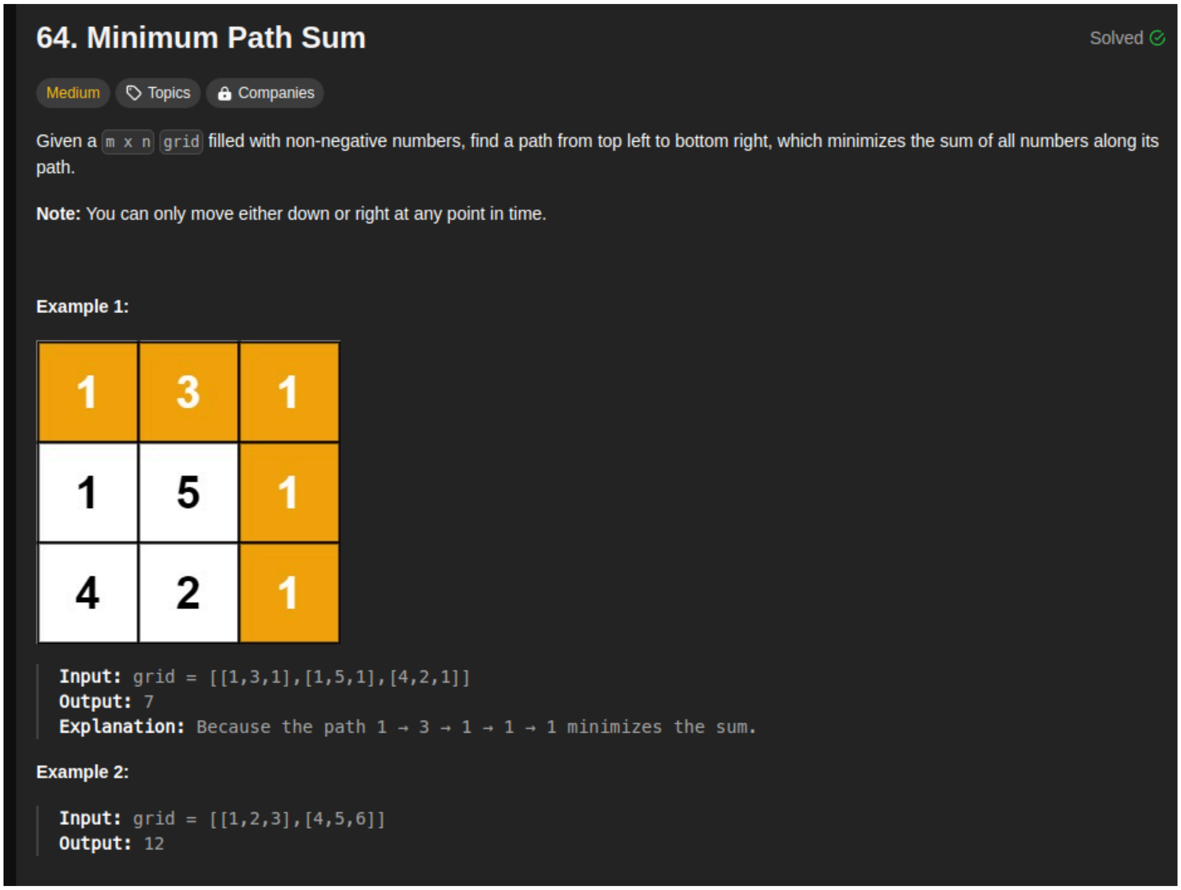


64. Minimum Path Sum

Solved ✓

Medium | Topics | 🔒 Companies

Given a `m x n` `grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:

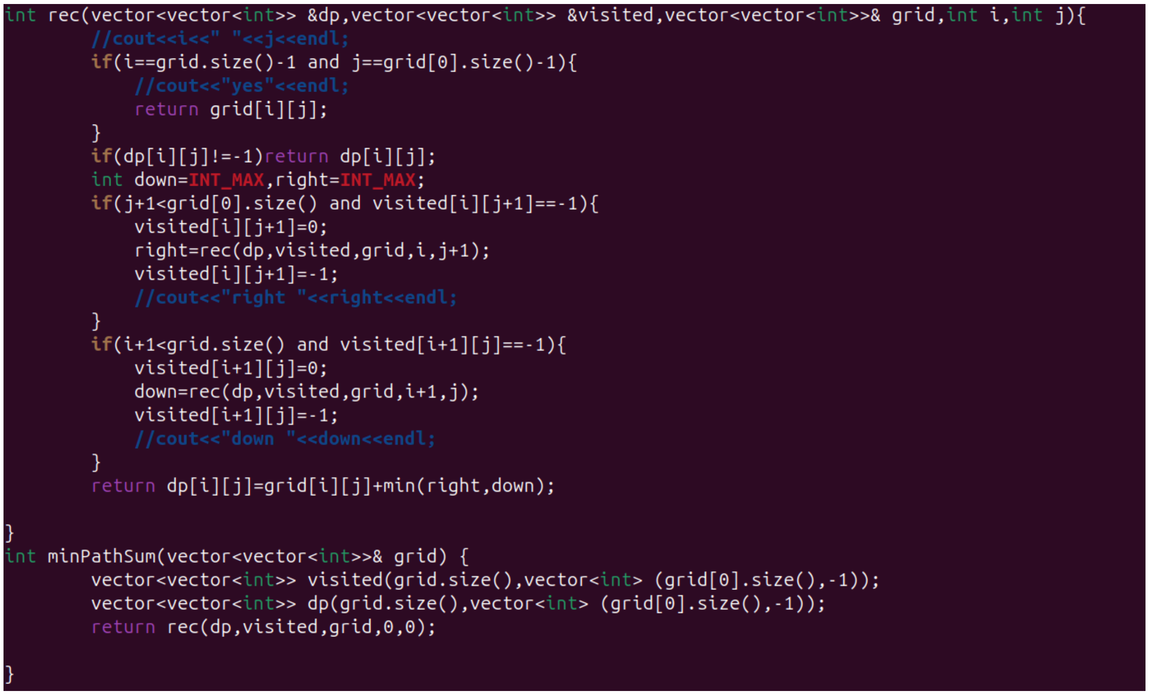
1	3	1
1	5	1
4	2	1

```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

Example 2:

```
Input: grid = [[1,2,3],[4,5,6]]
Output: 12
```

Code:



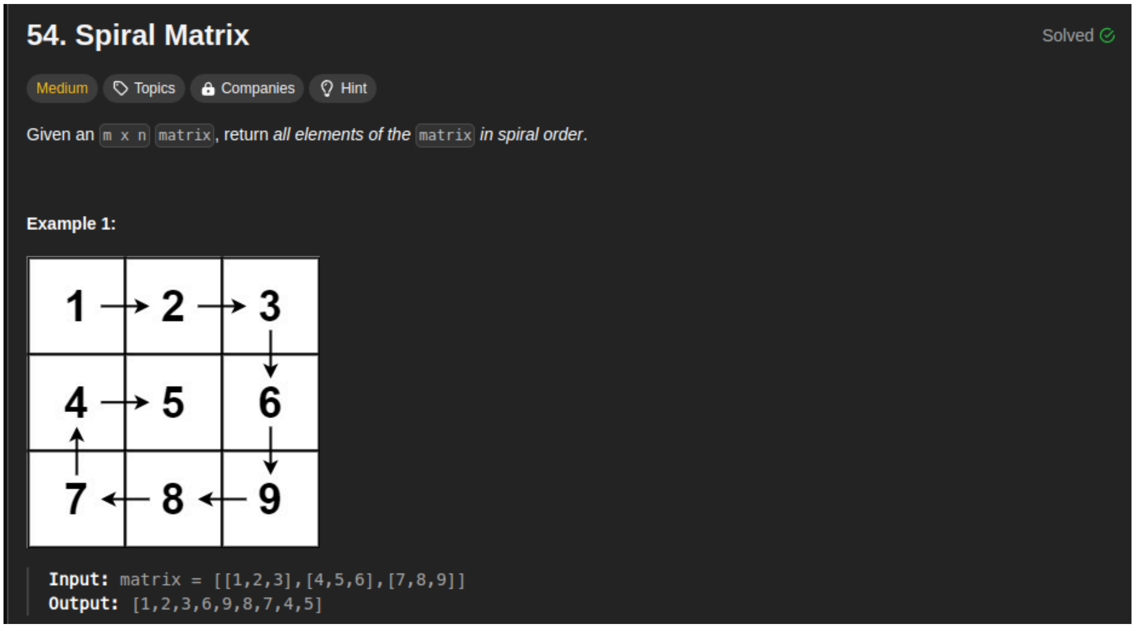
```cpp
int rec(vector<vector<int>> &dp,vector<vector<int>> &visited,vector<vector<int>>& grid,int i,int j){
        //cout<<i<<" "<<j<<endl;
        if(i==grid.size()-1 and j==grid[0].size()-1){
            //cout<<"yes"<<endl;
            return grid[i][j];
        }
        if(dp[i][j]!=-1)return dp[i][j];
        int down=INT_MAX,right=INT_MAX;
        if(j+1<grid[0].size() and visited[i][j+1]==-1){
            visited[i][j+1]=0;
            right=rec(dp,visited,grid,i,j+1);
            visited[i][j+1]=-1;
            //cout<<"right "<<right<<endl;
        }
        if(i+1<grid.size() and visited[i+1][j]==-1){
            visited[i+1][j]=0;
            down=rec(dp,visited,grid,i+1,j);
            visited[i+1][j]=-1;
            //cout<<"down "<<down<<endl;
        }
        return dp[i][j]=grid[i][j]+min(right,down);

}
int minPathSum(vector<vector<int>>& grid) {
        vector<vector<int>> visited(grid.size(),vector<int> (grid[0].size(),-1));
        vector<vector<int>> dp(grid.size(),vector<int> (grid[0].size(),-1));
        return rec(dp,visited,grid,0,0);

}
```

TimeComplexity:O(2n)**

SpaceComplexity:O(m*n)



54. Spiral Matrix

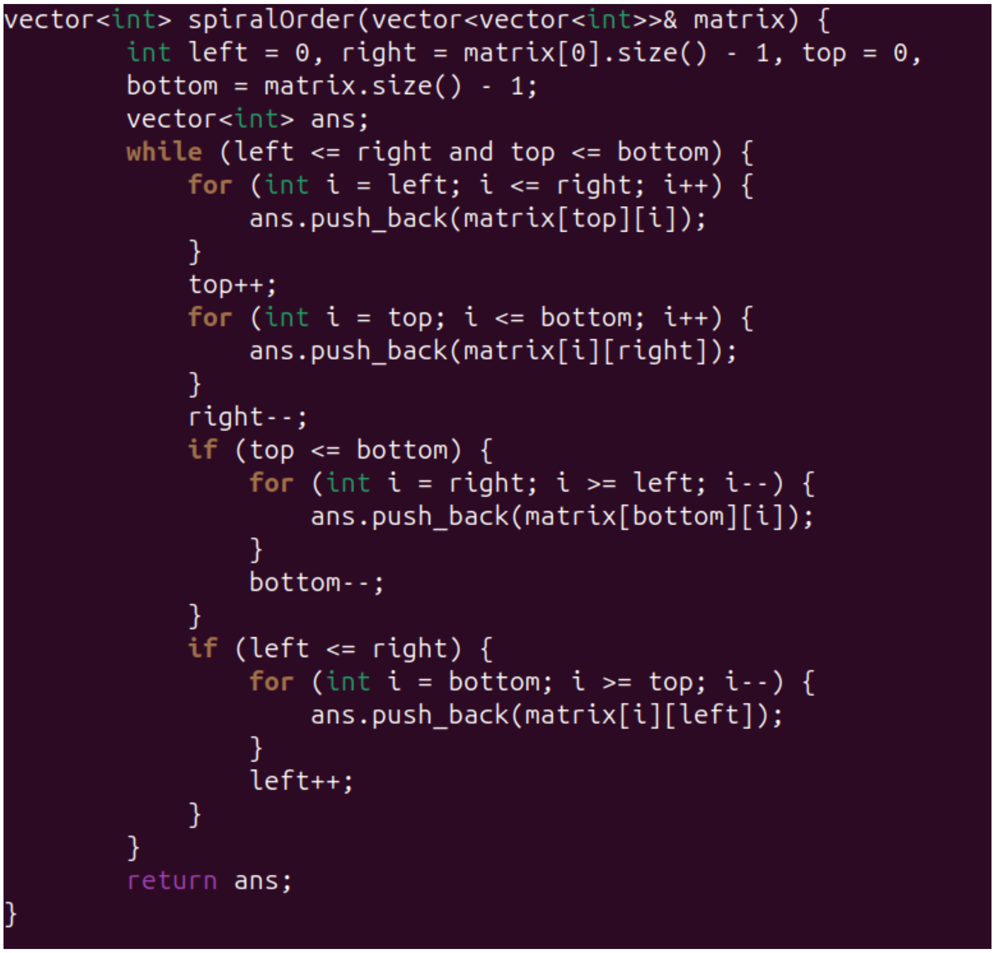
Solved

Given an `m x n` `matrix`, return *all elements of the* `matrix` *in spiral order*.

Example 1:

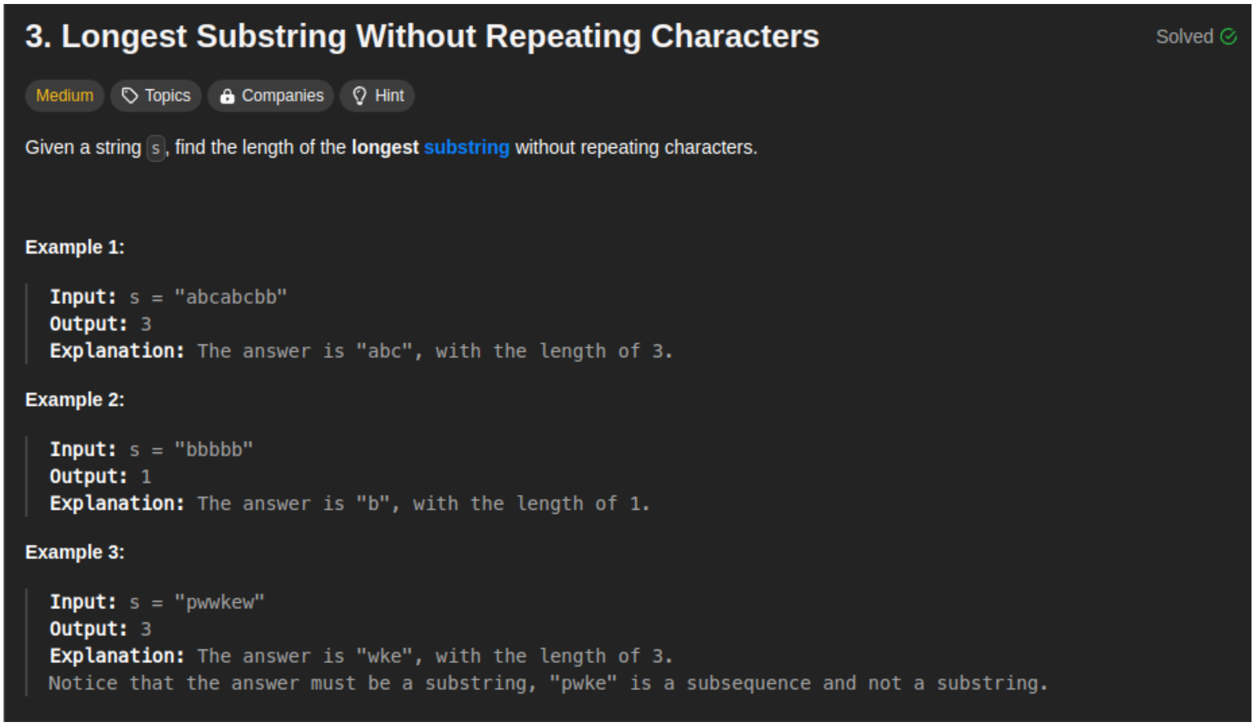
```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
```

Code:



```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int left = 0, right = matrix[0].size() - 1, top = 0,
        bottom = matrix.size() - 1;
        vector<int> ans;
        while (left <= right and top <= bottom) {
            for (int i = left; i <= right; i++) {
                ans.push_back(matrix[top][i]);
            }
            top++;
            for (int i = top; i <= bottom; i++) {
                ans.push_back(matrix[i][right]);
            }
            right--;
            if (top <= bottom) {
                for (int i = right; i >= left; i--) {
                    ans.push_back(matrix[bottom][i]);
                }
                bottom--;
            }
            if (left <= right) {
                for (int i = bottom; i >= top; i--) {
                    ans.push_back(matrix[i][left]);
                }
                left++;
            }
        }
        return ans;
}
```

TimeComplexity:O(m*n)
SpaceComplexity:O(m*n)



3. Longest Substring Without Repeating Characters

Solved

Given a string `s`, find the length of the **longest** substring without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

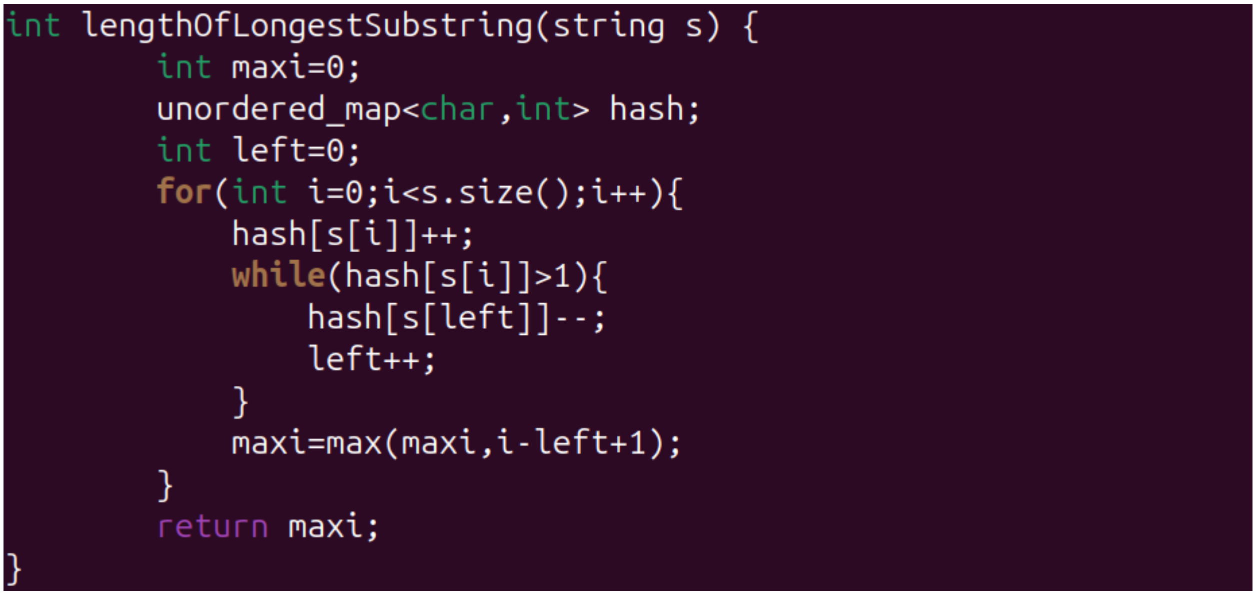
Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

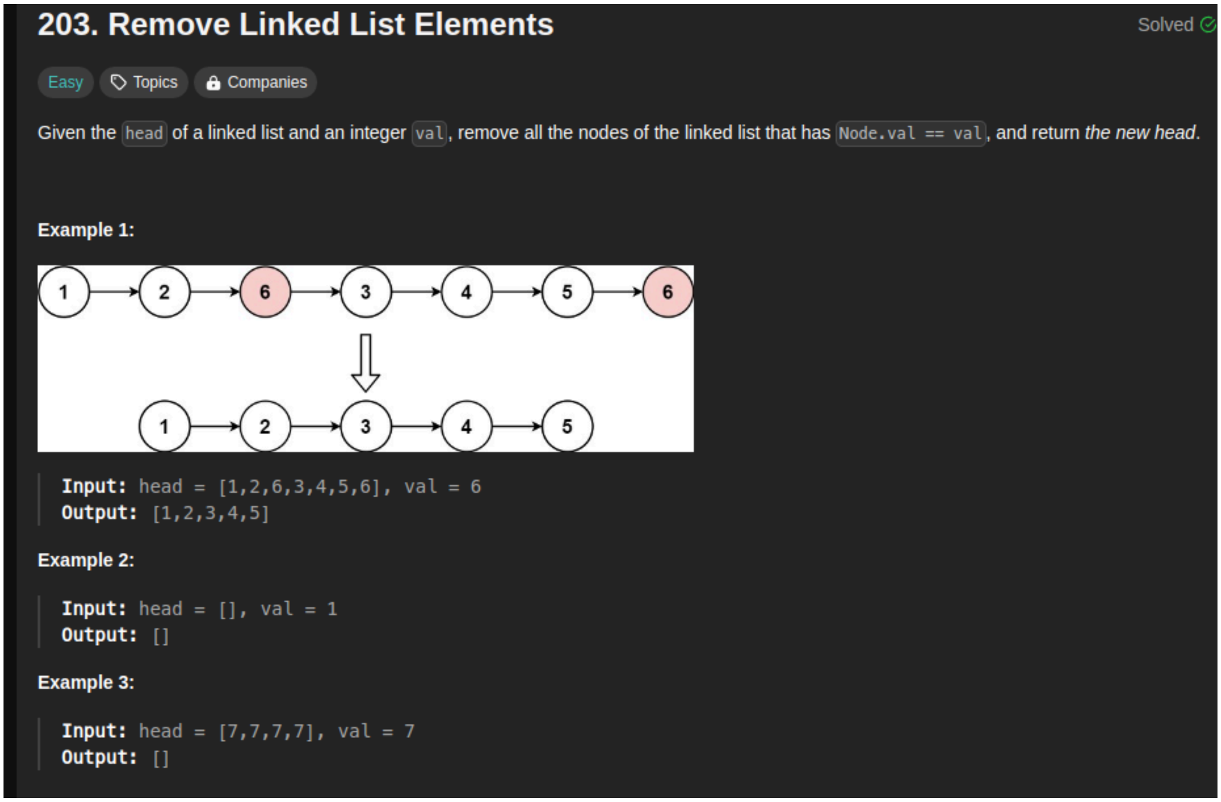
```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

Code:



```
int lengthOfLongestSubstring(string s) {
        int maxi=0;
        unordered_map<char,int> hash;
        int left=0;
        for(int i=0;i<s.size();i++){
            hash[s[i]]++;
            while(hash[s[i]]>1){
                hash[s[left]]--;
                left++;
            }
            maxi=max(maxi,i-left+1);
        }
        return maxi;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)



203. Remove Linked List Elements

Solved ✓

Easy | Topics | 🔒 Companies

Given the `head` of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

Example 1:

```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

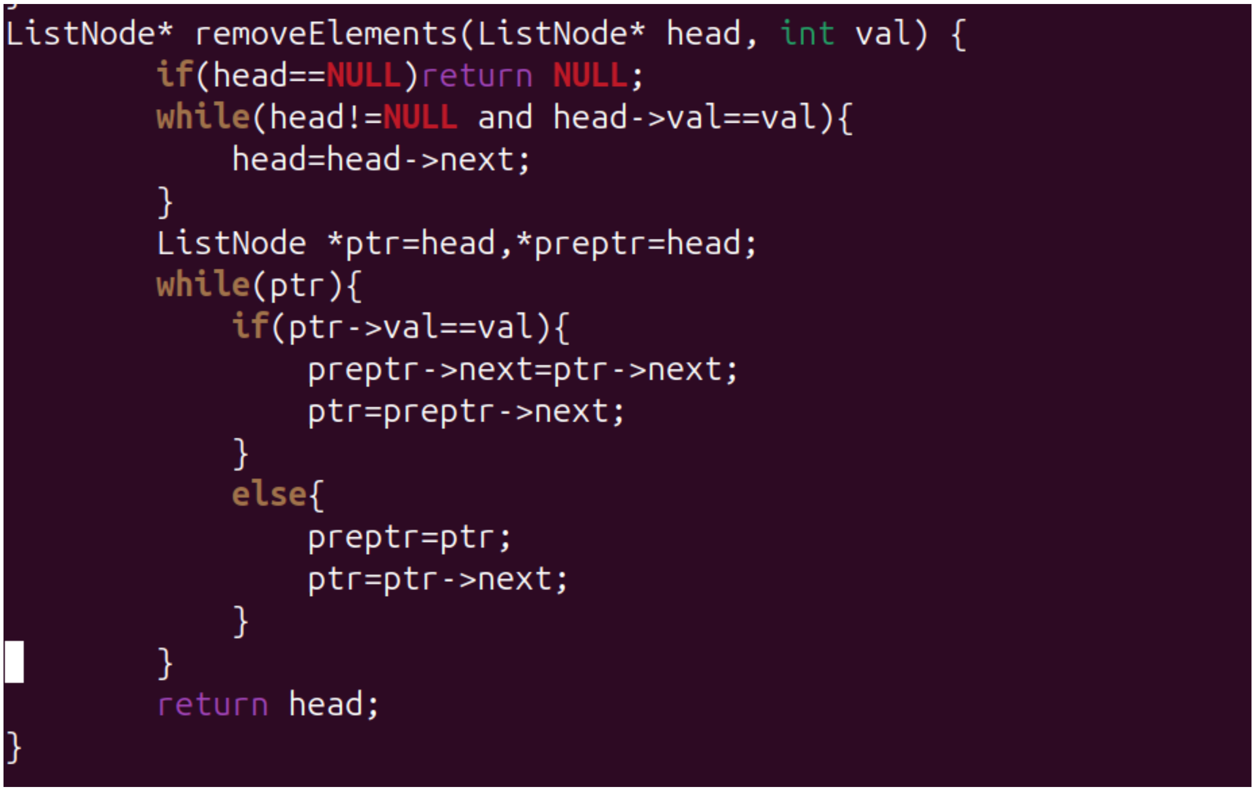
Example 2:

```
Input: head = [], val = 1
Output: []
```

Example 3:

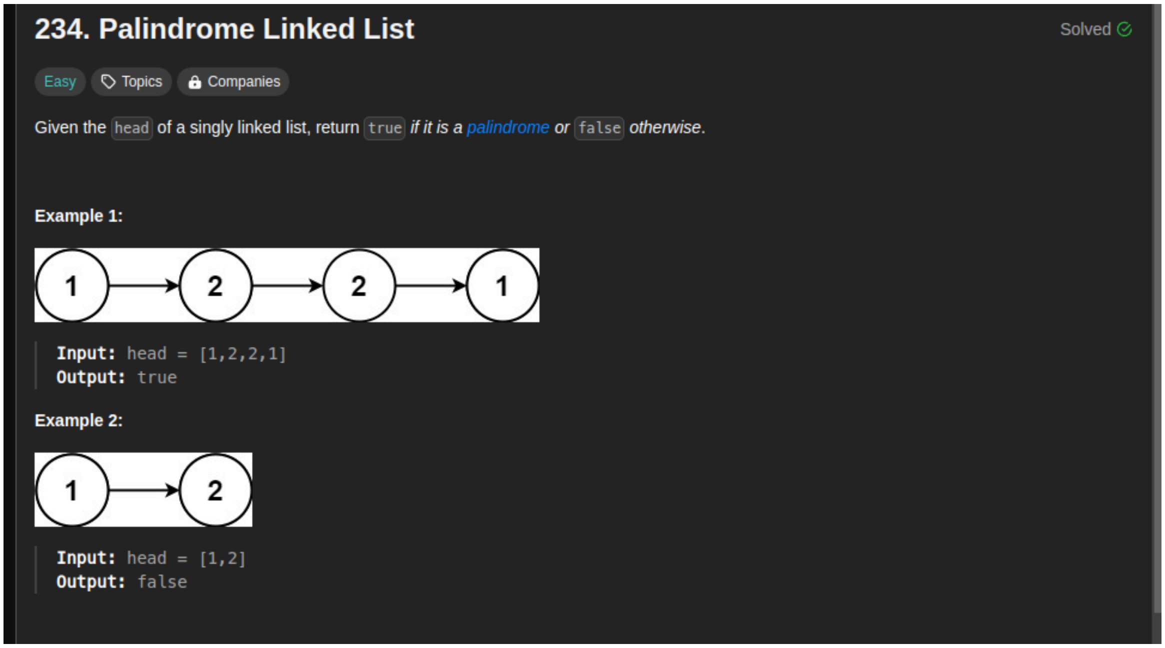
```
Input: head = [7,7,7,7], val = 7
Output: []
```

Code:



```
ListNode* removeElements(ListNode* head, int val) {
        if(head==NULL)return NULL;
        while(head!=NULL and head->val==val){
            head=head->next;
        }
        ListNode *ptr=head,*preptr=head;
        while(ptr){
            if(ptr->val==val){
                preptr->next=ptr->next;
                ptr=preptr->next;
            }
            else{
                preptr=ptr;
                ptr=ptr->next;
            }
        }
        return head;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(1)



234. Palindrome Linked List

Solved ✓

Easy | Topics | Companies

Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

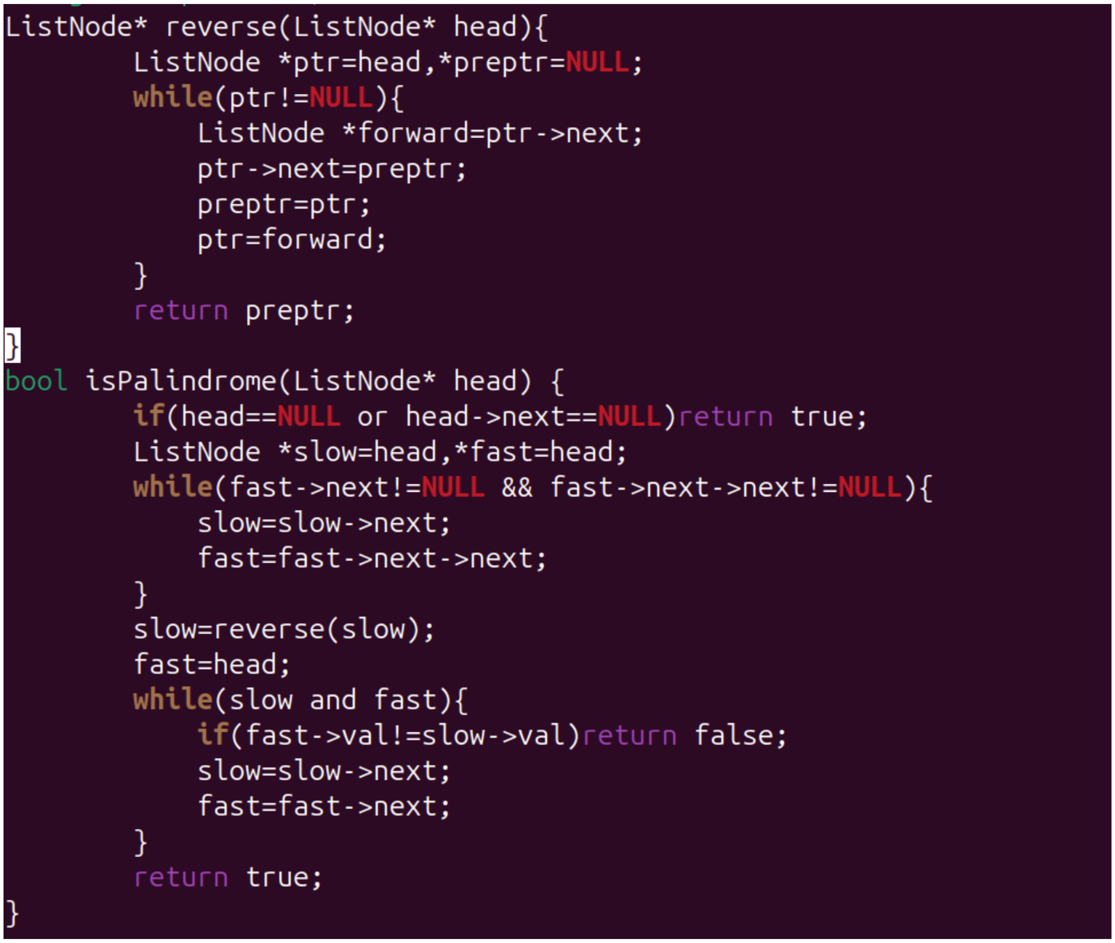
Example 1:

```
Input: head = [1,2,2,1]
Output: true
```

Example 2:

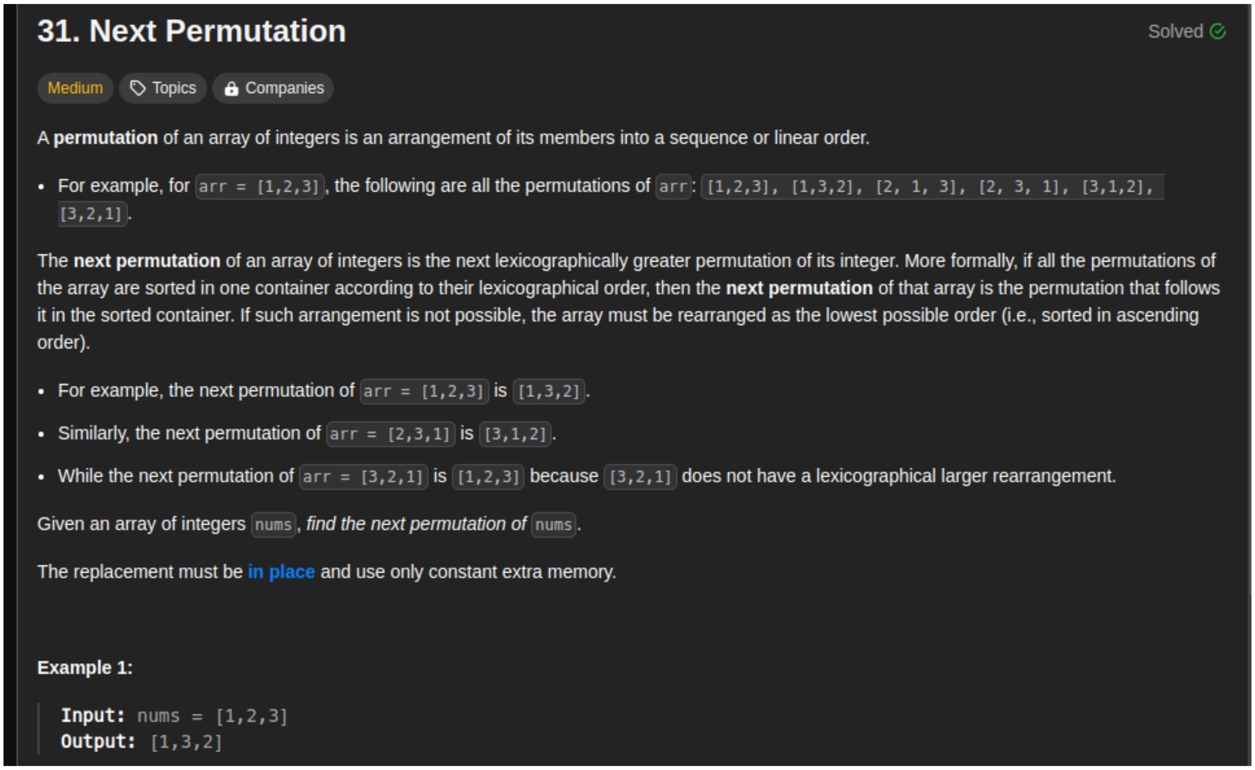
```
Input: head = [1,2]
Output: false
```

Code:



```
ListNode* reverse(ListNode* head){
        ListNode *ptr=head,*preptr=NULL;
        while(ptr!=NULL){
            ListNode *forward=ptr->next;
            ptr->next=preptr;
            preptr=ptr;
            ptr=forward;
        }
        return preptr;
}
bool isPalindrome(ListNode* head) {
        if(head==NULL or head->next==NULL)return true;
        ListNode *slow=head,*fast=head;
        while(fast->next!=NULL && fast->next->next!=NULL){
            slow=slow->next;
            fast=fast->next->next;
        }
        slow=reverse(slow);
        fast=head;
        while(slow and fast){
            if(fast->val!=slow->val)return false;
            slow=slow->next;
            fast=fast->next;
        }
        return true;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(1)



31. Next Permutation

Solved ✓

Medium | Topics | 🔒 Companies

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for `arr = [1,2,3]`, the following are all the permutations of `arr`: `[1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1]`.

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of `arr = [1,2,3]` is `[1,3,2]`.
- Similarly, the next permutation of `arr = [2,3,1]` is `[3,1,2]`.
- While the next permutation of `arr = [3,2,1]` is `[1,2,3]` because `[3,2,1]` does not have a lexicographical larger rearrangement.

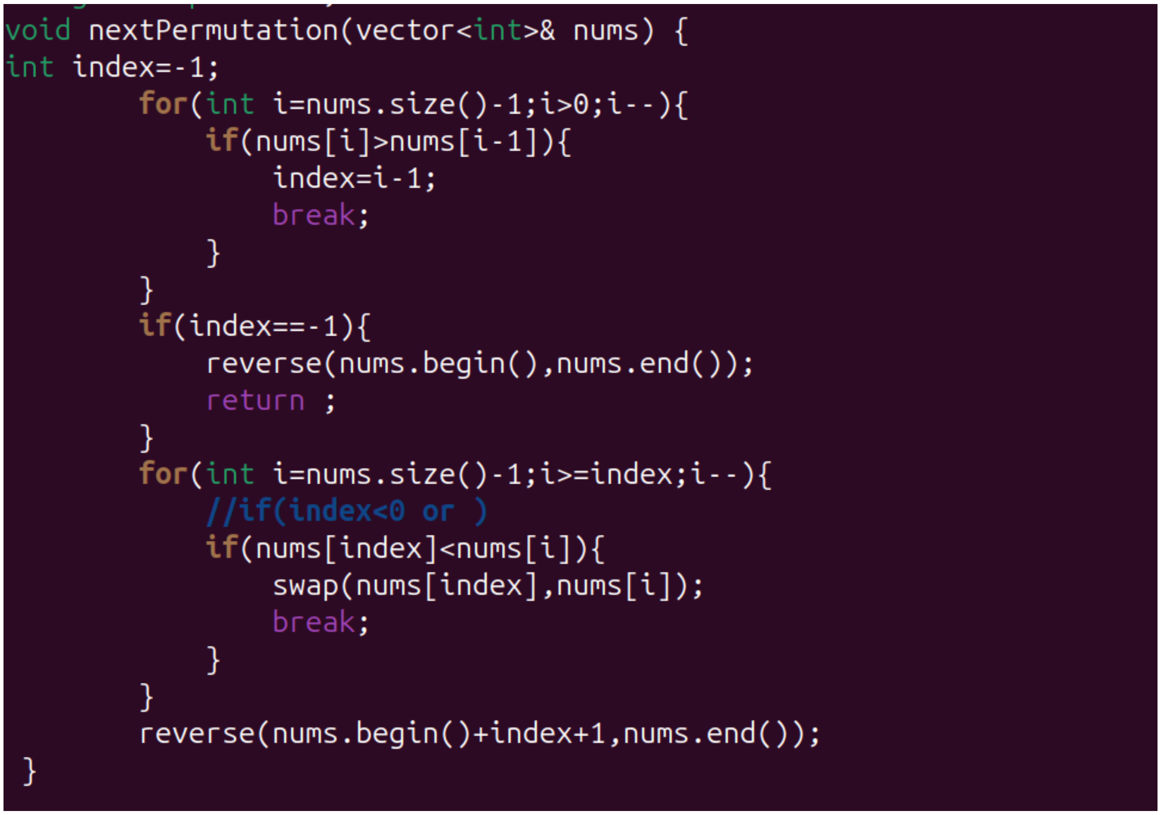
Given an array of integers `nums`, *find the next permutation of* `nums`.

The replacement must be in place and use only constant extra memory.

Example 1:

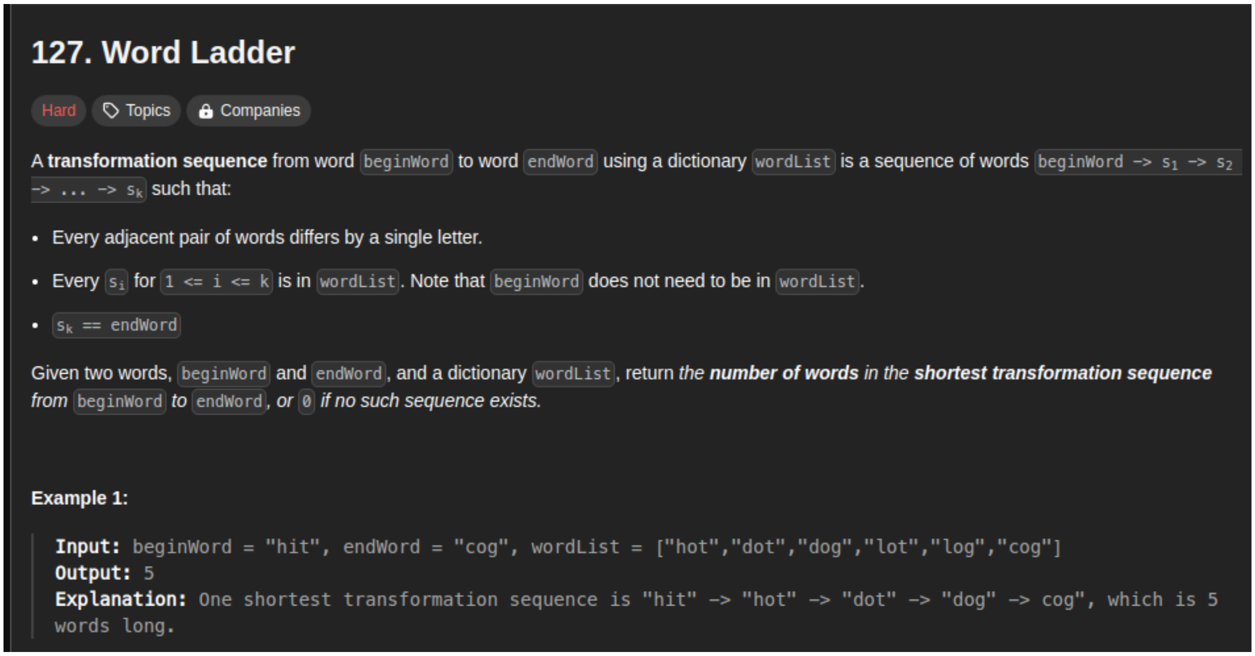
> **Input:** nums = [1,2,3]
> **Output:** [1,3,2]

Code:



```
void nextPermutation(vector<int>& nums) {
int index=-1;
        for(int i=nums.size()-1;i>0;i--){
            if(nums[i]>nums[i-1]){
                index=i-1;
                break;
            }
        }
        if(index==-1){
            reverse(nums.begin(),nums.end());
            return ;
        }
        for(int i=nums.size()-1;i>=index;i--){
            //if(index<0 or )
            if(nums[index]<nums[i]){
                swap(nums[index],nums[i]);
                break;
            }
        }
        reverse(nums.begin()+index+1,nums.end());
 }
```

TimeComplexity:O(n)
SpaceComplexity:O(1)



127. Word Ladder

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord` -> s_1 -> s_2 -> ... -> s_k such that:

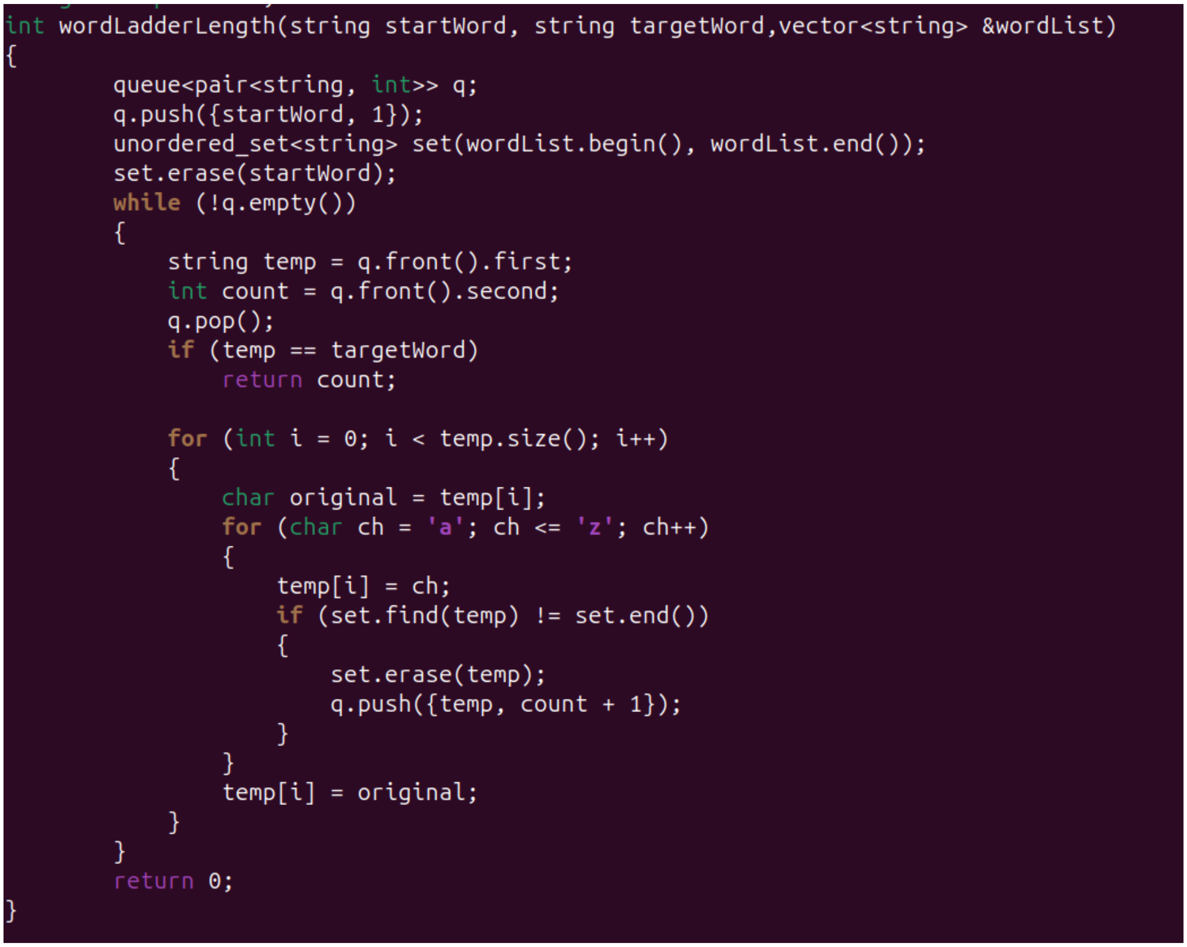
- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5
words long.
```

Code:



```
int wordLadderLength(string startWord, string targetWord,vector<string> &wordList)
{
        queue<pair<string, int>> q;
        q.push({startWord, 1});
        unordered_set<string> set(wordList.begin(), wordList.end());
        set.erase(startWord);
        while (!q.empty())
        {
            string temp = q.front().first;
            int count = q.front().second;
            q.pop();
            if (temp == targetWord)
                return count;

            for (int i = 0; i < temp.size(); i++)
            {
                char original = temp[i];
                for (char ch = 'a'; ch <= 'z'; ch++)
                {
                    temp[i] = ch;
                    if (set.find(temp) != set.end())
                    {
                        set.erase(temp);
                        q.push({temp, count + 1});
                    }
                }
                temp[i] = original;
            }
        }
        return 0;
}
```

TimeComplexity:O(n2)**
SpaceComplexity:O(n2)**