            i++;
        }
        if((len%2)!=0){
            temp=temp.next;
        }
        while(temp!=null){
            if((!stack.isEmpty())&&stack.pop()!=temp.val){
                return false;
            }
            temp=temp.next;
        }
        return true;
    }
}
```

Output:

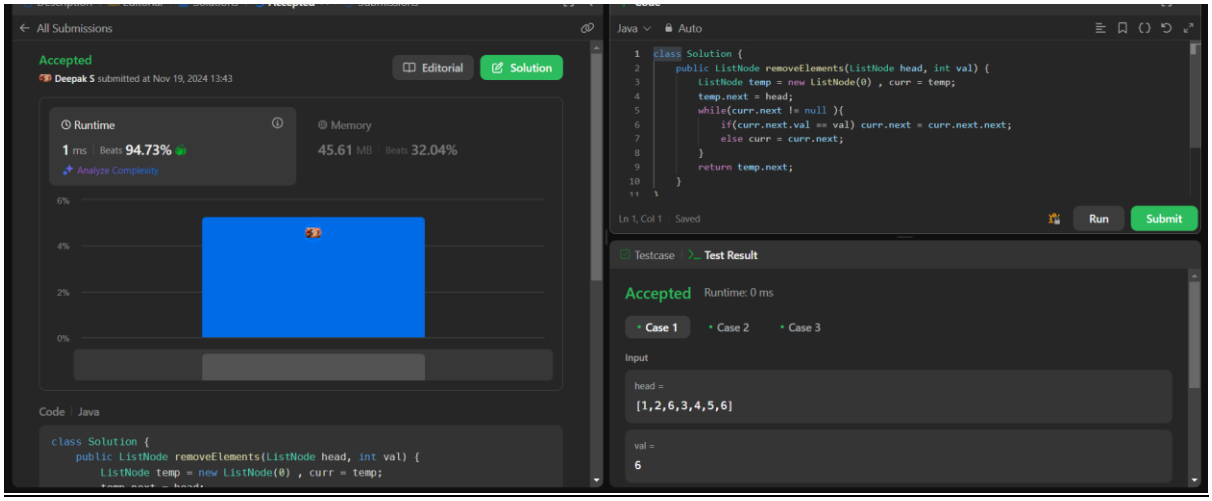


Remove linked list elements

Code:

```
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode temp = new ListNode(0) , curr = temp;
        temp.next = head;
        while(curr.next != null ){
            if(curr.next.val == val) curr.next = curr.next.next;
            else curr = curr.next;
        }
        return temp.next;
    }
}
```

Code:

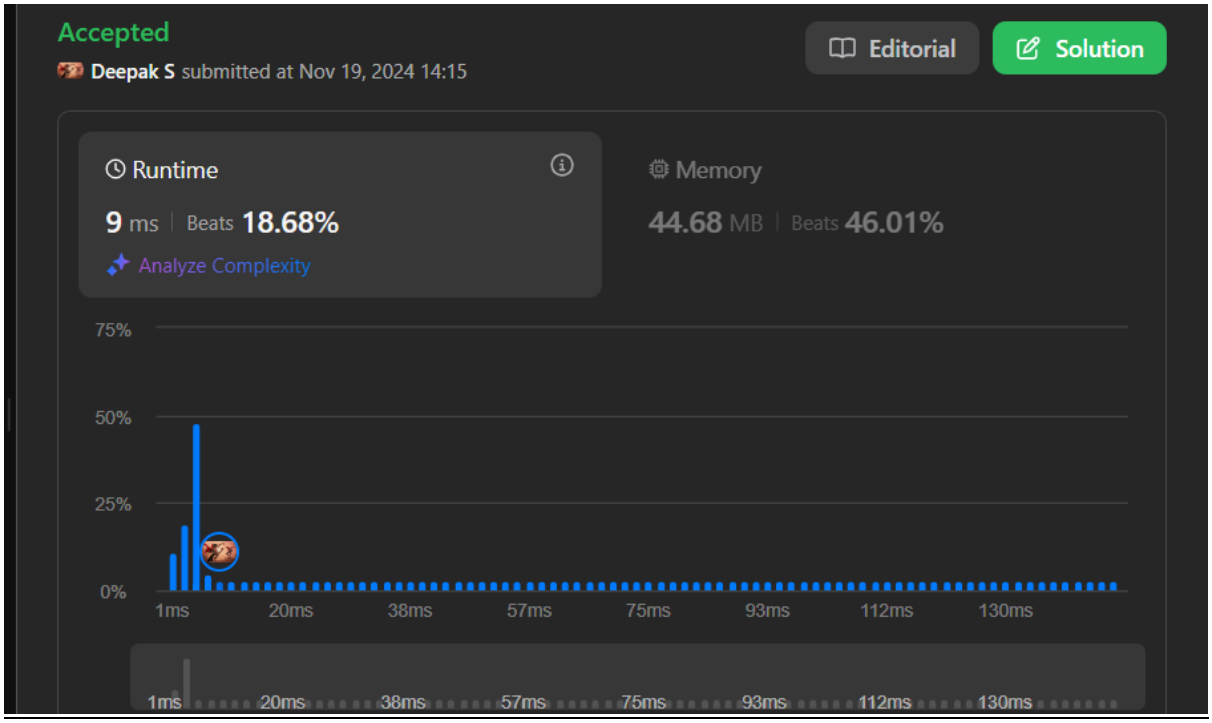


[Longest Substring Without Repeating Characters]

Code:

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int maxLength = 0;
        HashSet<Character> charSet = new HashSet<>();
        int left = 0;

        for (int right = 0; right < n; right++) {
            while (charSet.contains(s.charAt(right))) {
                charSet.remove(s.charAt(left));
                left++;
            }
            charSet.add(s.charAt(right));
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```

Output:

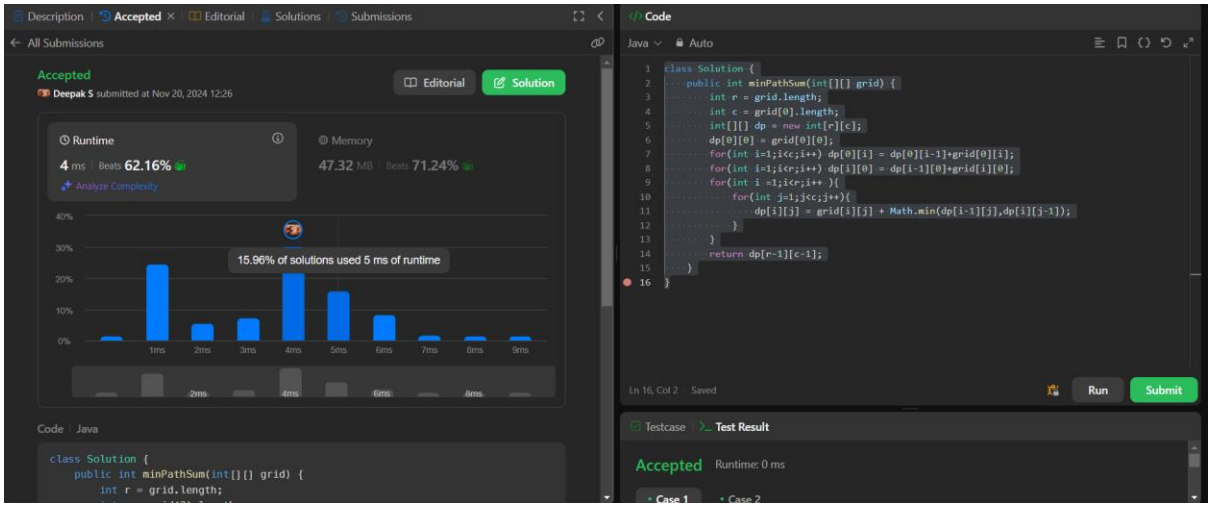


Minimum Path Sum

Code:

```
class Solution {
    public int minPathSum(int[][] grid) {
        int r = grid.length;
        int c = grid[0].length;
        int[][] dp = new int[r][c];
        dp[0][0] = grid[0][0];
        for(int i=1;i<c;i++) dp[0][i] = dp[0][i-1]+grid[0][i];
        for(int i=1;i<r;i++) dp[i][0] = dp[i-1][0]+grid[i][0];
        for(int i =1;i<r;i++ ){
            for(int j=1;j<c;j++){
                dp[i][j] = grid[i][j] + Math.min(dp[i-1][j],dp[i][j-1]);
            }
        }
        return dp[r-1][c-1];
    }
}
```

OUTPUT:

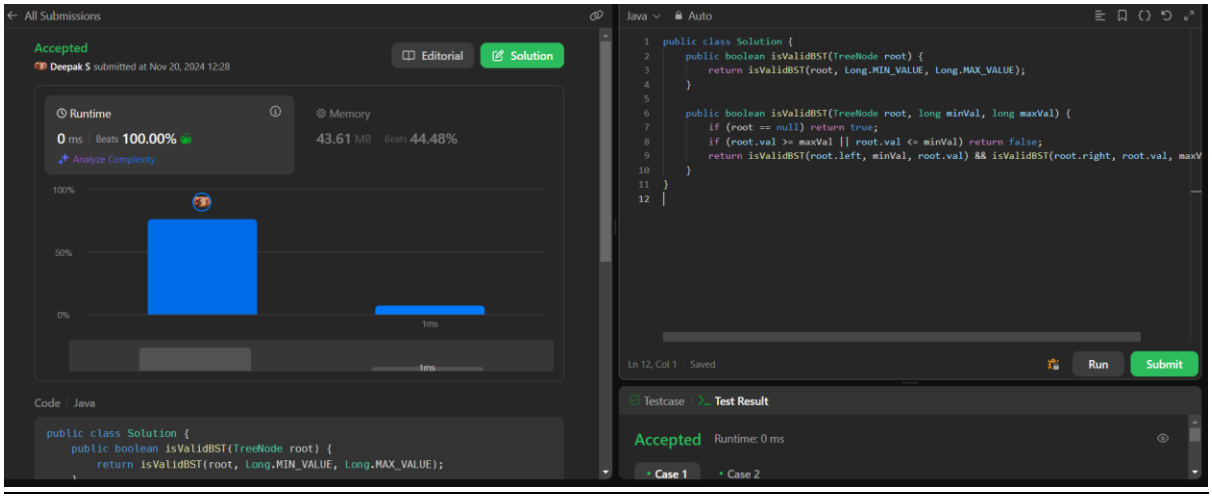


[Validate Binary Search Tree]

CODE:

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root, long minVal, long maxVal) {
        if (root == null) return true;
        if (root.val >= maxVal || root.val <= minVal) return false;
        return isValidBST(root.left, minVal, root.val) && isValidBST(root.right,
root.val, maxVal);
    }
}
```

Output:



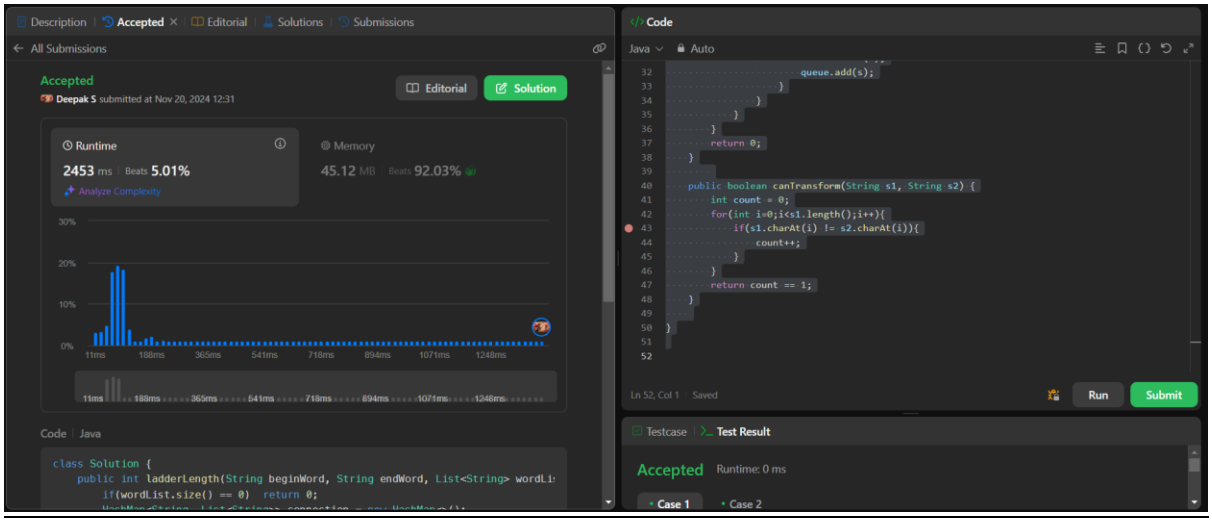
[Word Ladder](#)

Code:

```
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
        if(wordList.size() == 0)  return 0;
        HashMap<String, List<String>> connection = new HashMap<>();
        wordList.add(beginWord);
        for(String s : wordList) {
            connection.put(s, new ArrayList<String>());
        }
        for(String s1 : wordList) {
            for(String s2 : wordList) {
                if(canTransform(s1,s2)){
```

```
                    connection.get(s1).add(s2);
                    connection.get(s2).add(s1);
                }
            }
        }
        Queue<String> queue = new LinkedList();
        queue.add(beginWord);
        int dist = 0;
        Set<String> visited = new HashSet();
        while(!queue.isEmpty()){
            int size = queue.size();
            dist++;
            for(int i=0;i<size;i++){
                String cur = queue.poll();
                if(cur.equals(endWord)) {
                    return dist;
                }
                for(String s : connection.get(cur)) {
                    if(!visited.contains(s)) {
                        visited.add(s);
                        queue.add(s);
                    }
                }
            }
        }
        return 0;
    }
    public boolean canTransform(String s1, String s2) {
        int count = 0;
        for(int i=0;i<s1.length();i++){
            if(s1.charAt(i) != s2.charAt(i)){
                count++;
            }
        }
        return count == 1;
    }

}
```

Output:



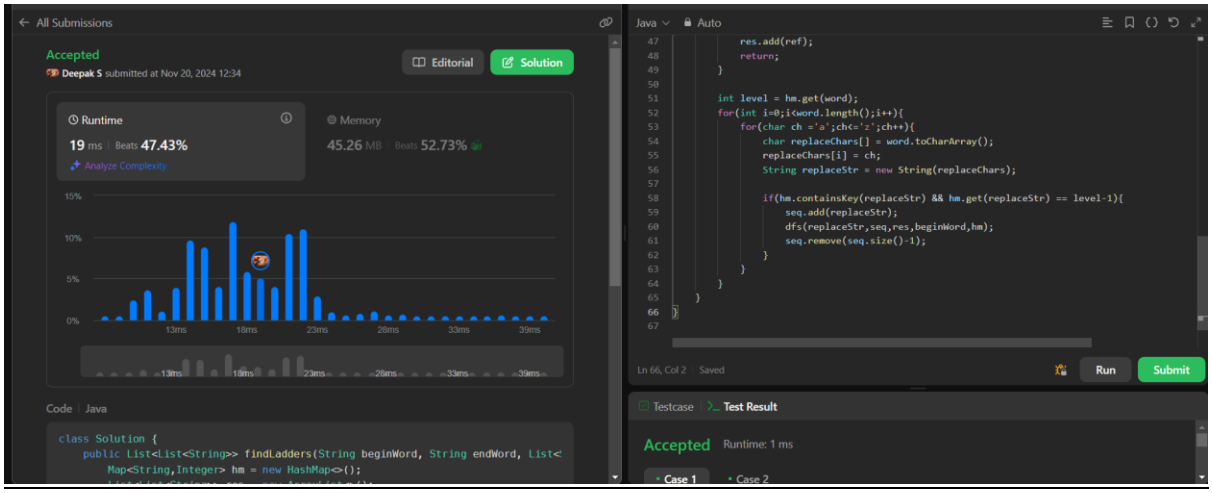
Word Ladder II

Code:

```
class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord,
List<String> wordList) {
        Map<String,Integer> hm = new HashMap<>();
        List<List<String>> res = new ArrayList<>();

        Queue<String> q = new LinkedList<>();
        q.add(beginWord);
        hm.put(beginWord,1);

        HashSet<String> hs = new HashSet<>();
        for(String w : wordList) hs.add(w);
        hs.remove(beginWord);
        while(!q.isEmpty()){
            String word = q.poll();
            if(word.equals(endWord)){
                break;
            }

            for(int i=0;i<word.length();i++){
                int level = hm.get(word);
                for(char ch='a';ch<='z';ch++){
                    char[] replaceChars = word.toCharArray();
                    replaceChars[i] = ch;
                    String replaceString = new String(replaceChars);

                    if(hs.contains(replaceString)){
                        q.add(replaceString);
                        hm.put(replaceString,level+1);
                        hs.remove(replaceString);
                    }
                }
            }
```

```
        }

        if(hm.containsKey(endWord) == true){
            List<String> seq = new ArrayList<>();
            seq.add(endWord);
            dfs(endWord,seq,res,beginWord,hm);
        }
        return res;
    }

    public void dfs(String word,List<String> seq,List<List<String>> res,String
beginWord,Map<String,Integer> hm){
        if(word.equals(beginWord)){
            List<String> ref = new ArrayList<>(seq);
            Collections.reverse(ref);
            res.add(ref);
            return;
        }

        int level = hm.get(word);
        for(int i=0;i<word.length();i++){
            for(char ch ='a';ch<='z';ch++){
                char replaceChars[] = word.toCharArray();
                replaceChars[i] = ch;
                String replaceStr = new String(replaceChars);

                if(hm.containsKey(replaceStr) && hm.get(replaceStr) == level-1){
                    seq.add(replaceStr);
                    dfs(replaceStr,seq,res,beginWord,hm);
                    seq.remove(seq.size()-1);
                }
            }
        }
    }
}
```

Output:



Course Schedule

CODE:

```
class Solution {
    public boolean canFinish(int n, int[][] prerequisites) {
        List<Integer>[] adj = new List[n];
        int[] indegree = new int[n];
        List<Integer> ans = new ArrayList<>();

        for (int[] pair : prerequisites) {
            int course = pair[0];
            int prerequisite = pair[1];
            if (adj[prerequisite] == null) {
                adj[prerequisite] = new ArrayList<>();
            }
            adj[prerequisite].add(course);
            indegree[course]++;
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }

        while (!queue.isEmpty()) {
            int current = queue.poll();
            ans.add(current);

            if (adj[current] != null) {
                for (int next : adj[current]) {
                    indegree[next]--;
                    if (indegree[next] == 0) {
                        queue.offer(next);
```

```
                    }
                }
            }
        }

        return ans.size() == n;
    }
}
```

OUTPUT: