Recursion

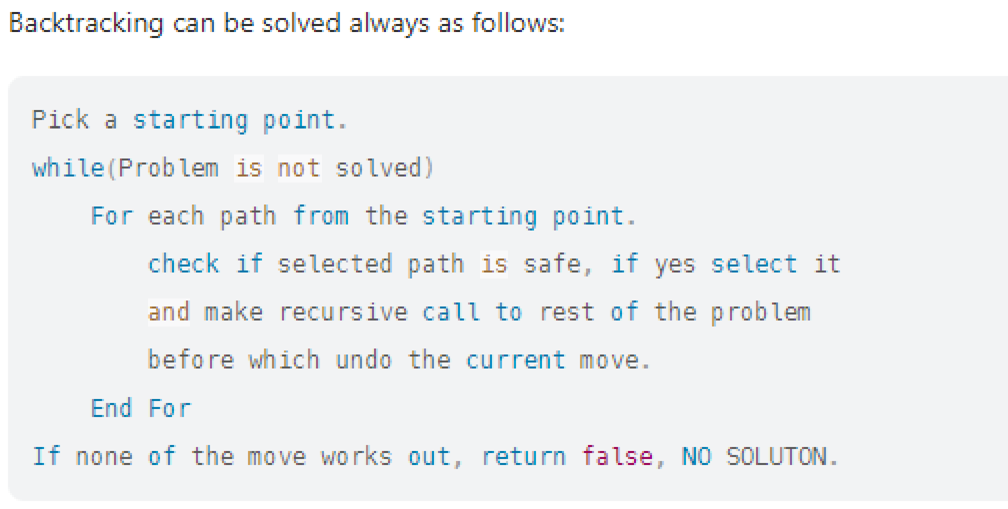
Backtracking

Trees

Graphs

DP

Recursion backtracking

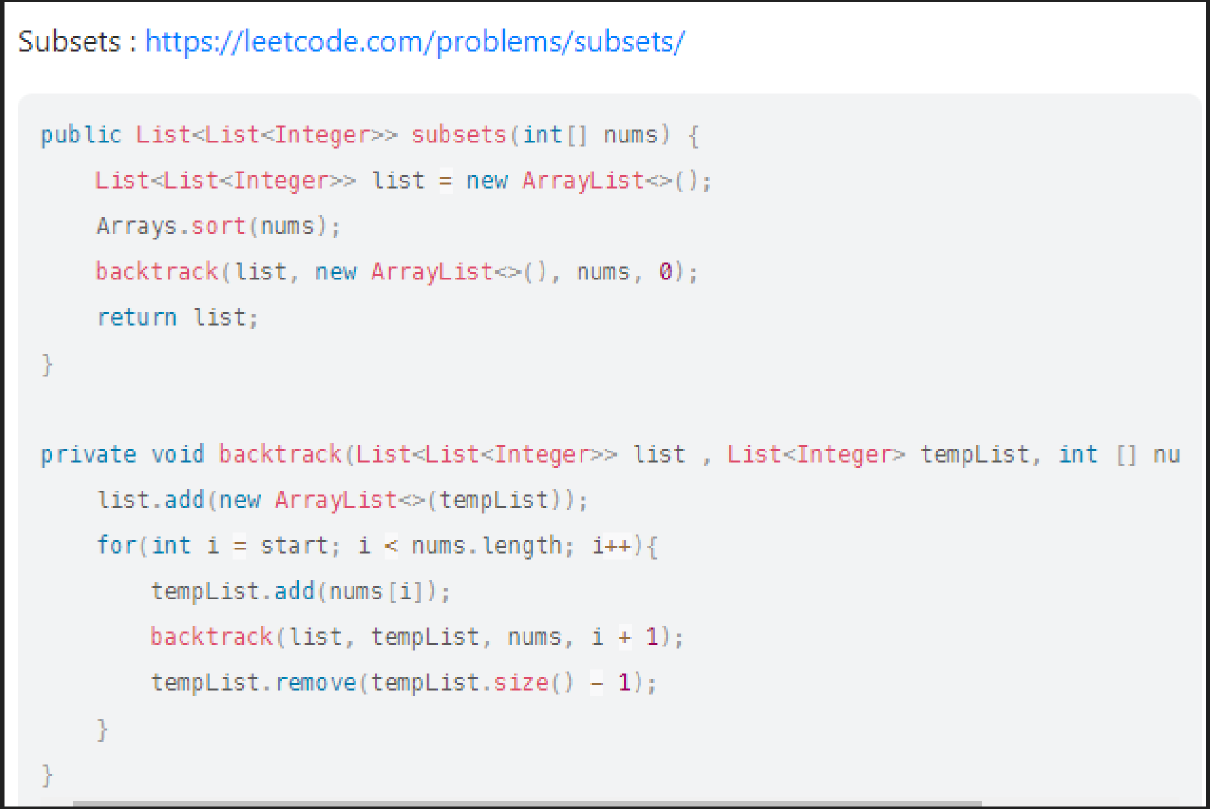


Backtracking can be solved always as follows:

```
Pick a starting point.
while(Problem is not solved)
    For each path from the starting point.
        check if selected path is safe, if yes select it
        and make recursive call to rest of the problem
        before which undo the current move.
    End For
If none of the move works out, return false, NO SOLUTON.
```

Subsets

https://leetcode.com/problems/subsets/

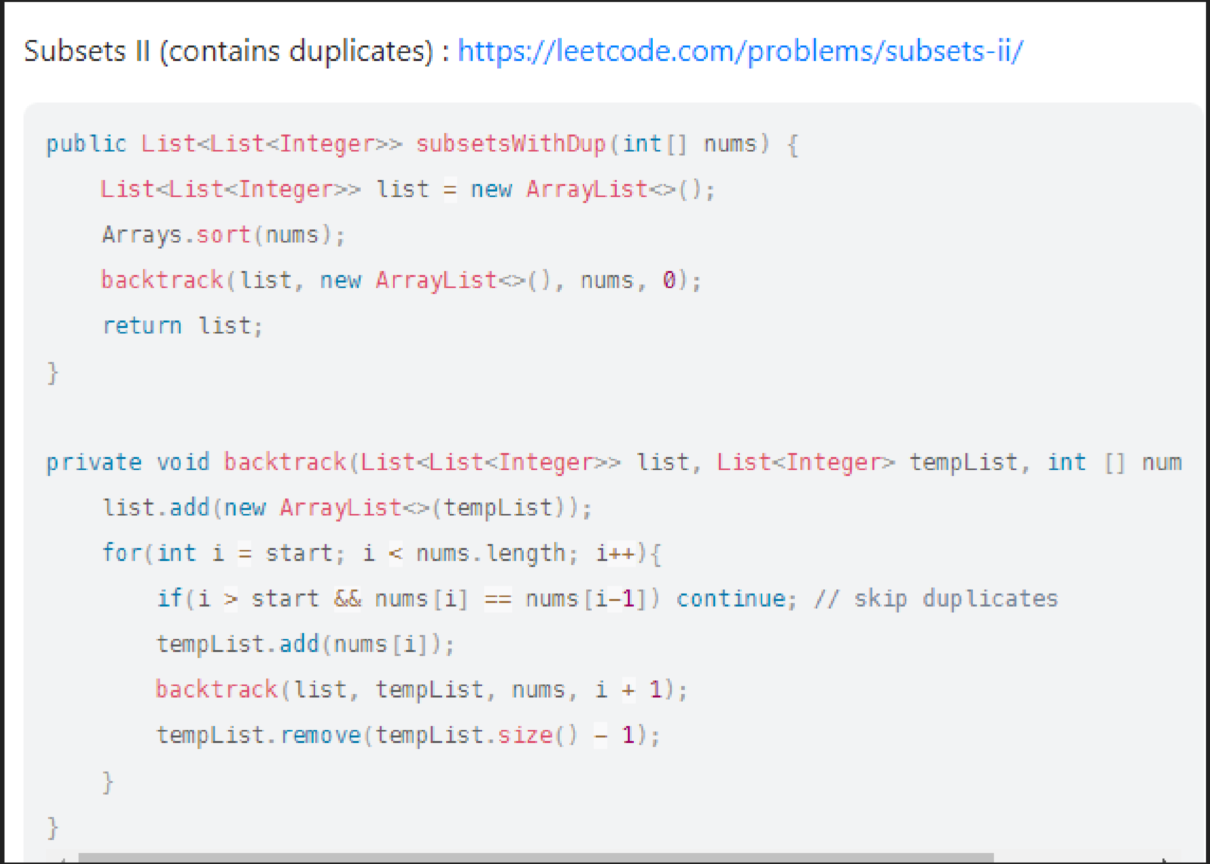


Subsets : https://leetcode.com/problems/subsets/

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;
}

private void backtrack(List<List<Integer>> list , List<Integer> tempList, int [] nu
    list.add(new ArrayList<>(tempList));
    for(int i = start; i < nums.length; i++){
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);
    }
}
```

Subsets 2

https://leetcode.com/problems/subsets-ii/

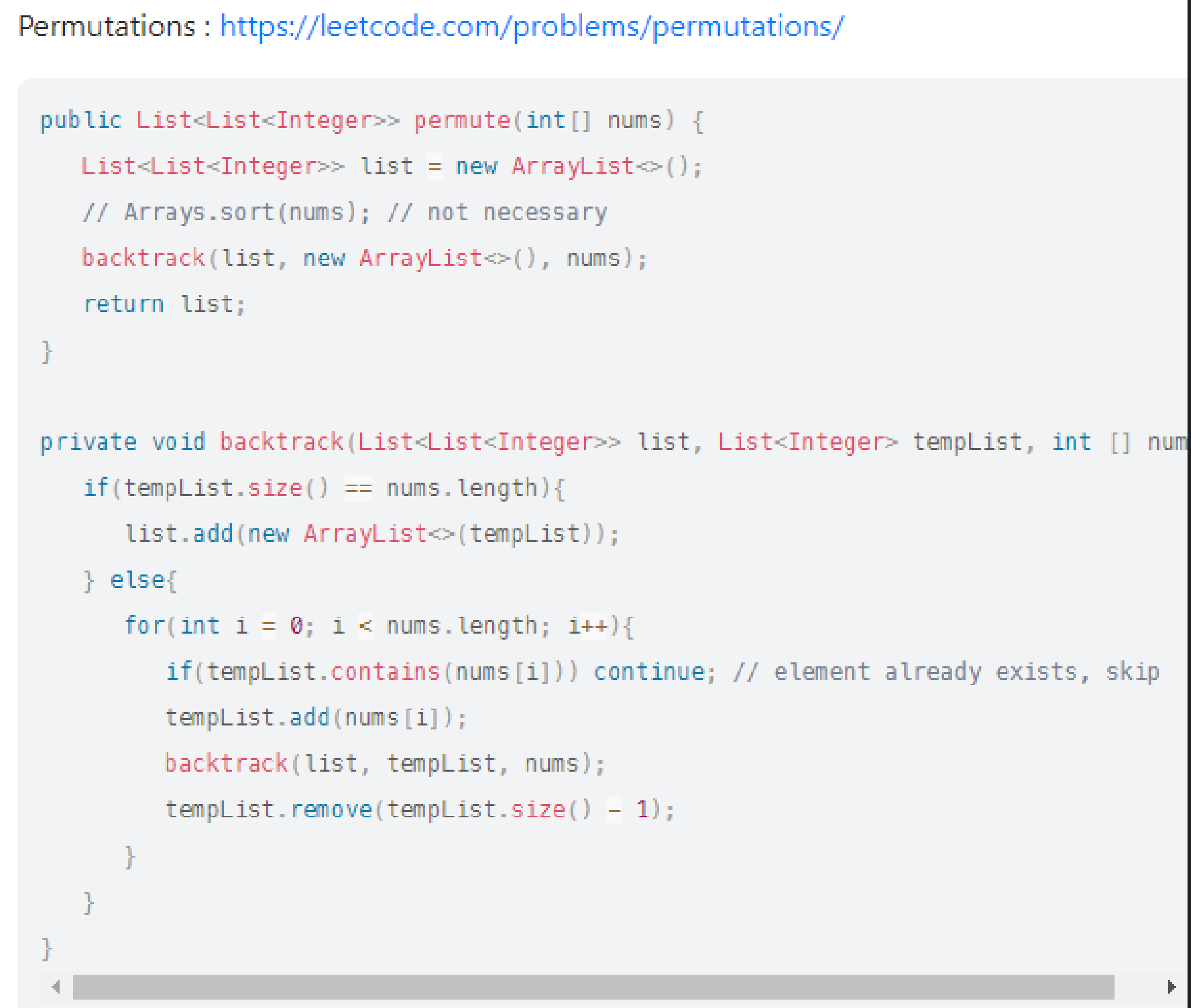


Subsets II (contains duplicates) : https://leetcode.com/problems/subsets-ii/

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
    list.add(new ArrayList<>(tempList));
    for(int i = start; i < nums.length; i++){
        if(i > start && nums[i] == nums[i-1]) continue; // skip duplicates
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);
    }
}
```

Permutations

https://leetcode.com/problems/permutations/

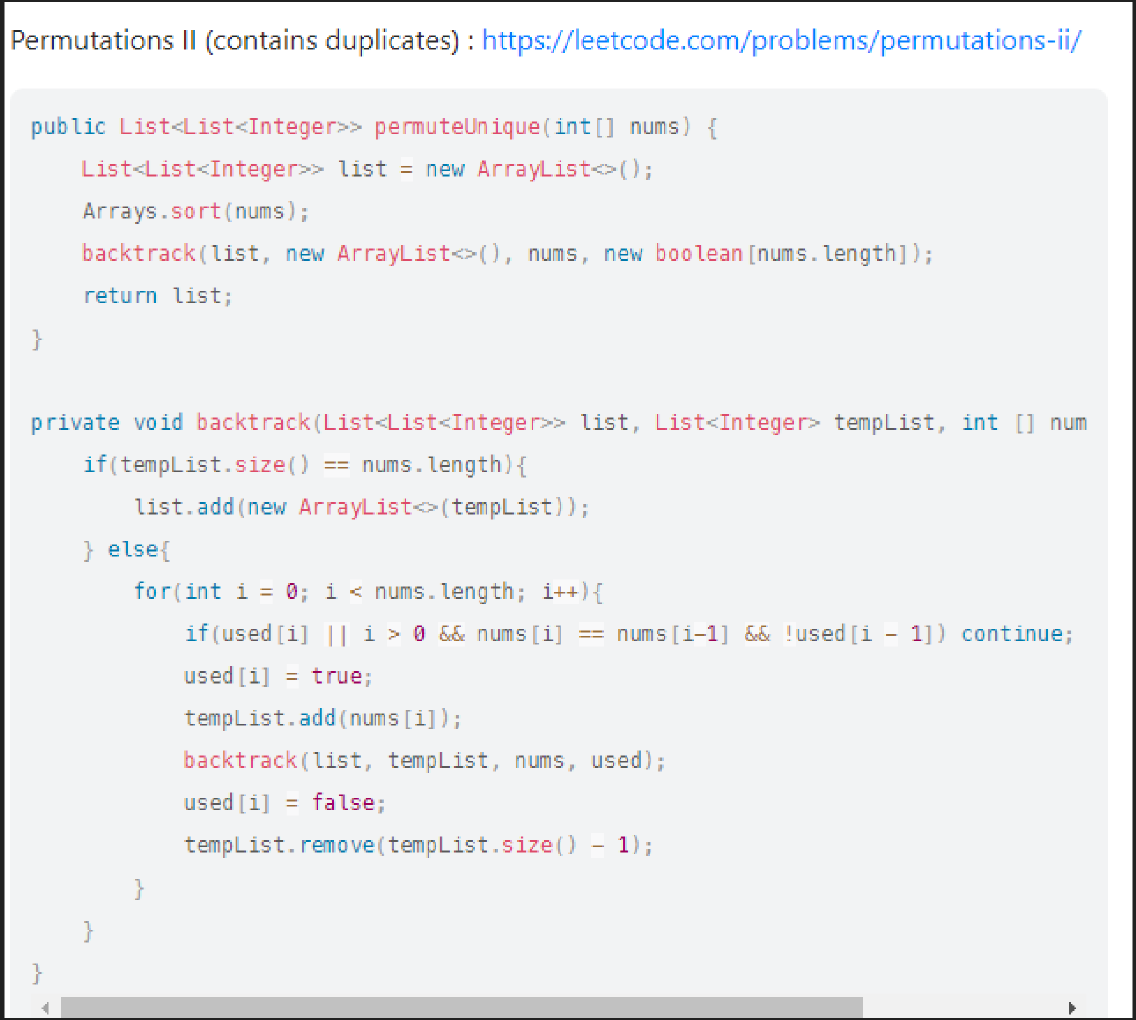


Permutations : https://leetcode.com/problems/permutations/

```
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> list = new ArrayList<>();
   // Arrays.sort(nums); // not necessary
   backtrack(list, new ArrayList<>(), nums);
   return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
   if(tempList.size() == nums.length){
      list.add(new ArrayList<>(tempList));
   } else{
      for(int i = 0; i < nums.length; i++){
         if(tempList.contains(nums[i])) continue; // element already exists, skip
         tempList.add(nums[i]);
         backtrack(list, tempList, nums);
         tempList.remove(tempList.size() - 1);
      }
   }
}
```

Permutations 2

https://leetcode.com/problems/permutations-ii/

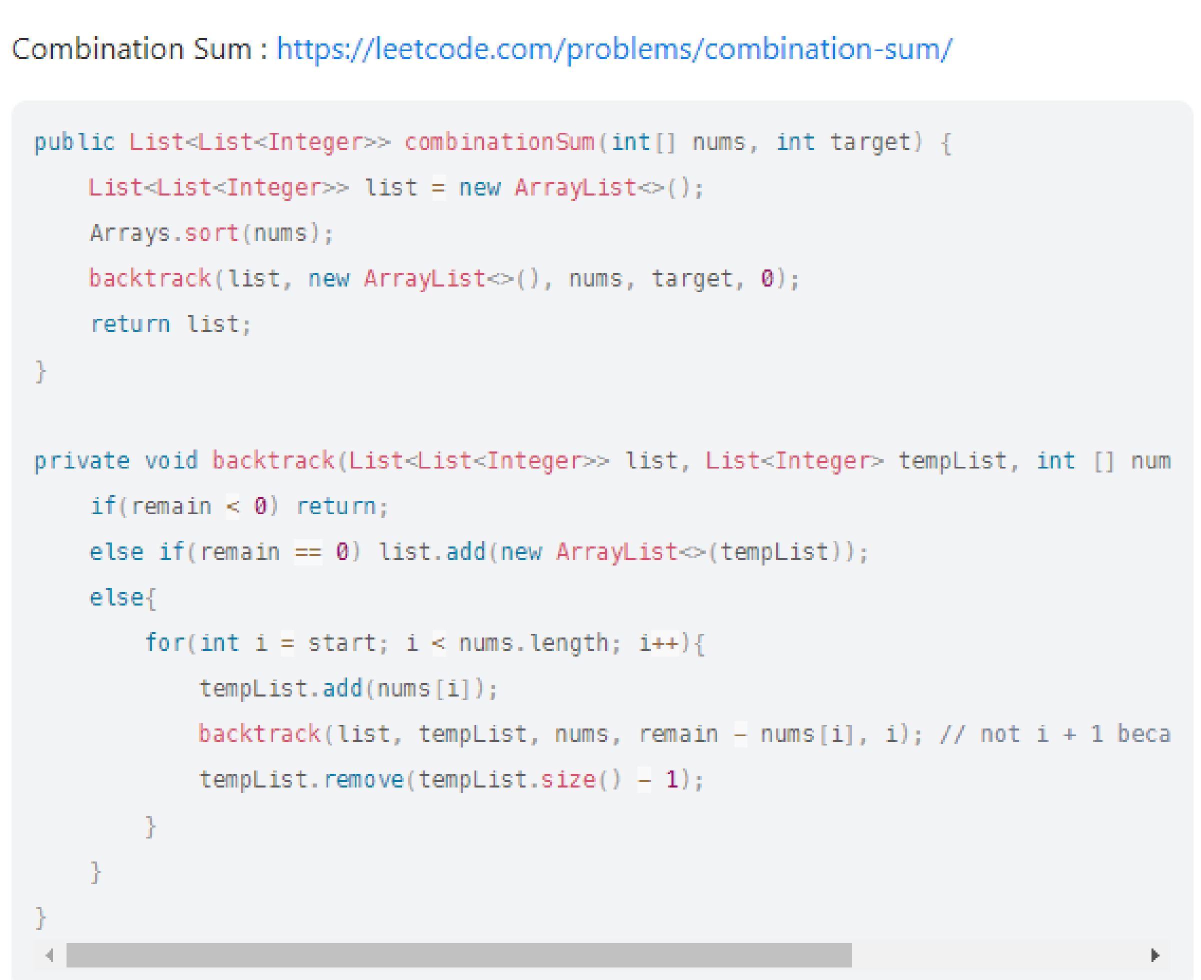


Permutations II (contains duplicates) : https://leetcode.com/problems/permutations-ii/

```java
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, new boolean[nums.length]);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
    if(tempList.size() == nums.length){
        list.add(new ArrayList<>(tempList));
    } else{
        for(int i = 0; i < nums.length; i++){
            if(used[i] || i > 0 && nums[i] == nums[i-1] && !used[i - 1]) continue;
            used[i] = true;
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, used);
            used[i] = false;
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Combination sum

https://leetcode.com/problems/combination-sum/

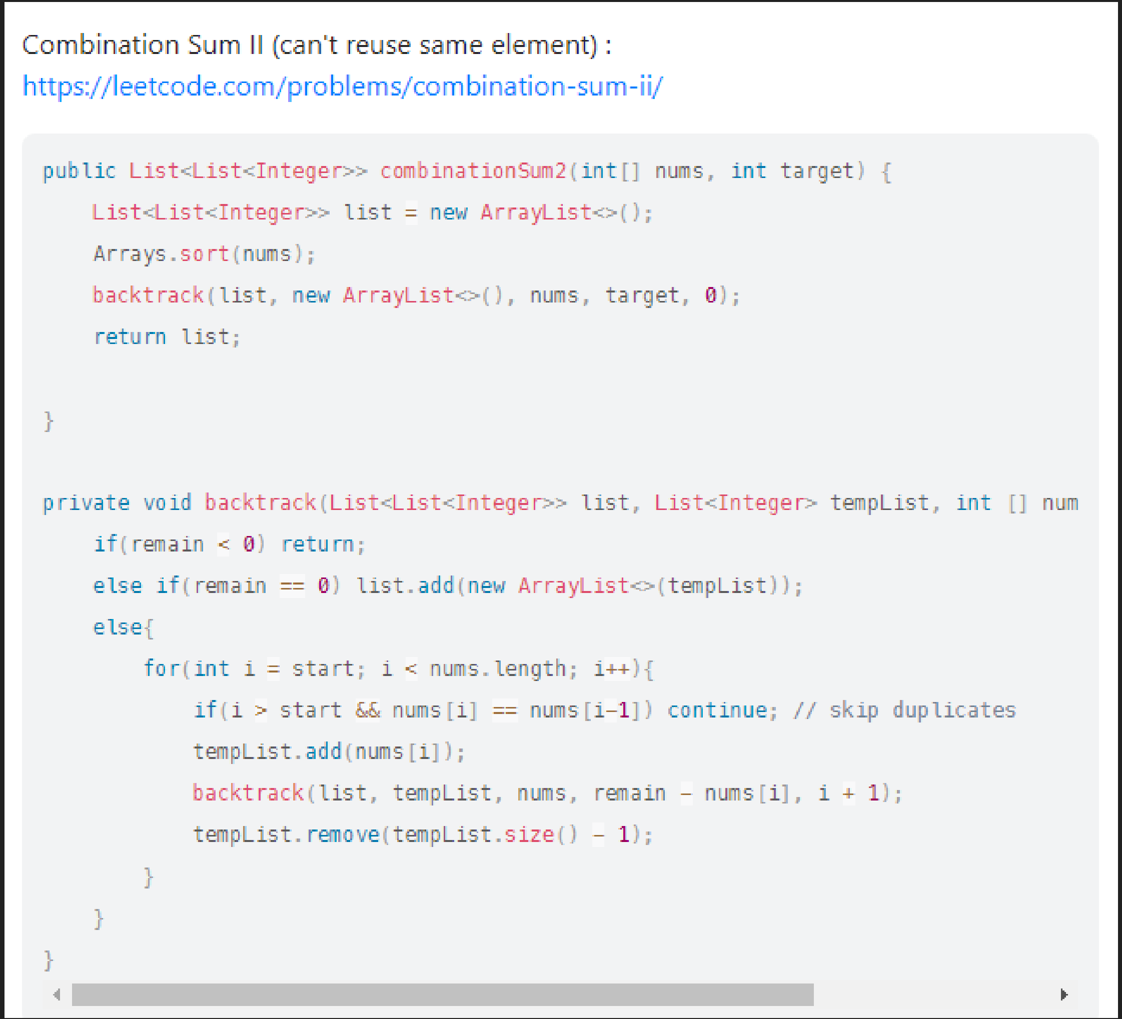


Combination Sum : https://leetcode.com/problems/combination-sum/

```
public List<List<Integer>> combinationSum(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i); // not i + 1 beca
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

Combination sum 2

https://leetcode.com/problems/combination-sum-ii/

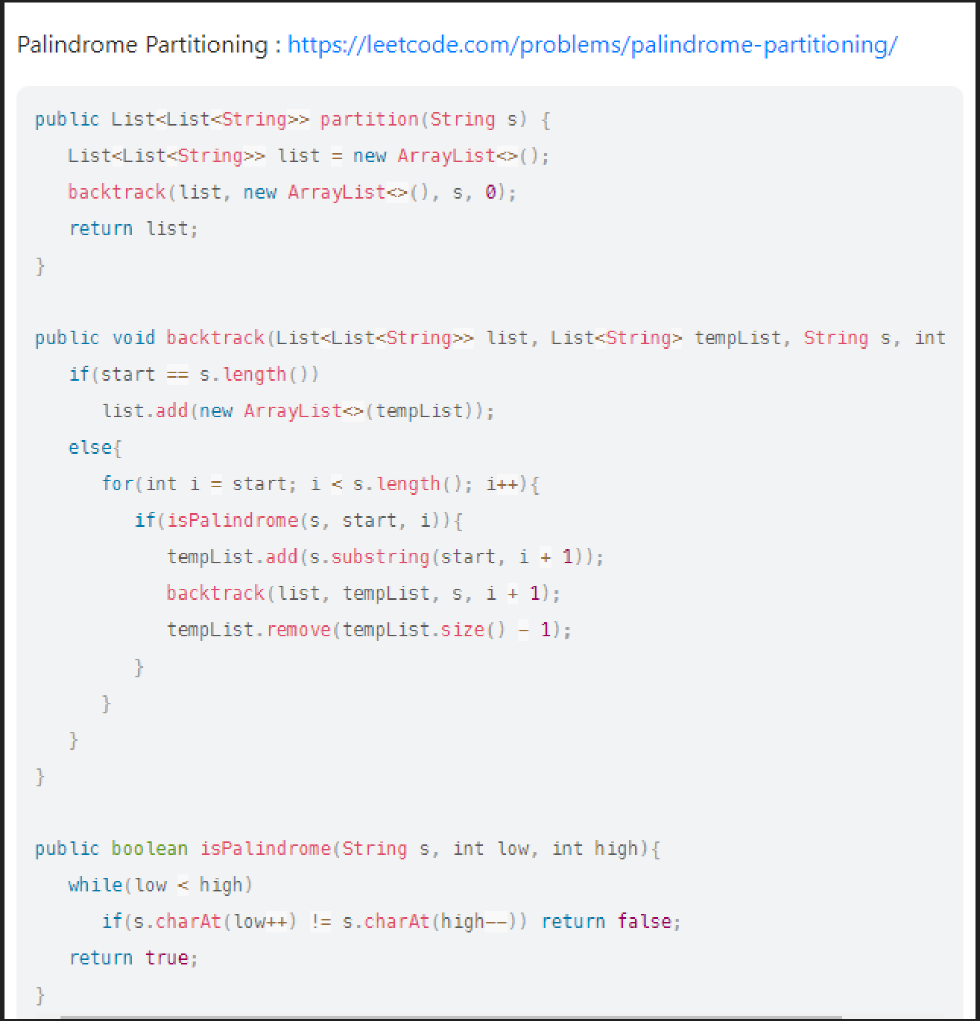


Combination Sum II (can't reuse same element) :
https://leetcode.com/problems/combination-sum-ii/

```
public List<List<Integer>> combinationSum2(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;

}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] num
    if(remain < 0) return;
    else if(remain == 0) list.add(new ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            if(i > start && nums[i] == nums[i-1]) continue; // skip duplicates
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

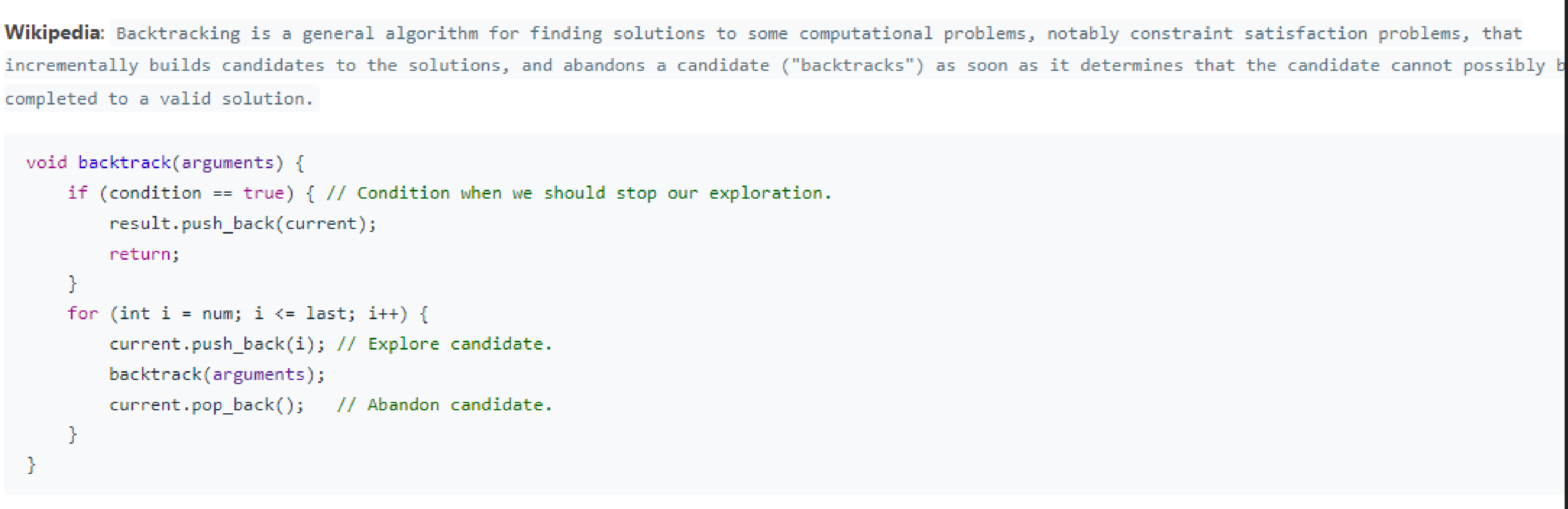
Palindrome partitioning

https://leetcode.com/problems/palindrome-partitioning/



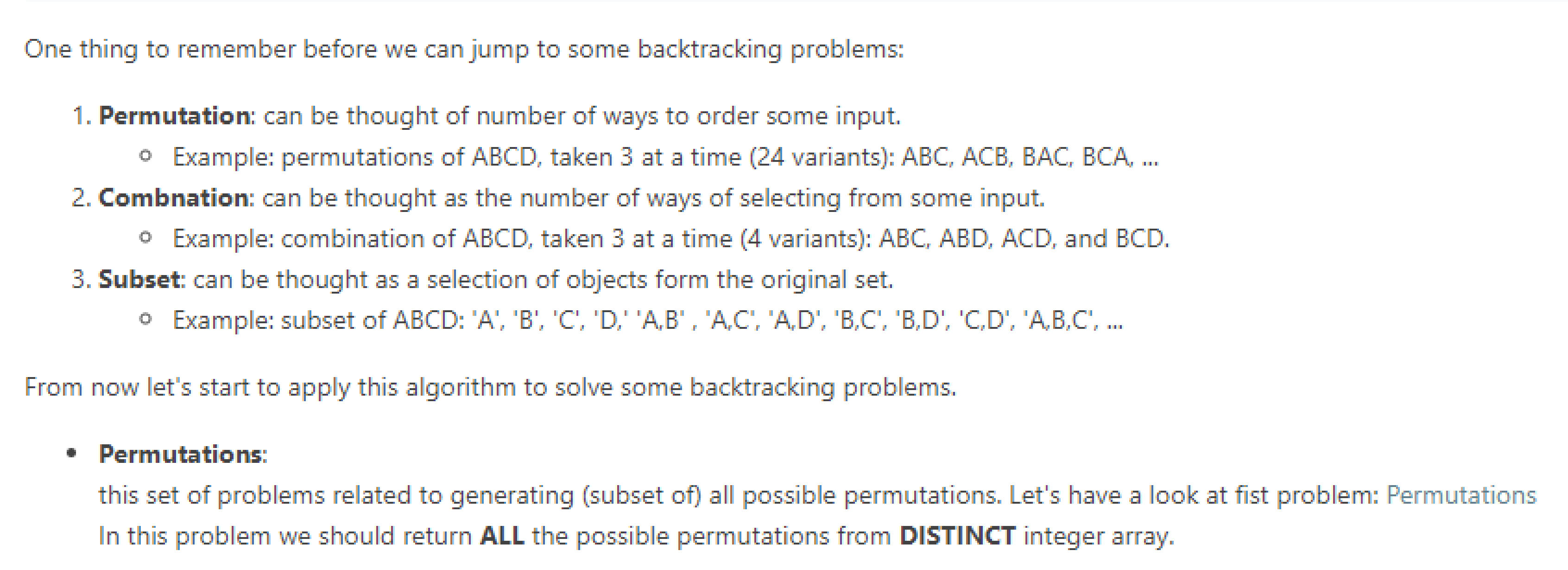
Palindrome Partitioning : https://leetcode.com/problems/palindrome-partitioning/

```java
public List<List<String>> partition(String s) {
   List<List<String>> list = new ArrayList<>();
   backtrack(list, new ArrayList<>(), s, 0);
   return list;
}

public void backtrack(List<List<String>> list, List<String> tempList, String s, int
   if(start == s.length())
      list.add(new ArrayList<>(tempList));
   else{
      for(int i = start; i < s.length(); i++){
         if(isPalindrome(s, start, i)){
            tempList.add(s.substring(start, i + 1));
            backtrack(list, tempList, s, i + 1);
            tempList.remove(tempList.size() - 1);
         }
      }
   }
}

public boolean isPalindrome(String s, int low, int high){
   while(low < high)
      if(s.charAt(low++) != s.charAt(high--)) return false;
   return true;
}
```



Wikipedia: `Backtracking is a general algorithm for finding solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly b completed to a valid solution.`

```
void backtrack(arguments) {
    if (condition == true) { // Condition when we should stop our exploration.
        result.push_back(current);
        return;
    }
    for (int i = num; i <= last; i++) {
        current.push_back(i); // Explore candidate.
        backtrack(arguments);
        current.pop_back();   // Abandon candidate.
    }
}
```

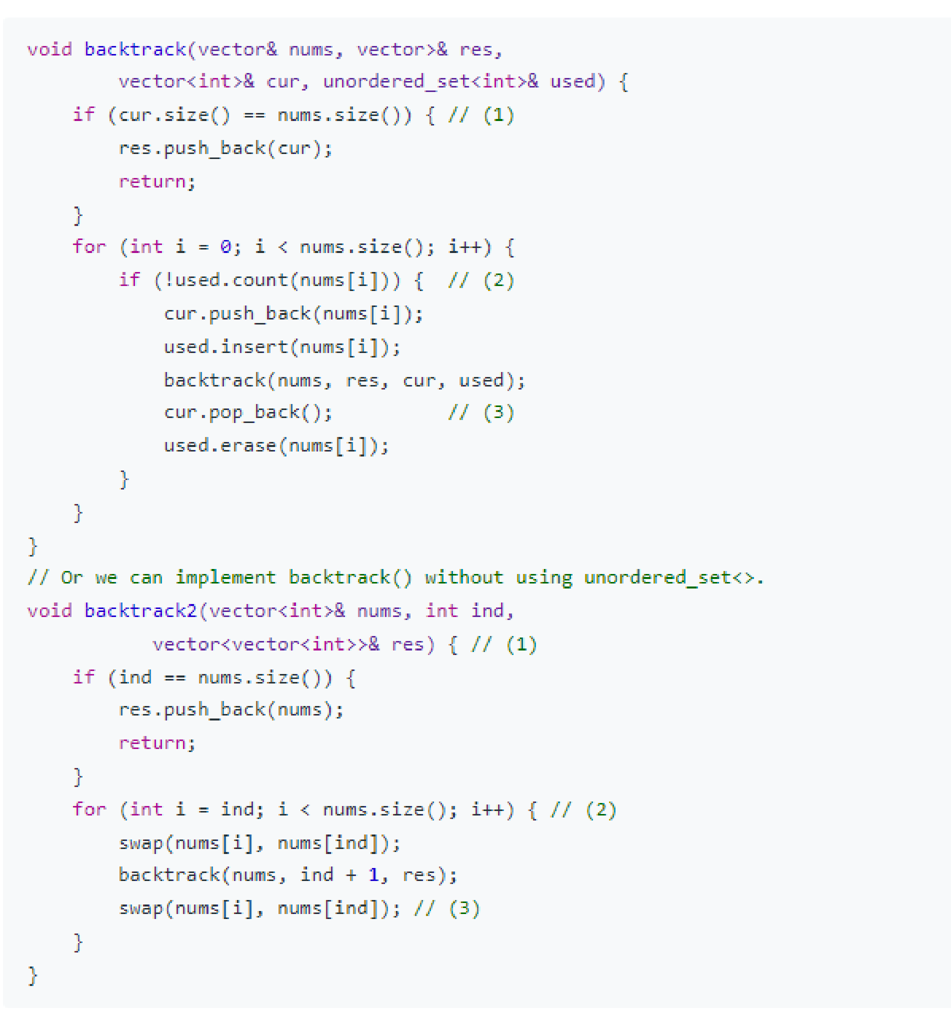


One thing to remember before we can jump to some backtracking problems:

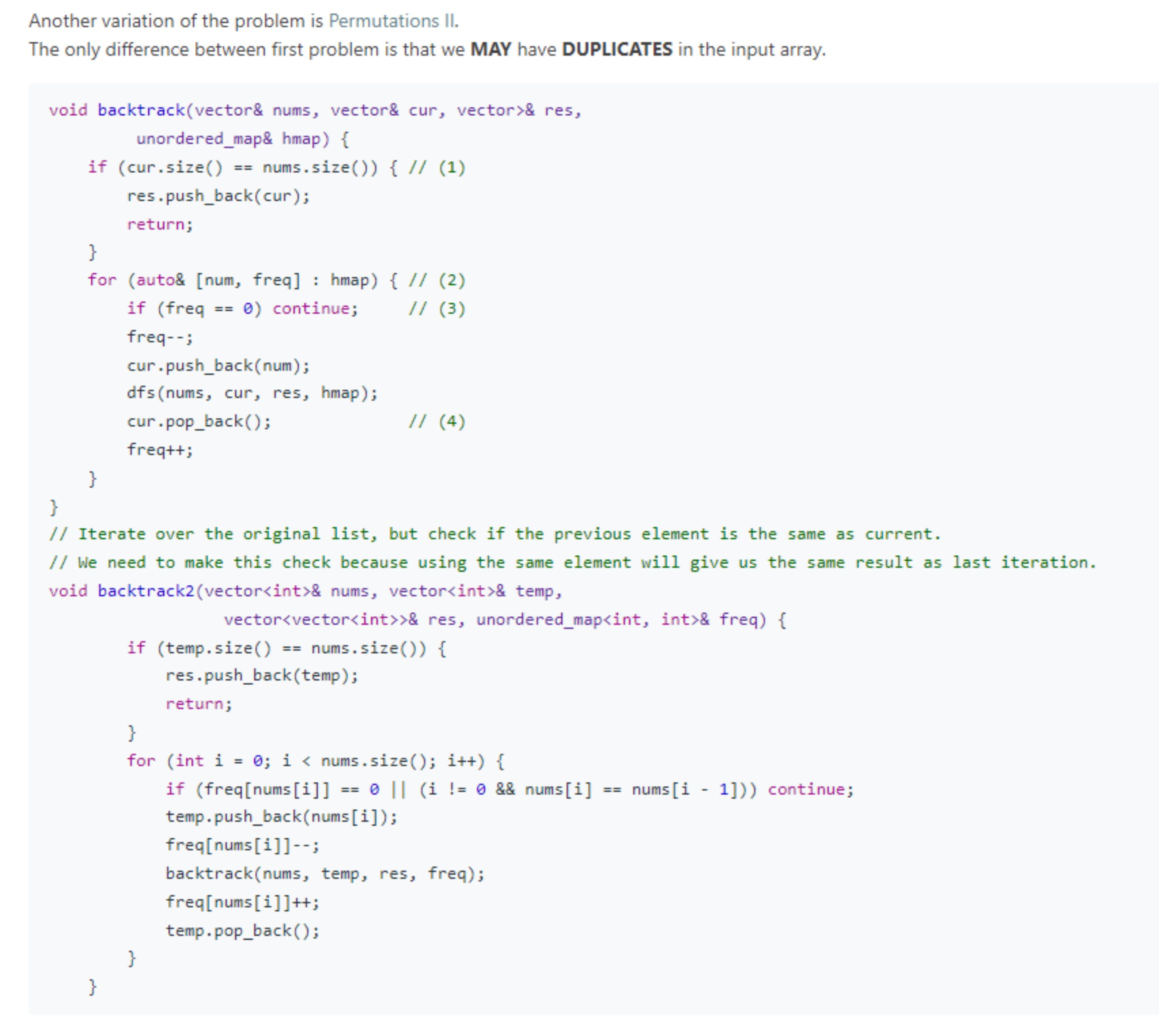
1. **Permutation**: can be thought of number of ways to order some input.
   - Example: permutations of ABCD, taken 3 at a time (24 variants): ABC, ACB, BAC, BCA, ...
2. **Combnation**: can be thought as the number of ways of selecting from some input.
   - Example: combination of ABCD, taken 3 at a time (4 variants): ABC, ABD, ACD, and BCD.
3. **Subset**: can be thought as a selection of objects form the original set.
   - Example: subset of ABCD: 'A', 'B', 'C', 'D,' 'A,B' , 'A,C', 'A,D', 'B,C', 'B,D', 'C,D', 'A,B,C', ...

From now let's start to apply this algorithm to solve some backtracking problems.

- **Permutations**:
  this set of problems related to generating (subset of) all possible permutations. Let's have a look at fist problem: Permutations
  In this problem we should return **ALL** the possible permutations from **DISTINCT** integer array.

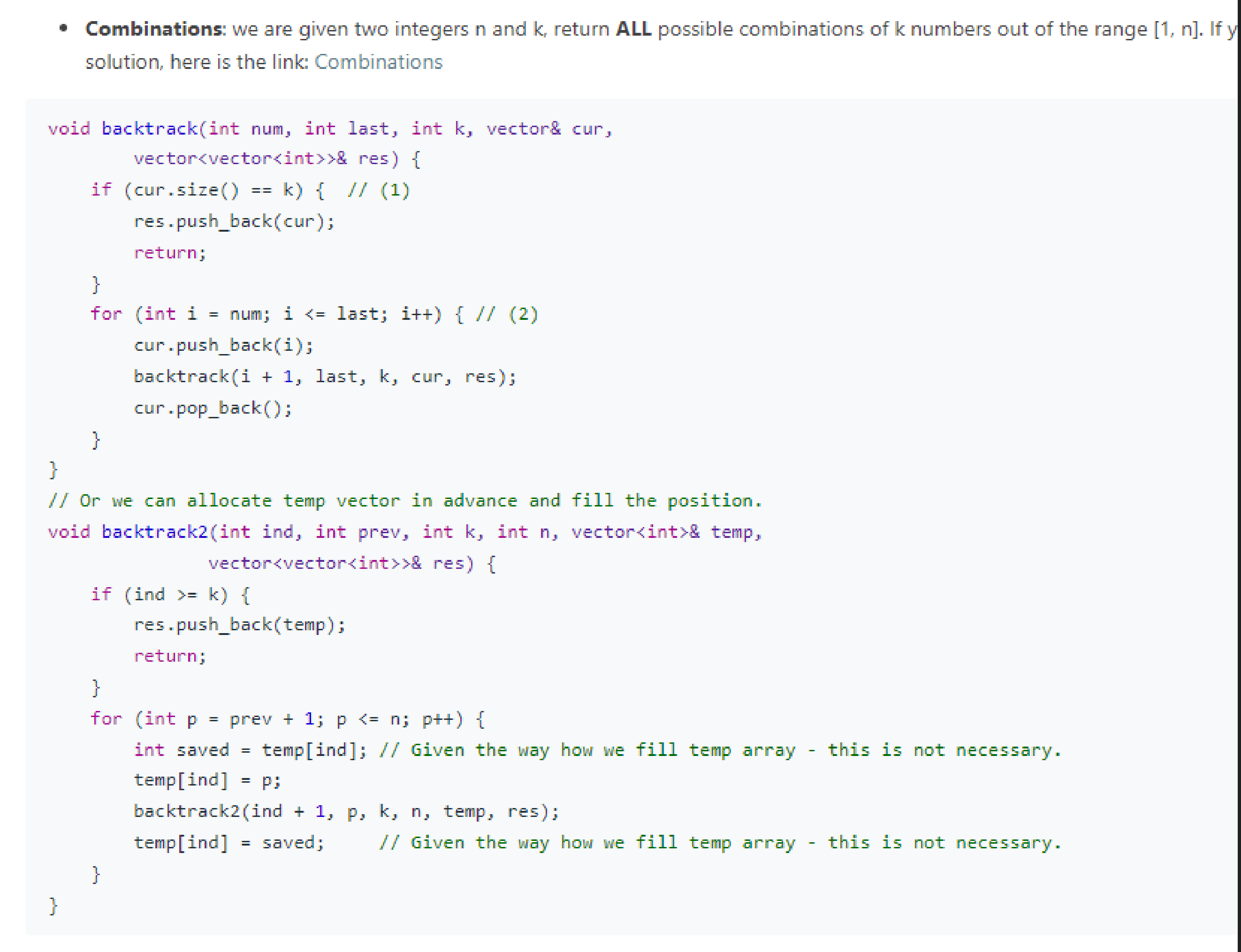


```
void backtrack(vector& nums, vector>& res,
        vector<int>& cur, unordered_set<int>& used) {
    if (cur.size() == nums.size()) { // (1)
        res.push_back(cur);
        return;
    }
    for (int i = 0; i < nums.size(); i++) {
        if (!used.count(nums[i])) {  // (2)
            cur.push_back(nums[i]);
            used.insert(nums[i]);
            backtrack(nums, res, cur, used);
            cur.pop_back();          // (3)
            used.erase(nums[i]);
        }
    }
}
// Or we can implement backtrack() without using unordered_set<>.
void backtrack2(vector<int>& nums, int ind,
           vector<vector<int>>& res) { // (1)
    if (ind == nums.size()) {
        res.push_back(nums);
        return;
    }
    for (int i = ind; i < nums.size(); i++) { // (2)
        swap(nums[i], nums[ind]);
        backtrack(nums, ind + 1, res);
        swap(nums[i], nums[ind]); // (3)
    }
}
```



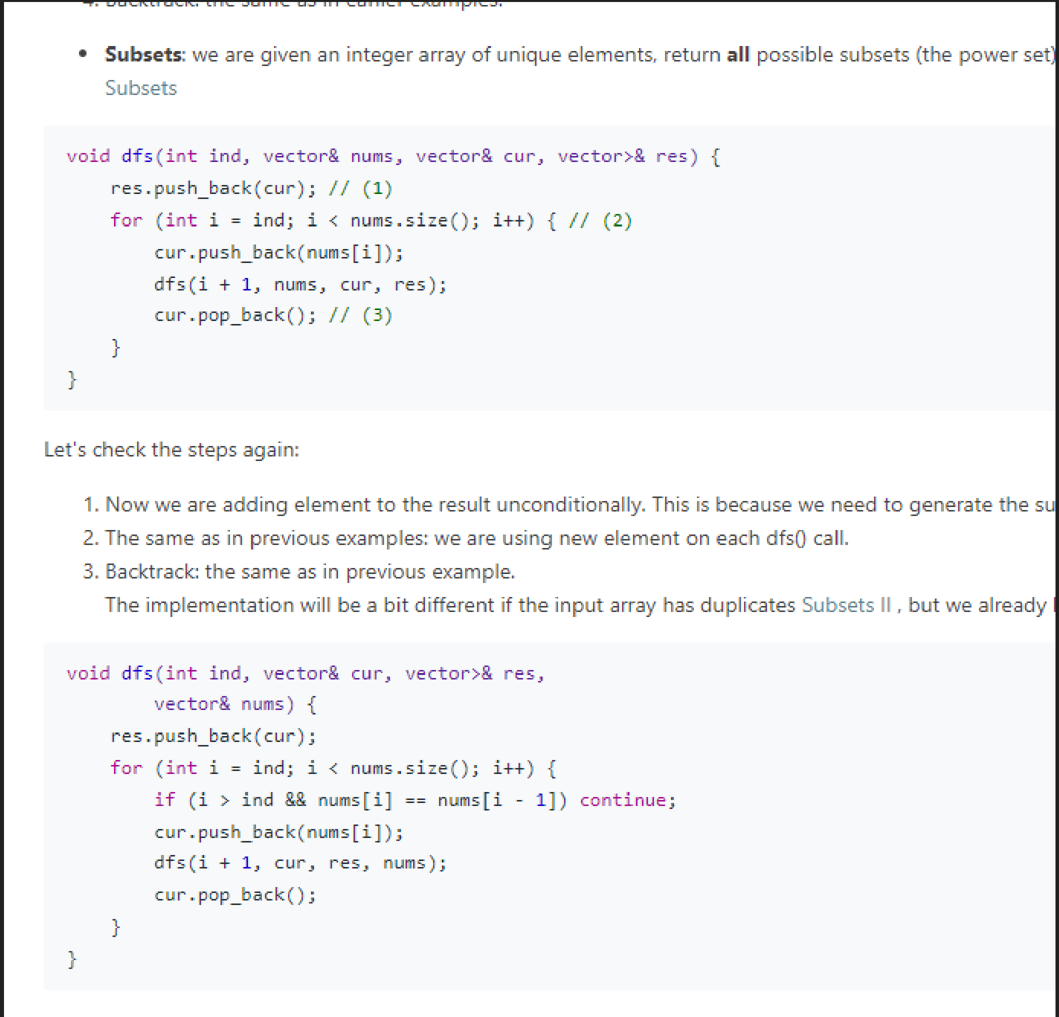
Another variation of the problem is Permutations II.
The only difference between first problem is that we **MAY** have **DUPLICATES** in the input array.

```
void backtrack(vector& nums, vector& cur, vector>& res,
         unordered_map& hmap) {
    if (cur.size() == nums.size()) { // (1)
        res.push_back(cur);
        return;
    }
    for (auto& [num, freq] : hmap) { // (2)
        if (freq == 0) continue;     // (3)
        freq--;
        cur.push_back(num);
        dfs(nums, cur, res, hmap);
        cur.pop_back();              // (4)
        freq++;
    }
}
// Iterate over the original list, but check if the previous element is the same as current.
// We need to make this check because using the same element will give us the same result as last iteration.
void backtrack2(vector<int>& nums, vector<int>& temp,
                  vector<vector<int>>& res, unordered_map<int, int>& freq) {
        if (temp.size() == nums.size()) {
            res.push_back(temp);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (freq[nums[i]] == 0 || (i != 0 && nums[i] == nums[i - 1])) continue;
            temp.push_back(nums[i]);
            freq[nums[i]]--;
            backtrack(nums, temp, res, freq);
            freq[nums[i]]++;
            temp.pop_back();
        }
    }
```



- **Combinations:** we are given two integers n and k, return **ALL** possible combinations of k numbers out of the range [1, n]. If y solution, here is the link: Combinations

```
void backtrack(int num, int last, int k, vector& cur,
        vector<vector<int>>& res) {
    if (cur.size() == k) {  // (1)
        res.push_back(cur);
        return;
    }
    for (int i = num; i <= last; i++) { // (2)
        cur.push_back(i);
        backtrack(i + 1, last, k, cur, res);
        cur.pop_back();
    }
}
// Or we can allocate temp vector in advance and fill the position.
void backtrack2(int ind, int prev, int k, int n, vector<int>& temp,
               vector<vector<int>>& res) {
    if (ind >= k) {
        res.push_back(temp);
        return;
    }
    for (int p = prev + 1; p <= n; p++) {
        int saved = temp[ind]; // Given the way how we fill temp array - this is not necessary.
        temp[ind] = p;
        backtrack2(ind + 1, p, k, n, temp, res);
        temp[ind] = saved;     // Given the way how we fill temp array - this is not necessary.
    }
}
```



4. Backtrack: the same as in earlier examples.

- **Subsets**: we are given an integer array of unique elements, return **all** possible subsets (the power set)
  Subsets

```
void dfs(int ind, vector& nums, vector& cur, vector>& res) {
    res.push_back(cur); // (1)
    for (int i = ind; i < nums.size(); i++) { // (2)
        cur.push_back(nums[i]);
        dfs(i + 1, nums, cur, res);
        cur.pop_back(); // (3)
    }
}
```

Let's check the steps again:

1. Now we are adding element to the result unconditionally. This is because we need to generate the su
2. The same as in previous examples: we are using new element on each dfs() call.
3. Backtrack: the same as in previous example.
   The implementation will be a bit different if the input array has duplicates Subsets II , but we already

```
void dfs(int ind, vector& cur, vector>& res,
        vector& nums) {
    res.push_back(cur);
    for (int i = ind; i < nums.size(); i++) {
        if (i > ind && nums[i] == nums[i - 1]) continue;
        cur.push_back(nums[i]);
        dfs(i + 1, cur, res, nums);
        cur.pop_back();
    }
}
```

DYNAMIC PROGRAMMING

Longest increasing subsequence

Longest Increasing Subsequence

https://leetcode.com/problems/longest-increasing-subsequence/

https://leetcode.com/problems/largest-divisible-subset/

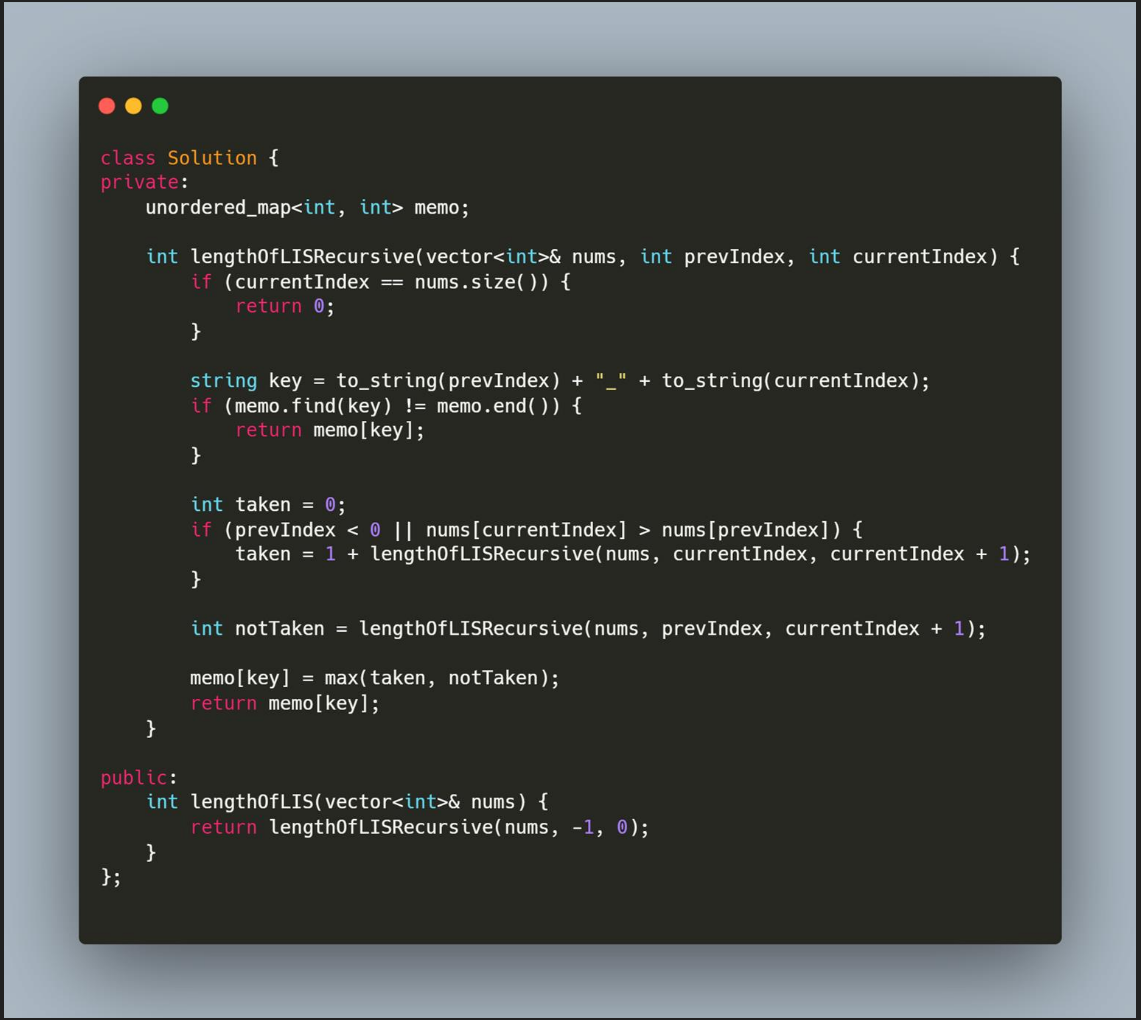
https://leetcode.com/problems/russian-doll-envelopes/

https://leetcode.com/problems/maximum-length-of-pair-chain/

https://leetcode.com/problems/number-of-longest-increasing-subsequence/

https://leetcode.com/problems/delete-and-earn/

https://leetcode.com/problems/longest-string-chain/



```
class Solution {
private:
    unordered_map<int, int> memo;

    int lengthOfLISRecursive(vector<int>& nums, int prevIndex, int currentIndex) {
        if (currentIndex == nums.size()) {
            return 0;
        }

        string key = to_string(prevIndex) + "_" + to_string(currentIndex);
        if (memo.find(key) != memo.end()) {
            return memo[key];
        }

        int taken = 0;
        if (prevIndex < 0 || nums[currentIndex] > nums[prevIndex]) {
            taken = 1 + lengthOfLISRecursive(nums, currentIndex, currentIndex + 1);
        }

        int notTaken = lengthOfLISRecursive(nums, prevIndex, currentIndex + 1);

        memo[key] = max(taken, notTaken);
        return memo[key];
    }

public:
    int lengthOfLIS(vector<int>& nums) {
        return lengthOfLISRecursive(nums, -1, 0);
    }
};
```

Longest common subsequence

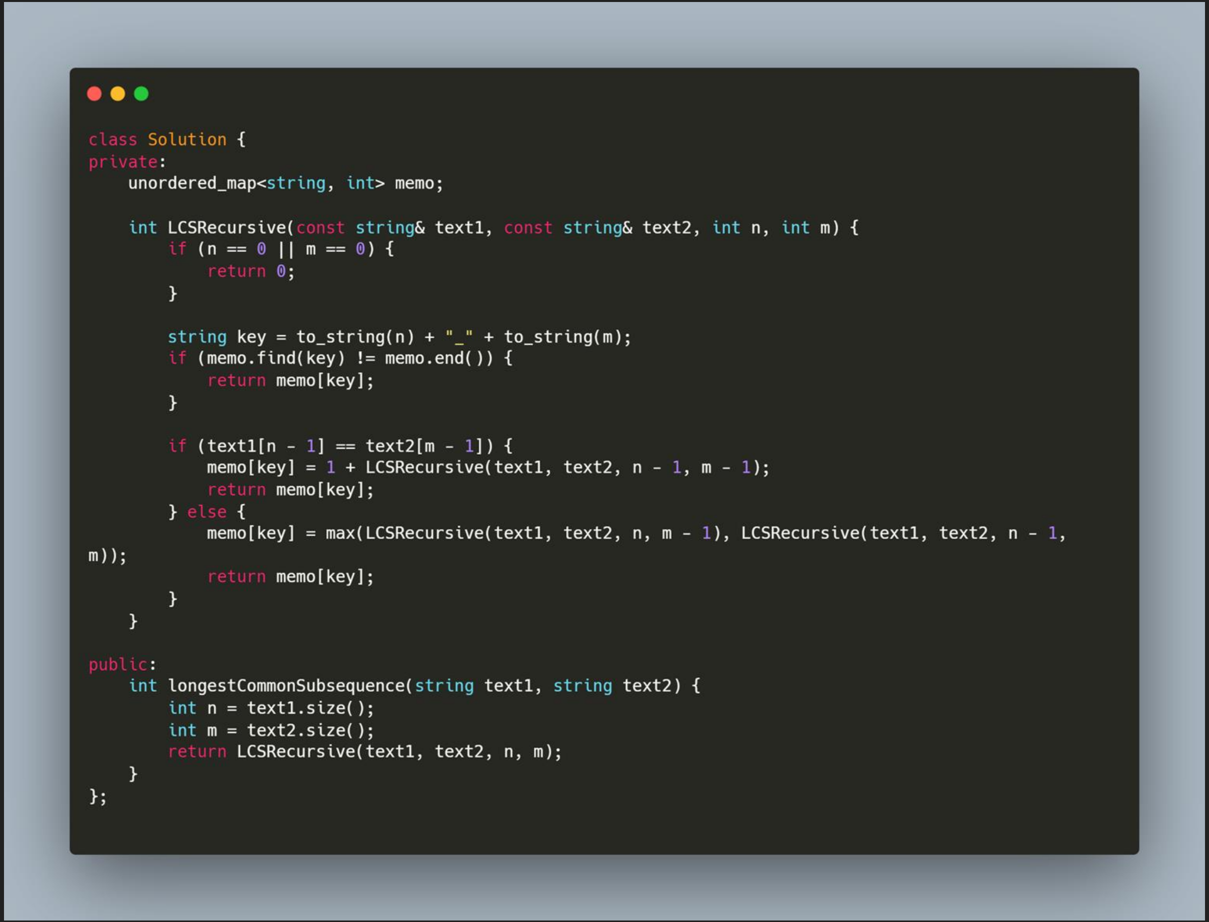
Longest Common Subsequence

https://leetcode.com/problems/longest-common-subsequence/

https://leetcode.com/problems/edit-distance/

https://leetcode.com/problems/distinct-subsequences/

https://leetcode.com/problems/minimum-ascii-delete-sum-for-two-strings/



```
class Solution {
private:
    unordered_map<string, int> memo;

    int LCSRecursive(const string& text1, const string& text2, int n, int m) {
        if (n == 0 || m == 0) {
            return 0;
        }

        string key = to_string(n) + "_" + to_string(m);
        if (memo.find(key) != memo.end()) {
            return memo[key];
        }

        if (text1[n - 1] == text2[m - 1]) {
            memo[key] = 1 + LCSRecursive(text1, text2, n - 1, m - 1);
            return memo[key];
        } else {
            memo[key] = max(LCSRecursive(text1, text2, n, m - 1), LCSRecursive(text1, text2, n - 1,
m));
            return memo[key];
        }
    }

public:
    int longestCommonSubsequence(string text1, string text2) {
        int n = text1.size();
        int m = text2.size();
        return LCSRecursive(text1, text2, n, m);
    }
};
```

Coin change

Coin Change:

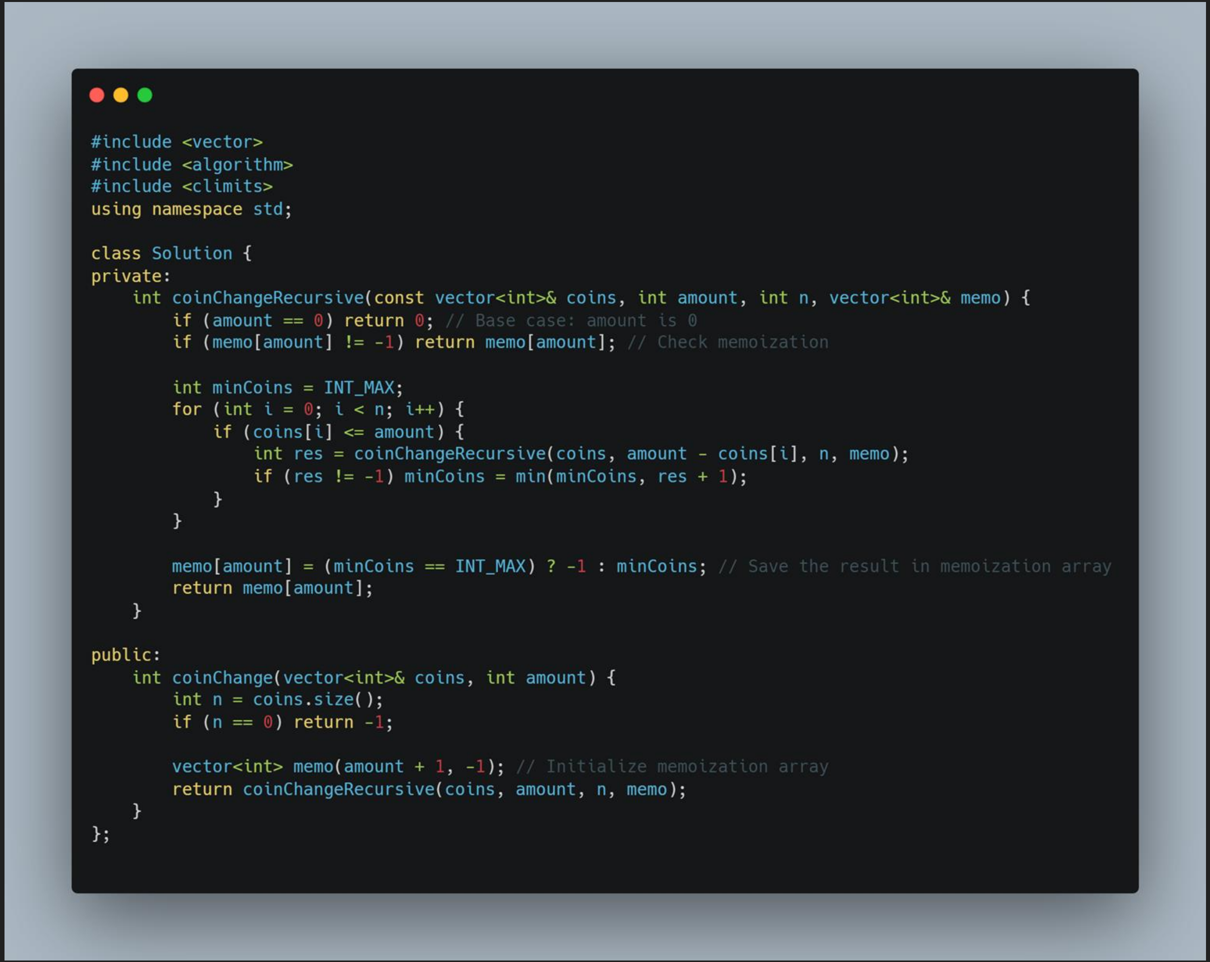
https://leetcode.com/problems/coin-change/

https://leetcode.com/problems/coin-change-2/

https://leetcode.com/problems/combination-sum-iv/

https://leetcode.com/problems/perfect-squares/

https://leetcode.com/problems/minimum-cost-for-tickets/



```cpp
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

class Solution {
private:
    int coinChangeRecursive(const vector<int>& coins, int amount, int n, vector<int>& memo) {
        if (amount == 0) return 0; // Base case: amount is 0
        if (memo[amount] != -1) return memo[amount]; // Check memoization

        int minCoins = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (coins[i] <= amount) {
                int res = coinChangeRecursive(coins, amount - coins[i], n, memo);
                if (res != -1) minCoins = min(minCoins, res + 1);
            }
        }

        memo[amount] = (minCoins == INT_MAX) ? -1 : minCoins; // Save the result in memoization array
        return memo[amount];
    }

public:
    int coinChange(vector<int>& coins, int amount) {
        int n = coins.size();
        if (n == 0) return -1;

        vector<int> memo(amount + 1, -1); // Initialize memoization array
        return coinChangeRecursive(coins, amount, n, memo);
    }
};
```

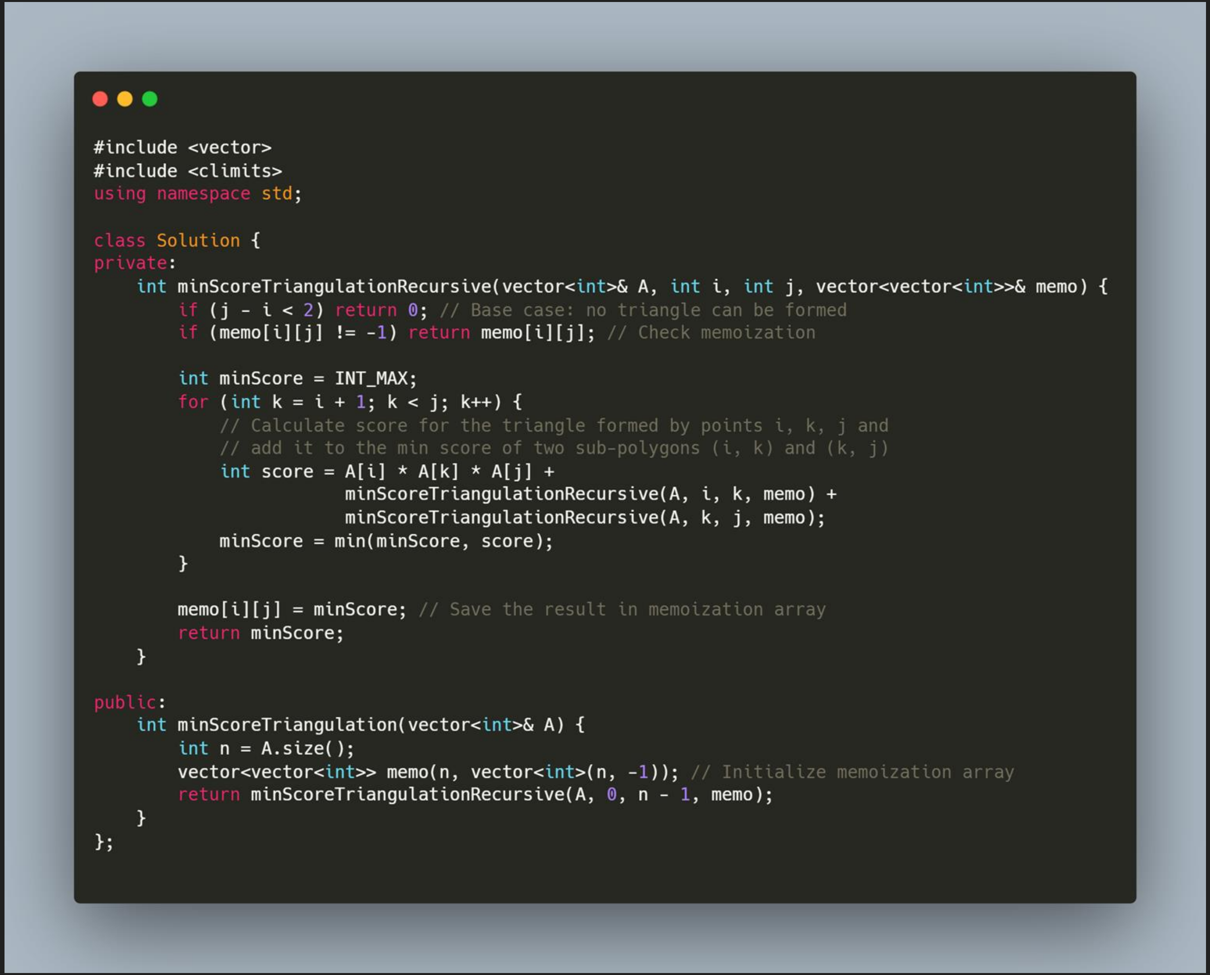
Matrix multiply

Matrix multiplication:

https://leetcode.com/problems/minimum-score-triangulation-of-polygon/

https://leetcode.com/problems/minimum-cost-tree-from-leaf-values/

https://leetcode.com/problems/burst-balloons/



```
#include <vector>
#include <climits>
using namespace std;

class Solution {
private:
    int minScoreTriangulationRecursive(vector<int>& A, int i, int j, vector<vector<int>>& memo) {
        if (j - i < 2) return 0; // Base case: no triangle can be formed
        if (memo[i][j] != -1) return memo[i][j]; // Check memoization

        int minScore = INT_MAX;
        for (int k = i + 1; k < j; k++) {
            // Calculate score for the triangle formed by points i, k, j and
            // add it to the min score of two sub-polygons (i, k) and (k, j)
            int score = A[i] * A[k] * A[j] +
                        minScoreTriangulationRecursive(A, i, k, memo) +
                        minScoreTriangulationRecursive(A, k, j, memo);
            minScore = min(minScore, score);
        }

        memo[i][j] = minScore; // Save the result in memoization array
        return minScore;
    }

public:
    int minScoreTriangulation(vector<int>& A) {
        int n = A.size();
        vector<vector<int>> memo(n, vector<int>(n, -1)); // Initialize memoization array
        return minScoreTriangulationRecursive(A, 0, n - 1, memo);
    }
};
```

Matrix 2d array

Matrix/2D Array:

https://leetcode.com/problems/matrix-block-sum/

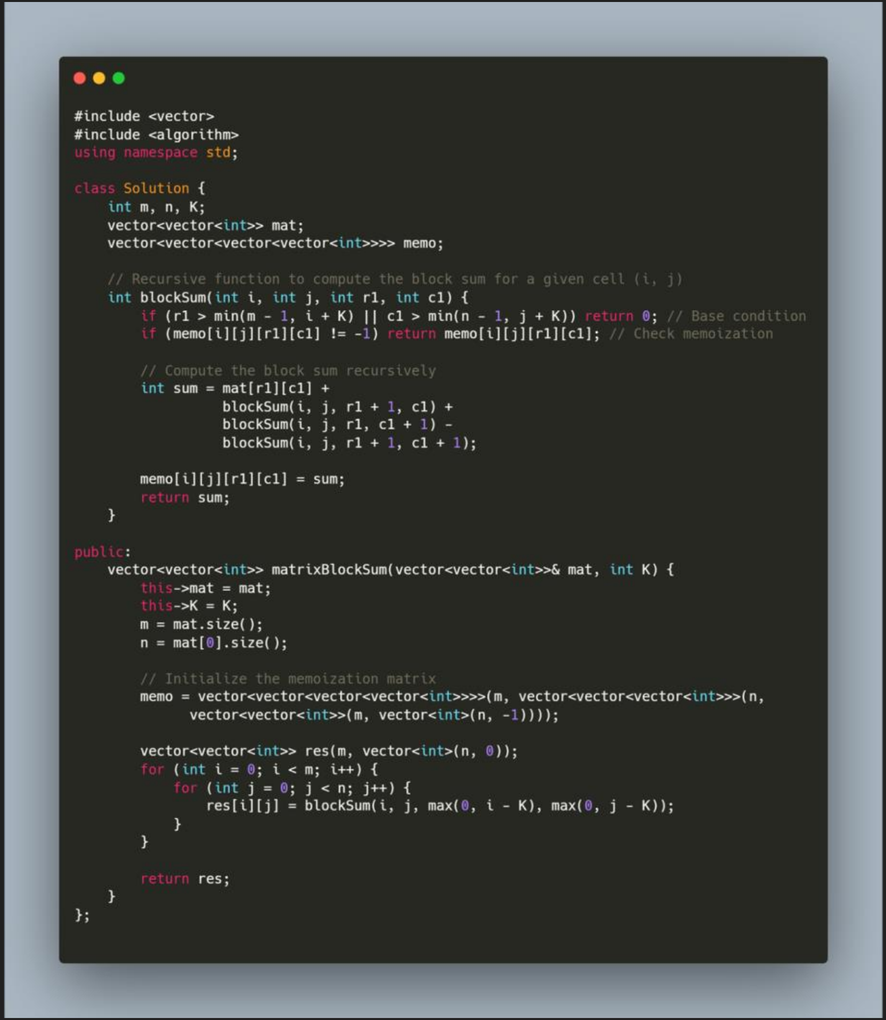
https://leetcode.com/problems/range-sum-query-2d-immutable/

https://leetcode.com/problems/dungeon-game/

https://leetcode.com/problems/triangle/

https://leetcode.com/problems/maximal-square/

https://leetcode.com/problems/minimum-falling-path-sum/



```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
    int m, n, K;
    vector<vector<int>> mat;
    vector<vector<vector<vector<int>>>> memo;

    // Recursive function to compute the block sum for a given cell (i, j)
    int blockSum(int i, int j, int r1, int c1) {
        if (r1 > min(m - 1, i + K) || c1 > min(n - 1, j + K)) return 0; // Base condition
        if (memo[i][j][r1][c1] != -1) return memo[i][j][r1][c1]; // Check memoization

        // Compute the block sum recursively
        int sum = mat[r1][c1] +
                  blockSum(i, j, r1 + 1, c1) +
                  blockSum(i, j, r1, c1 + 1) -
                  blockSum(i, j, r1 + 1, c1 + 1);

        memo[i][j][r1][c1] = sum;
        return sum;
    }

public:
    vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {
        this->mat = mat;
        this->K = K;
        m = mat.size();
        n = mat[0].size();

        // Initialize the memoization matrix
        memo = vector<vector<vector<vector<int>>>>(m, vector<vector<vector<int>>>(n,
              vector<vector<int>>(m, vector<int>(n, -1))));

        vector<vector<int>> res(m, vector<int>(n, 0));
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                res[i][j] = blockSum(i, j, max(0, i - K), max(0, j - K));
            }
        }

        return res;
    }
};
```

Hash DP

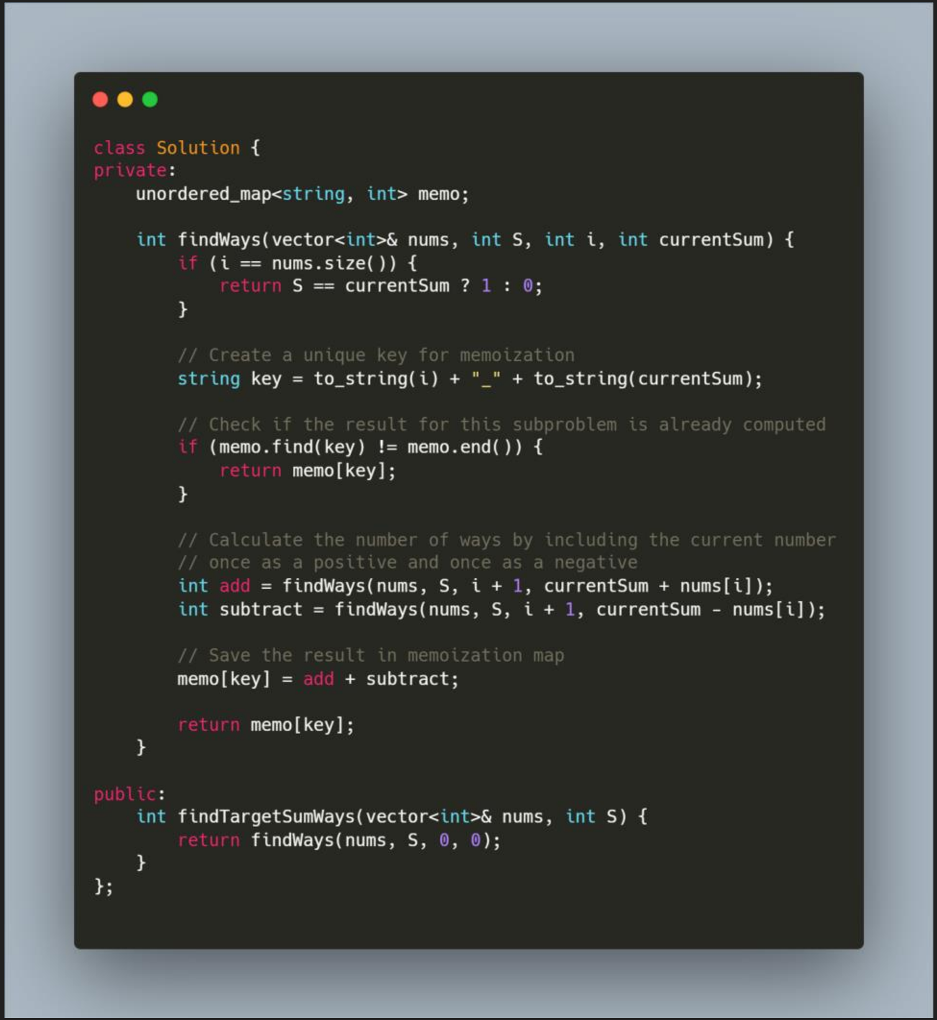
Hash + DP:

https://leetcode.com/problems/target-sum/

https://leetcode.com/problems/longest-arithmetic-sequence/

https://leetcode.com/problems/longest-arithmetic-subsequence-of-given-difference/

https://leetcode.com/problems/maximum-product-of-splitted-binary-tree/



```
class Solution {
private:
    unordered_map<string, int> memo;

    int findWays(vector<int>& nums, int S, int i, int currentSum) {
        if (i == nums.size()) {
            return S == currentSum ? 1 : 0;
        }

        // Create a unique key for memoization
        string key = to_string(i) + "_" + to_string(currentSum);

        // Check if the result for this subproblem is already computed
        if (memo.find(key) != memo.end()) {
            return memo[key];
        }

        // Calculate the number of ways by including the current number
        // once as a positive and once as a negative
        int add = findWays(nums, S, i + 1, currentSum + nums[i]);
        int subtract = findWays(nums, S, i + 1, currentSum - nums[i]);

        // Save the result in memoization map
        memo[key] = add + subtract;

        return memo[key];
    }

public:
    int findTargetSumWays(vector<int>& nums, int S) {
        return findWays(nums, S, 0, 0);
    }
};
```

State machine

State machine:

https://leetcode.com/problems/best-time-to-buy-and-sell-stock/

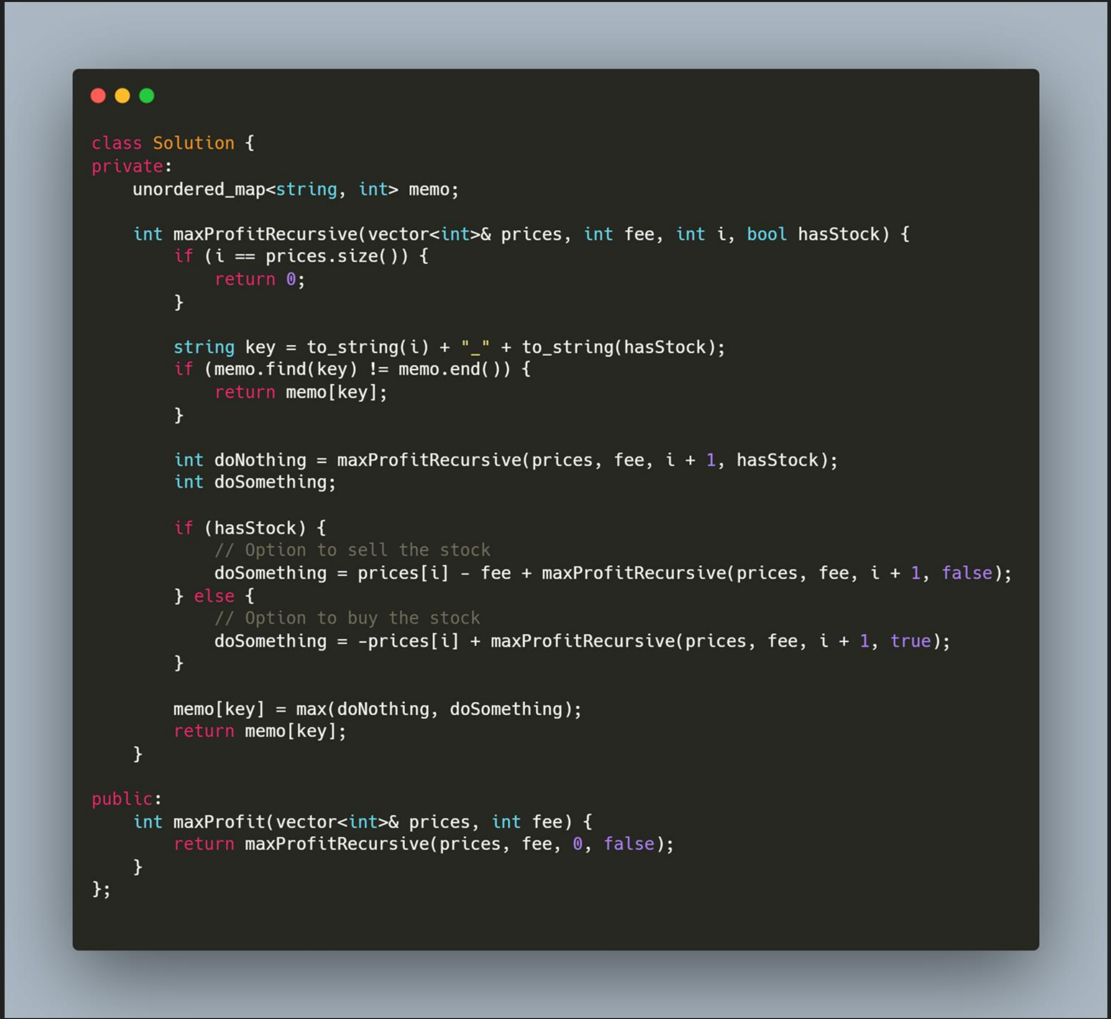
https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/

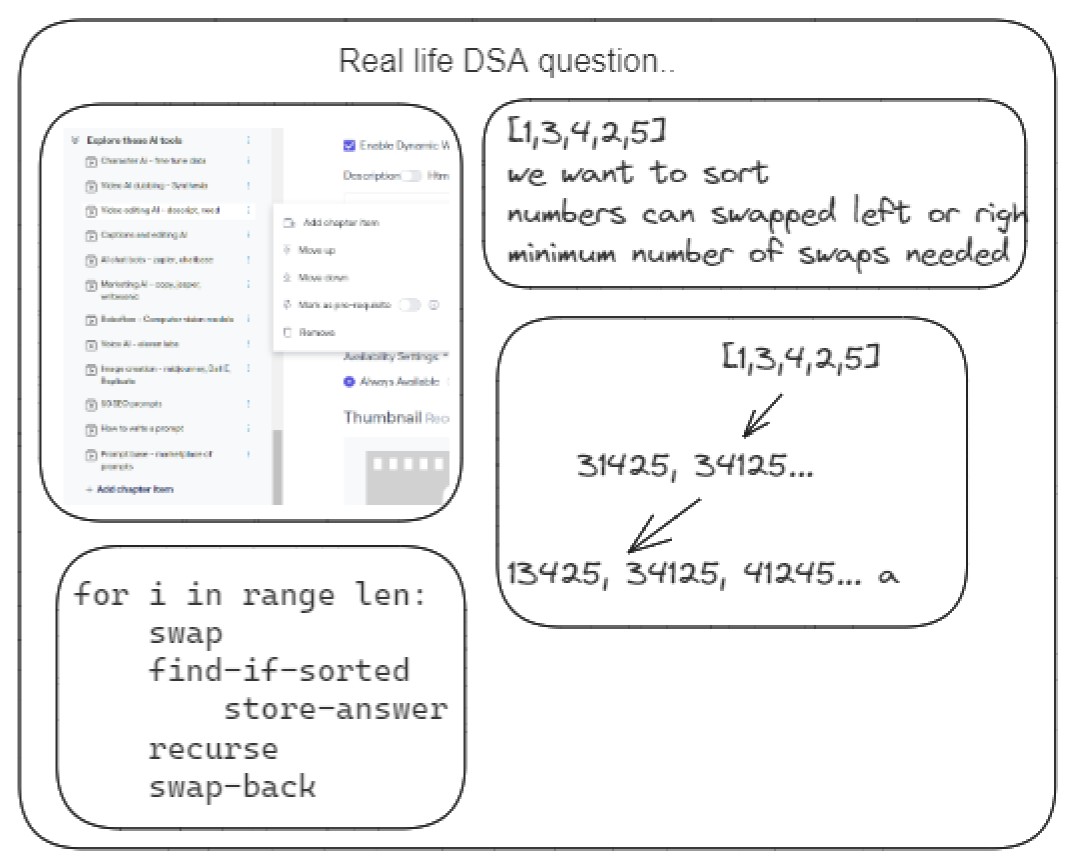
https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/



```
class Solution {
private:
    unordered_map<string, int> memo;

    int maxProfitRecursive(vector<int>& prices, int fee, int i, bool hasStock) {
        if (i == prices.size()) {
            return 0;
        }

        string key = to_string(i) + "_" + to_string(hasStock);
        if (memo.find(key) != memo.end()) {
            return memo[key];
        }

        int doNothing = maxProfitRecursive(prices, fee, i + 1, hasStock);
        int doSomething;

        if (hasStock) {
            // Option to sell the stock
            doSomething = prices[i] - fee + maxProfitRecursive(prices, fee, i + 1, false);
        } else {
            // Option to buy the stock
            doSomething = -prices[i] + maxProfitRecursive(prices, fee, i + 1, true);
        }

        memo[key] = max(doNothing, doSomething);
        return memo[key];
    }

public:
    int maxProfit(vector<int>& prices, int fee) {
        return maxProfitRecursive(prices, fee, 0, false);
    }
};
```

Real life



Real life DSA question..

Explore these AI tools
Character AI - fine tune data
Video AI dubbing - Synthesia
Video editing AI - descript, reed
Captions and editing AI
AI chat bots - zapier, chatbase
Marketing AI - copy, jasper, writesonic
Roboflow - Computer vision models
Voice AI - eleven labs
Image creation - midjourney, Dall E, Replicate
50 SEO prompts
How to write a prompt
Prompt base - marketplace of prompts
+ Add chapter item
Add chapter item
Move up
Move down
Mark as pre-requisite
Remove
☑ Enable Dynamic W
Description
Htm
Availability Settings *
● Always Available
Thumbnail Rec

[1,3,4,2,5]
we want to sort
numbers can swapped left or righ
minimum number of swaps needed

[1,3,4,2,5]

31425, 34125...

13425, 34125, 41245... a

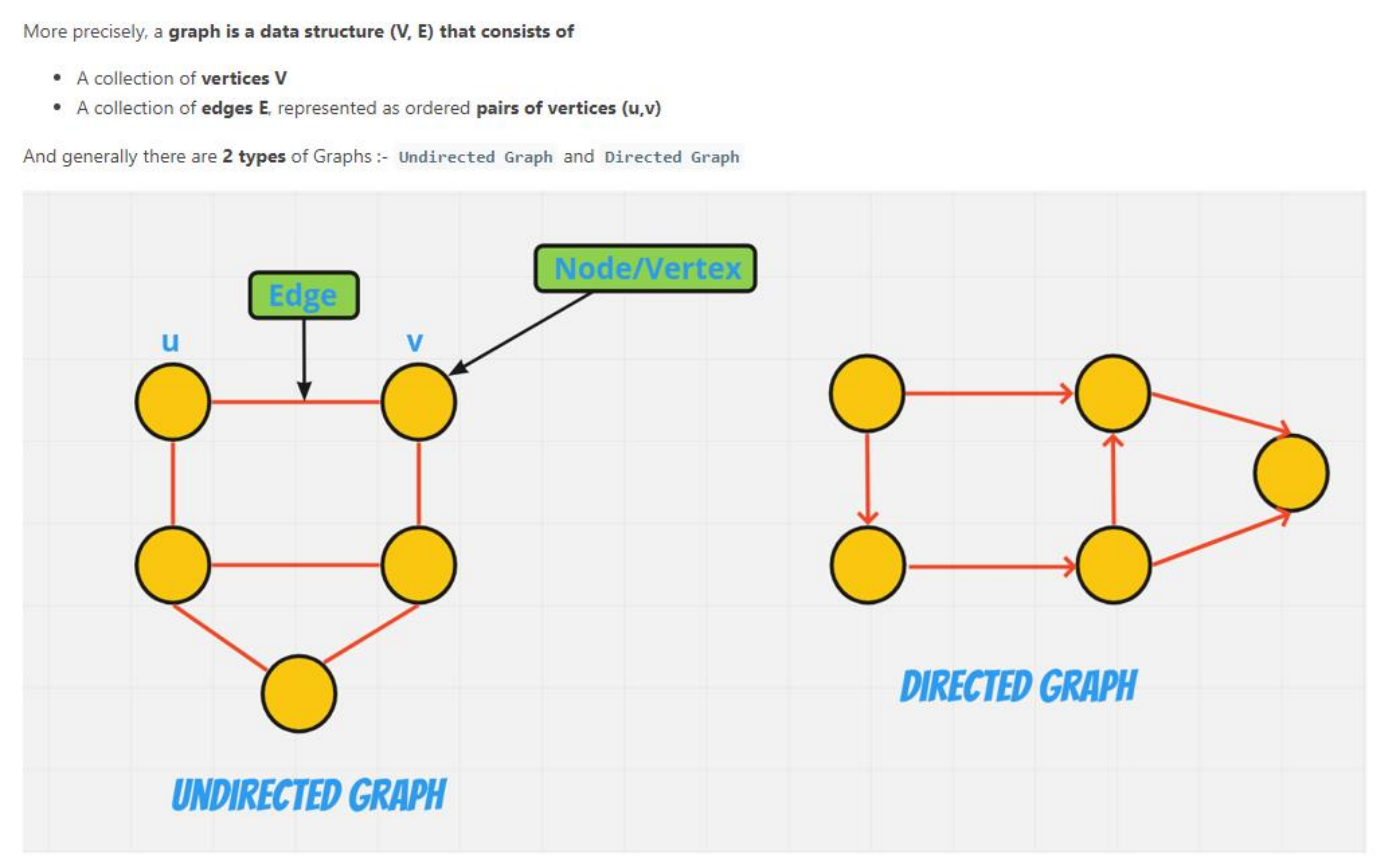
```
for i in range len:
    swap
    find-if-sorted
        store-answer
    recurse
    swap-back
```

Leetcode articles

https://leetcode.com/discuss/general-discussion/665604/Important-and-Useful-links-from-all-over-the-Leetcode

https://leetcode.com/discuss/general-discussion/458695/dynamic-programming-patterns

GRAPHS A-Z



More precisely, a **graph is a data structure (V, E) that consists of**

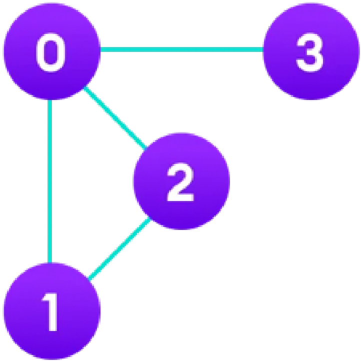
- A collection of **vertices V**
- A collection of **edges E**, represented as ordered **pairs of vertices (u,v)**

And generally there are **2 types** of Graphs :- `Undirected Graph` and `Directed Graph`

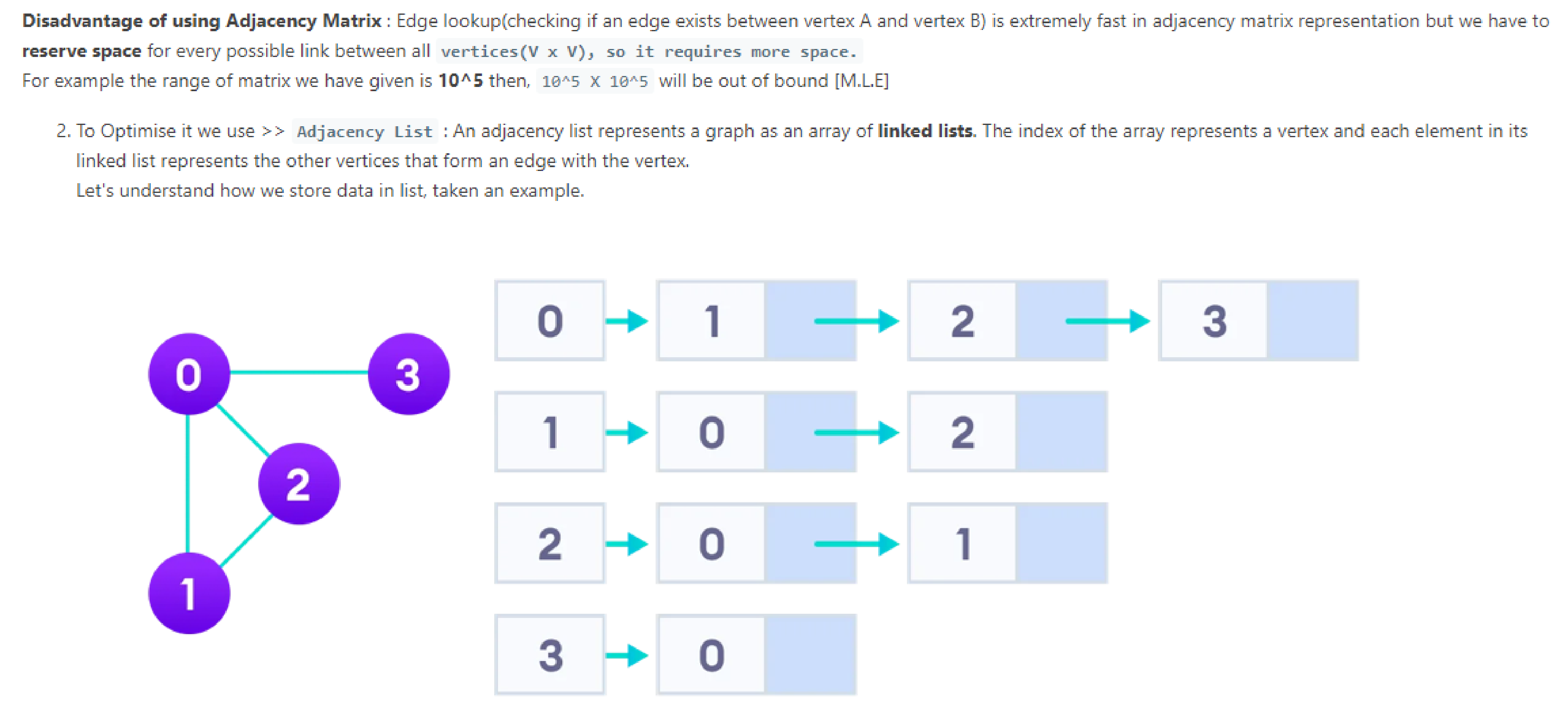
There are two ways to store Graph in a Data Structure.

1. `Adjacency Matrix` :- An adjacency matrix is a **2D array of V x V vertices**. Each row and column represent a vertex. If the value of any element `a[i][j]` is **1**, it represents that there is an edge connecting **vertex i** and **vertex j**.
   Let's understand how we store data in matrix.

- First of all, create a matrix of `n + 1 X n + 1`
- And Intially our 2-D Array will be filled with `0`
- Since it is an **undirected graph**, for edge `(0,2)`, we also need to mark edge `(2,0)`
- And as weight of the edge is not given, we'll mark it with `1`



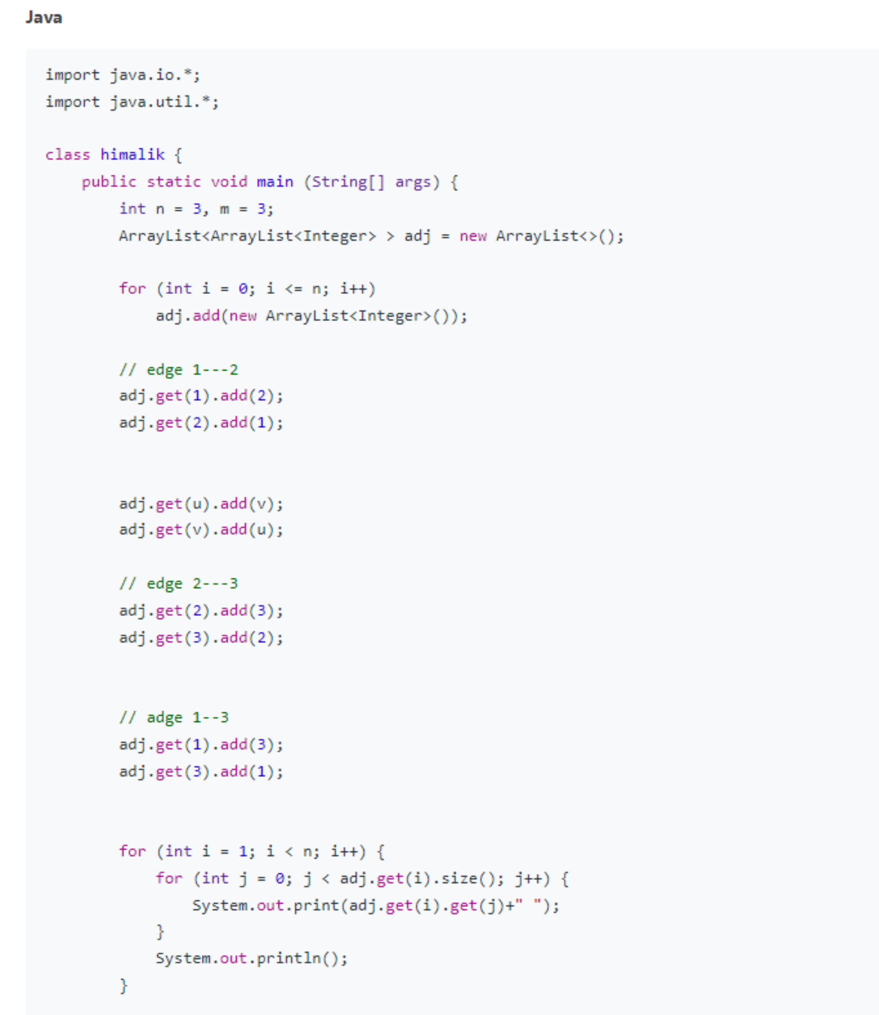
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0



Disadvantage of using Adjacency Matrix : Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to **reserve space** for every possible link between all `vertices(V x V), so it requires more space.`
For example the range of matrix we have given is **10^5** then, `10^5 X 10^5` will be out of bound [M.L.E]

2. To Optimise it we use >> `Adjacency List` : An adjacency list represents a graph as an array of **linked lists**. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
   Let's understand how we store data in list, taken an example.

0 → 1 → 2 → 3
1 → 0 → 2
2 → 0 → 1
3 → 0



Java

```java
import java.io.*;
import java.util.*;

class himalik {
    public static void main (String[] args) {
        int n = 3, m = 3;
        ArrayList<ArrayList<Integer> > adj = new ArrayList<>();

        for (int i = 0; i <= n; i++)
            adj.add(new ArrayList<Integer>());

        // edge 1---2
        adj.get(1).add(2);
        adj.get(2).add(1);


        adj.get(u).add(v);
        adj.get(v).add(u);

        // edge 2---3
        adj.get(2).add(3);
        adj.get(3).add(2);


        // adge 1--3
        adj.get(1).add(3);
        adj.get(3).add(1);


        for (int i = 1; i < n; i++) {
            for (int j = 0; j < adj.get(i).size(); j++) {
                System.out.print(adj.get(i).get(j)+" ");
            }
            System.out.println();
        }
```

LEETCODE ARTICLES

https://leetcode.com/discuss/general-discussion/655708/Graph-For-Beginners-Problems-or-Pattern-or-Sample-Solutions

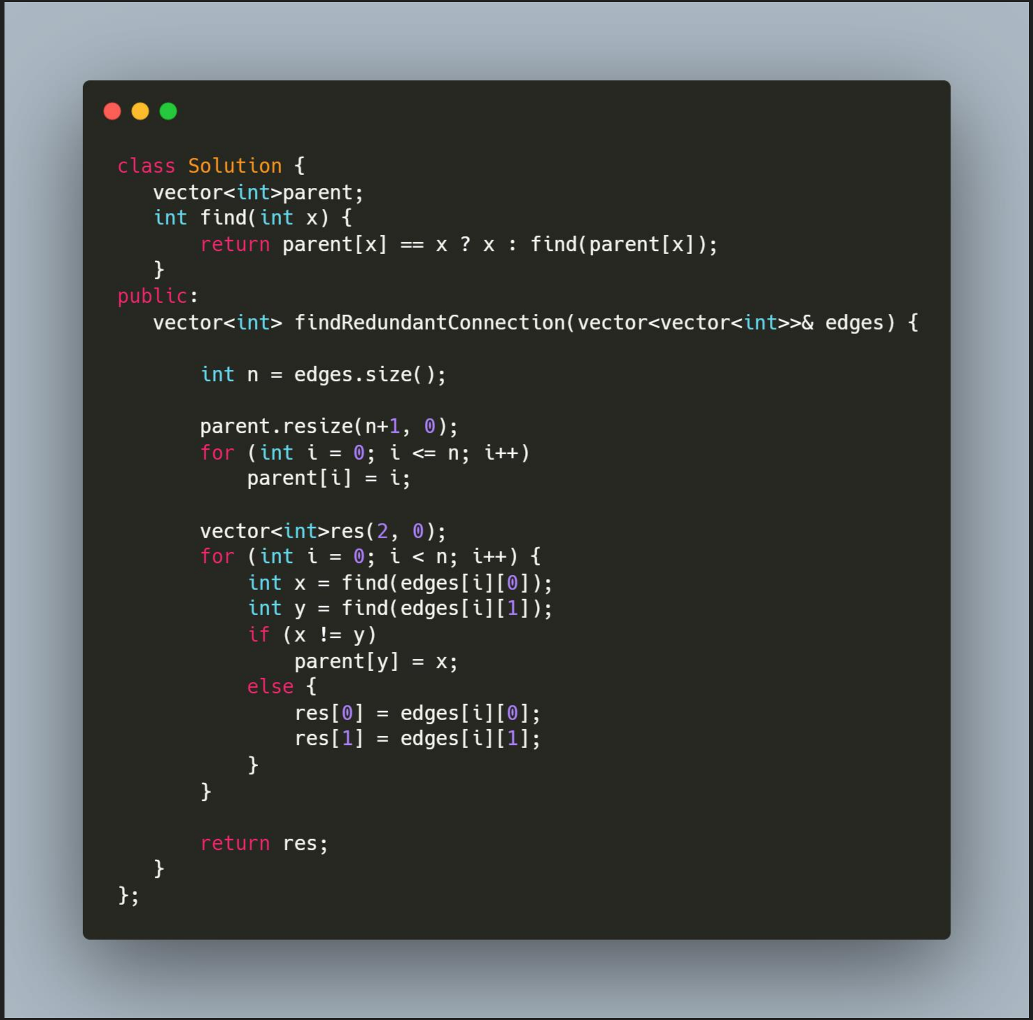
https://leetcode.com/discuss/study-guide/2360573/Become-Master-In-Graph

UNION FIND

Union Find:

Identify if problems talks about finding groups or components.

https://leetcode.com/problems/friend-circles/
https://leetcode.com/problems/redundant-connection/
https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/
https://leetcode.com/problems/number-of-operations-to-make-network-connected/
https://leetcode.com/problems/satisfiability-of-equality-equations/
https://leetcode.com/problems/accounts-merge/



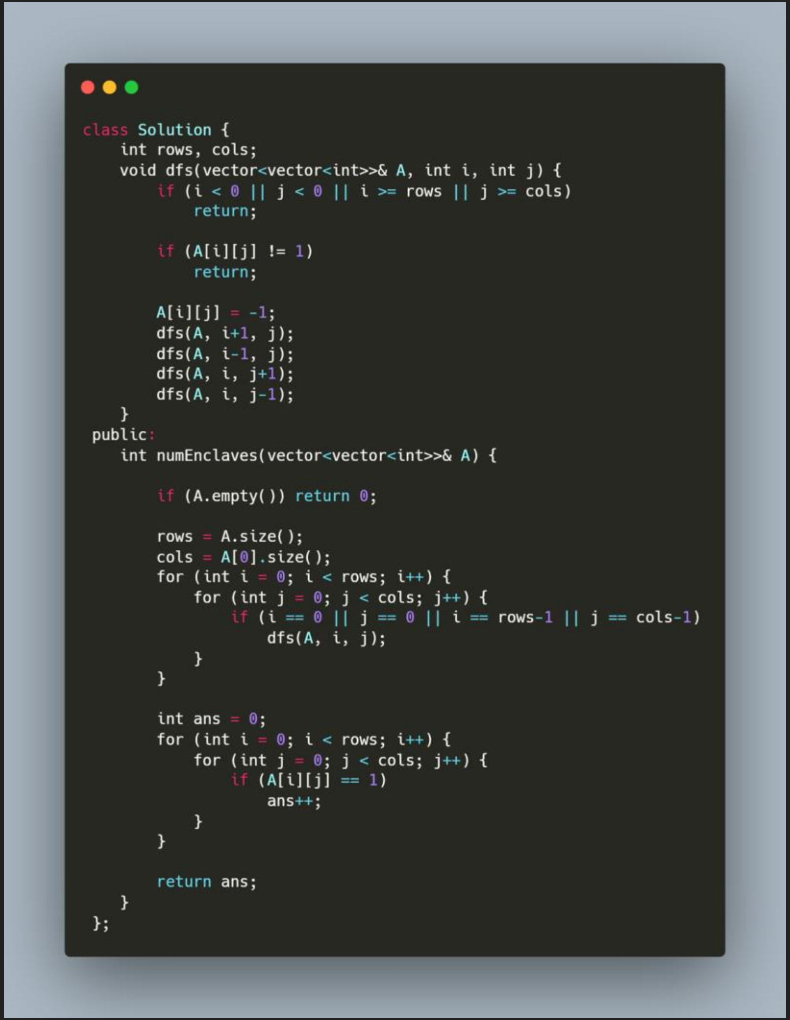
```
class Solution {
   vector<int>parent;
   int find(int x) {
       return parent[x] == x ? x : find(parent[x]);
   }
public:
   vector<int> findRedundantConnection(vector<vector<int>>& edges) {

       int n = edges.size();

       parent.resize(n+1, 0);
       for (int i = 0; i <= n; i++)
           parent[i] = i;

       vector<int>res(2, 0);
       for (int i = 0; i < n; i++) {
           int x = find(edges[i][0]);
           int y = find(edges[i][1]);
           if (x != y)
               parent[y] = x;
           else {
               res[0] = edges[i][0];
               res[1] = edges[i][1];
           }
       }

       return res;
   }
};
```

DFS

Start DFS from nodes at boundary:

https://leetcode.com/problems/surrounded-regions/

https://leetcode.com/problems/number-of-enclaves/



```
class Solution {
    int rows, cols;
    void dfs(vector<vector<int>>& A, int i, int j) {
        if (i < 0 || j < 0 || i >= rows || j >= cols)
            return;

        if (A[i][j] != 1)
            return;

        A[i][j] = -1;
        dfs(A, i+1, j);
        dfs(A, i-1, j);
        dfs(A, i, j+1);
        dfs(A, i, j-1);
    }
public:
    int numEnclaves(vector<vector<int>>& A) {

        if (A.empty()) return 0;

        rows = A.size();
        cols = A[0].size();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i == 0 || j == 0 || i == rows-1 || j == cols-1)
                    dfs(A, i, j);
            }
        }

        int ans = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (A[i][j] == 1)
                    ans++;
            }
        }

        return ans;
    }
};
```

DFS PART 2

DFS from each unvisited node/Island problems

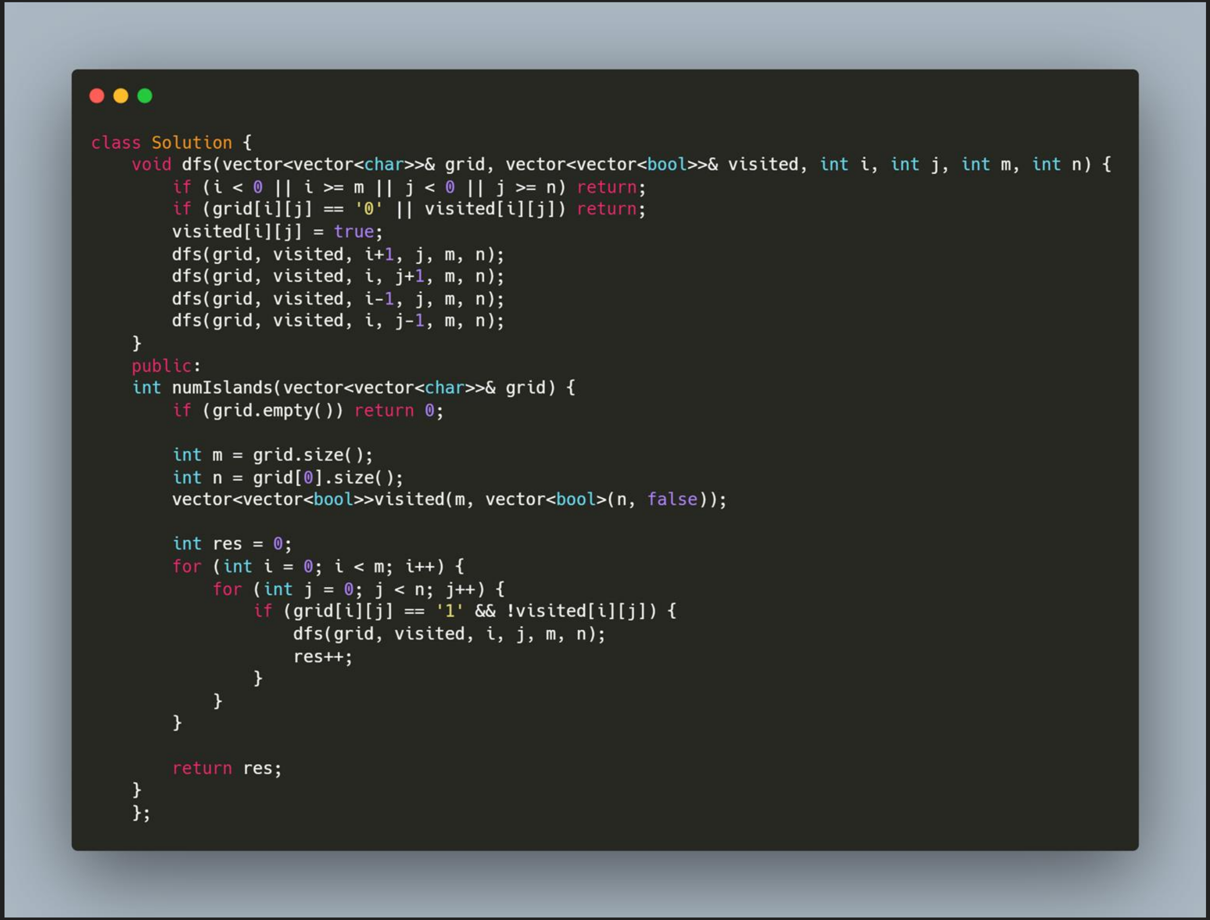
https://leetcode.com/problems/number-of-closed-islands/

https://leetcode.com/problems/number-of-islands/

https://leetcode.com/problems/keys-and-rooms/

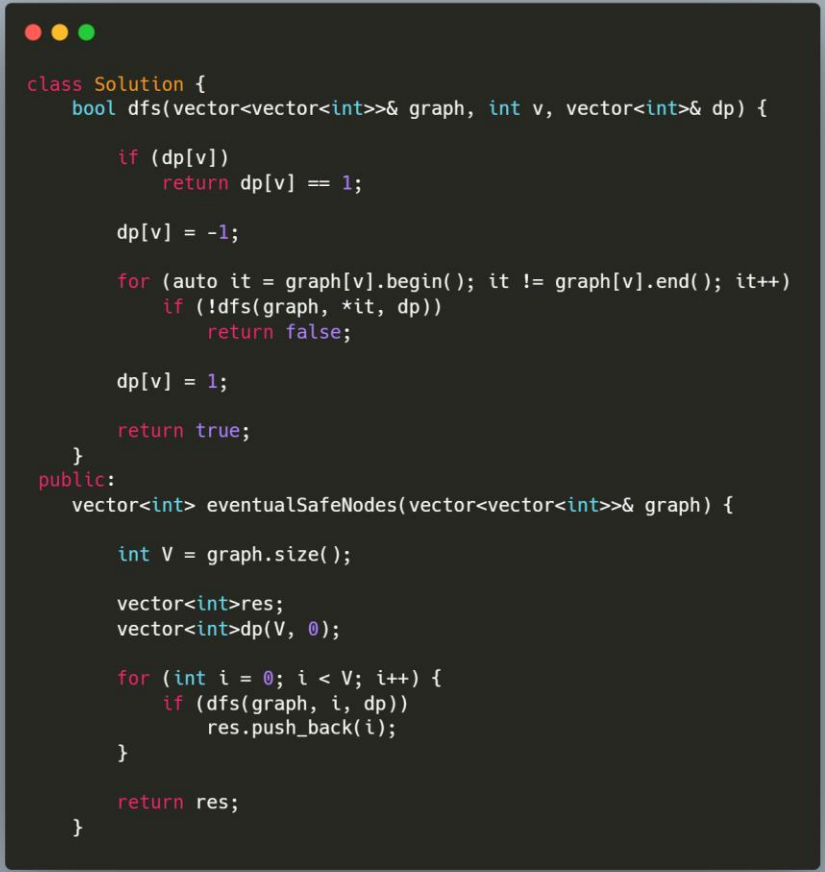
https://leetcode.com/problems/max-area-of-island/

https://leetcode.com/problems/flood-fill/



```
class Solution {
    void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int j, int m, int n) {
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == '0' || visited[i][j]) return;
        visited[i][j] = true;
        dfs(grid, visited, i+1, j, m, n);
        dfs(grid, visited, i, j+1, m, n);
        dfs(grid, visited, i-1, j, m, n);
        dfs(grid, visited, i, j-1, m, n);
    }
    public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty()) return 0;

        int m = grid.size();
        int n = grid[0].size();
        vector<vector<bool>>visited(m, vector<bool>(n, false));

        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    dfs(grid, visited, i, j, m, n);
                    res++;
                }
            }
        }

        return res;
    }
    };
```

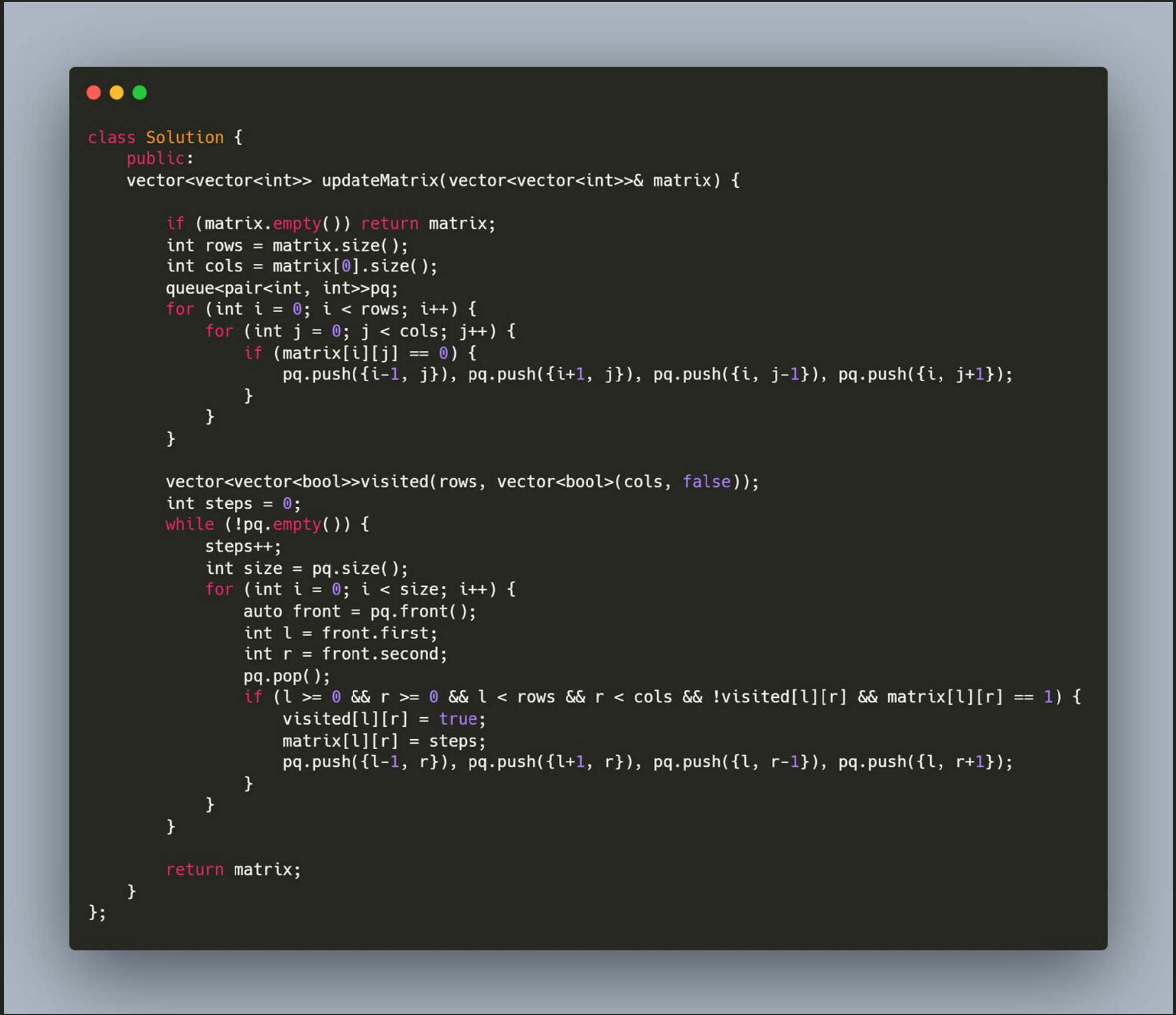
CYCLE FINDING



```
class Solution {
    bool dfs(vector<vector<int>>& graph, int v, vector<int>& dp) {

        if (dp[v])
            return dp[v] == 1;

        dp[v] = -1;

        for (auto it = graph[v].begin(); it != graph[v].end(); it++)
            if (!dfs(graph, *it, dp))
                return false;

        dp[v] = 1;

        return true;
    }
 public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {

        int V = graph.size();

        vector<int>res;
        vector<int>dp(V, 0);

        for (int i = 0; i < V; i++) {
            if (dfs(graph, i, dp))
                res.push_back(i);
        }

        return res;
    }
```

Breadth First Search

1. Shortest Path: https://leetcode.com/problems/01-matrix/ https://leetcode.com/problems/as-far-from-land-as-possible/ https://leetcode.com/problems/rotting-oranges/ https://leetcode.com/problems/shortest-path-in-binary-matrix/

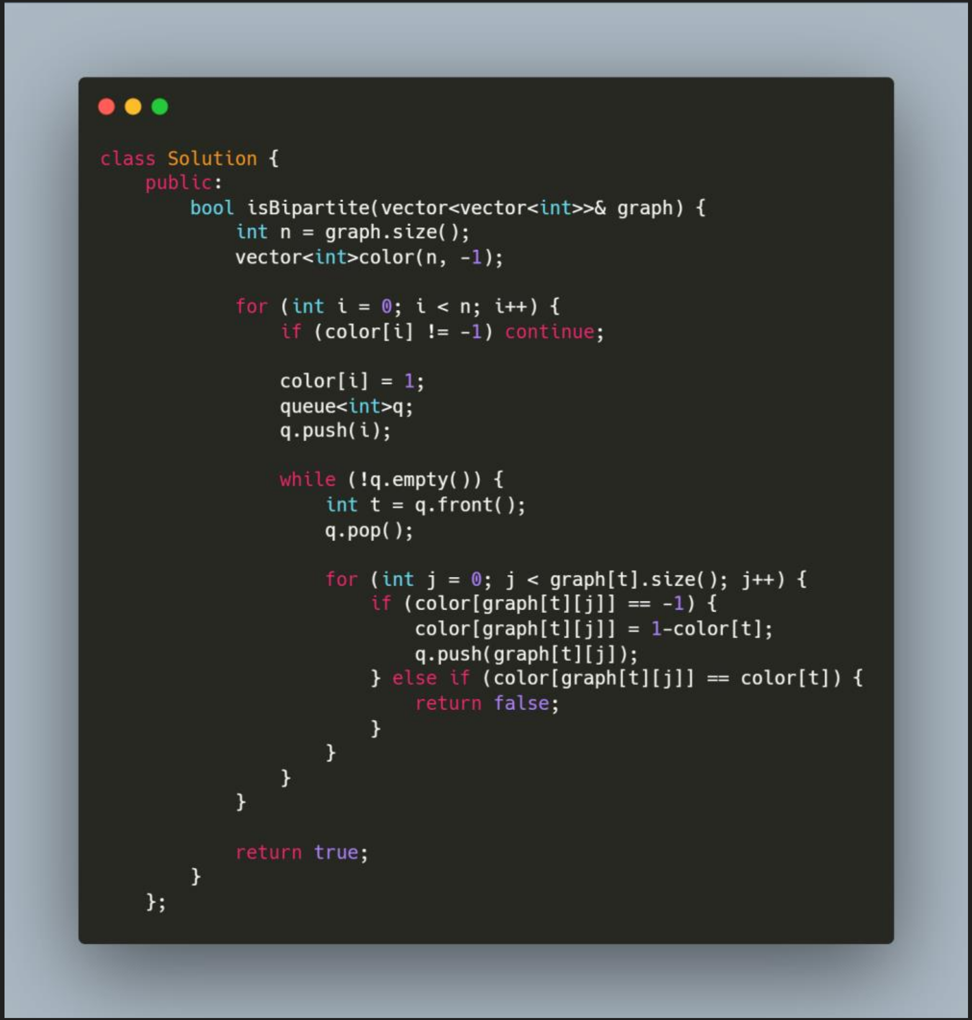


```
class Solution {
    public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {

        if (matrix.empty()) return matrix;
        int rows = matrix.size();
        int cols = matrix[0].size();
        queue<pair<int, int>>pq;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j] == 0) {
                    pq.push({i-1, j}), pq.push({i+1, j}), pq.push({i, j-1}), pq.push({i, j+1});
                }
            }
        }

        vector<vector<bool>>visited(rows, vector<bool>(cols, false));
        int steps = 0;
        while (!pq.empty()) {
            steps++;
            int size = pq.size();
            for (int i = 0; i < size; i++) {
                auto front = pq.front();
                int l = front.first;
                int r = front.second;
                pq.pop();
                if (l >= 0 && r >= 0 && l < rows && r < cols && !visited[l][r] && matrix[l][r] == 1) {
                    visited[l][r] = true;
                    matrix[l][r] = steps;
                    pq.push({l-1, r}), pq.push({l+1, r}), pq.push({l, r-1}), pq.push({l, r+1});
                }
            }
        }

        return matrix;
    }
};
```

GRAPH COLORING

Graph coloring/Bipartition
https://leetcode.com/problems/possible-bipartition/
https://leetcode.com/problems/is-graph-bipartite/

Problems asks to check if its possible to divide the graph nodes into 2 groups
Apply BFS for same. Below is a sample graph coloring approach.

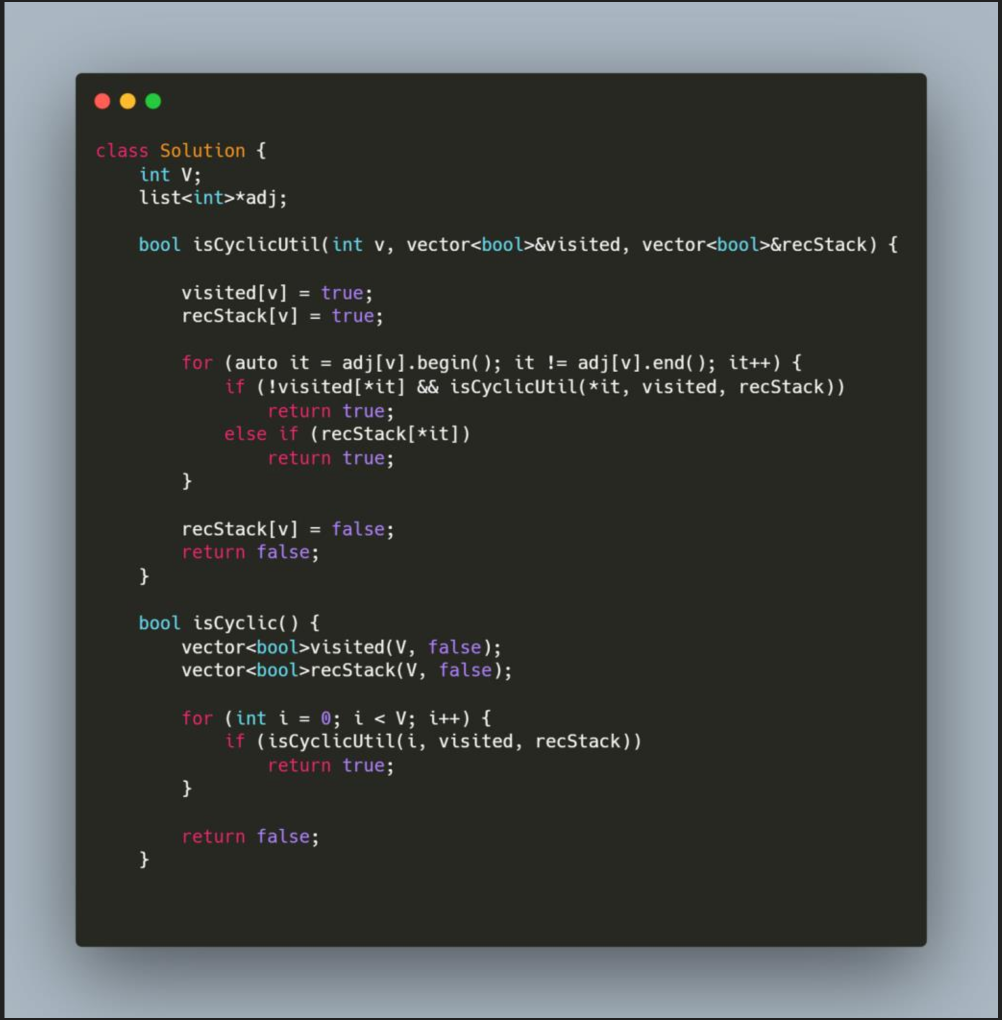


```
class Solution {
    public:
        bool isBipartite(vector<vector<int>>& graph) {
            int n = graph.size();
            vector<int>color(n, -1);

            for (int i = 0; i < n; i++) {
                if (color[i] != -1) continue;

                color[i] = 1;
                queue<int>q;
                q.push(i);

                while (!q.empty()) {
                    int t = q.front();
                    q.pop();

                    for (int j = 0; j < graph[t].size(); j++) {
                        if (color[graph[t][j]] == -1) {
                            color[graph[t][j]] = 1-color[t];
                            q.push(graph[t][j]);
                        } else if (color[graph[t][j]] == color[t]) {
                            return false;
                        }
                    }
                }
            }

            return true;
        }
    };
```

TOPO SORTING

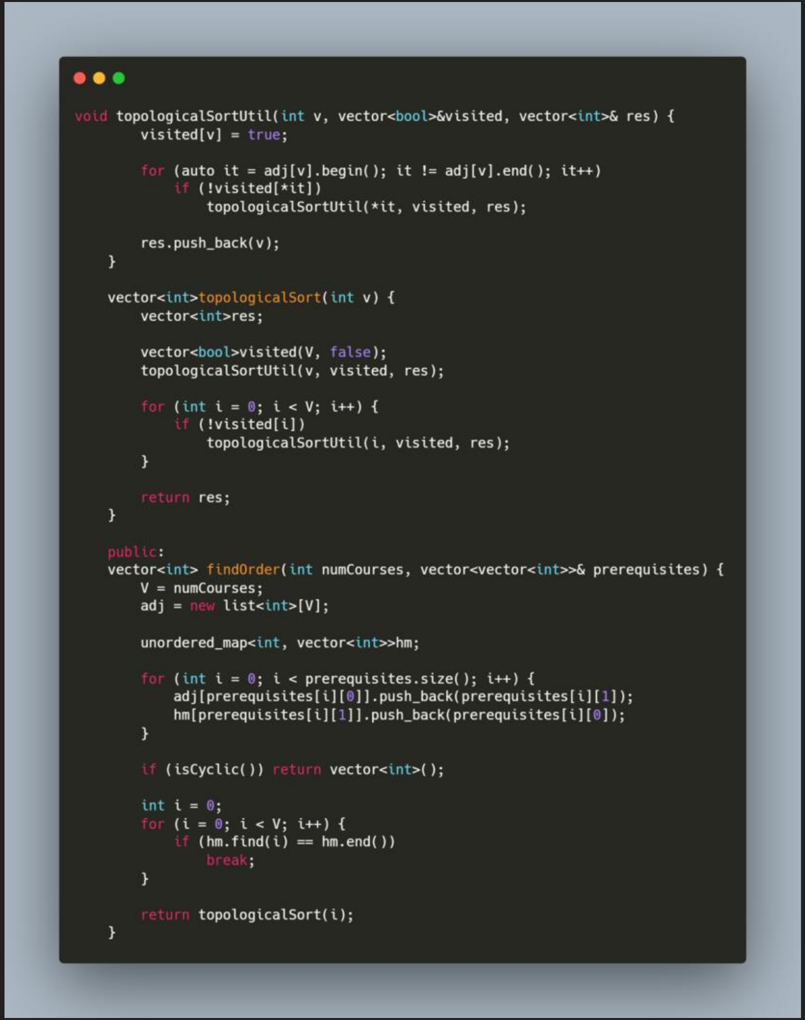
Topological Sort:
Check if its directed acyclic graph and we have to arrange the elements in an order in which we need to select the most independent node at first.
Number of in-node 0

https://leetcode.com/problems/course-schedule/
https://leetcode.com/problems/course-schedule-ii/



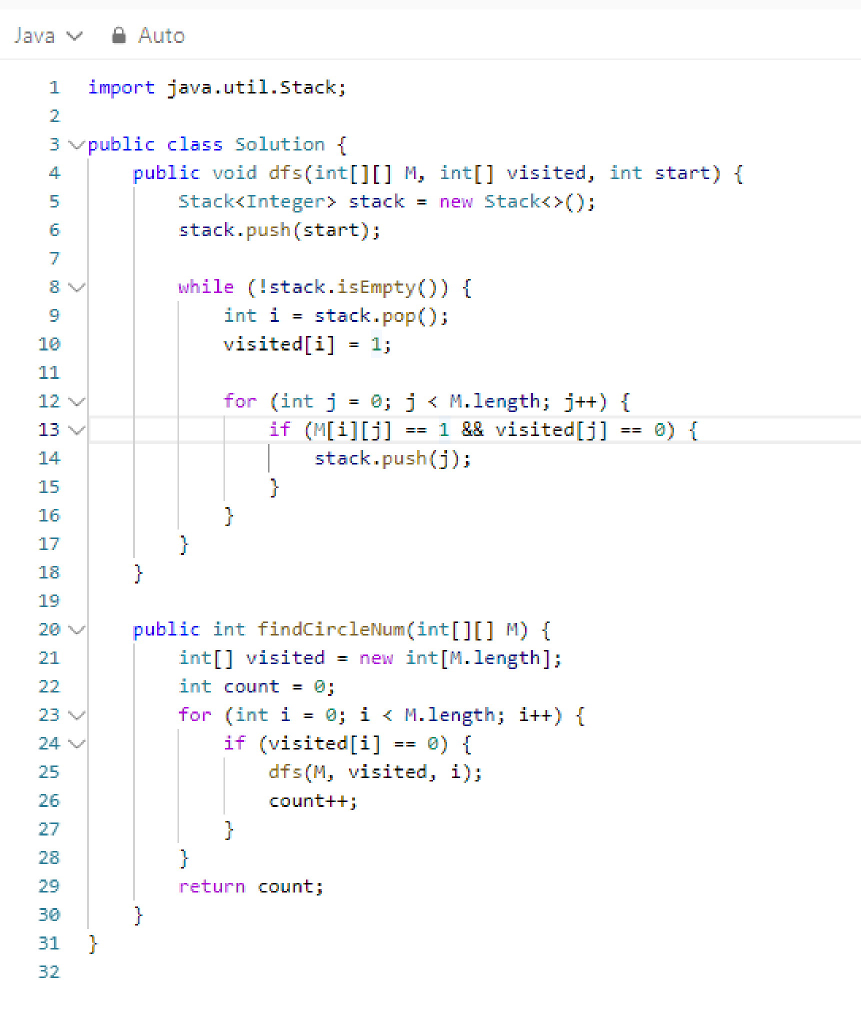
```
class Solution {
    int V;
    list<int>*adj;

    bool isCyclicUtil(int v, vector<bool>&visited, vector<bool>&recStack) {

        visited[v] = true;
        recStack[v] = true;

        for (auto it = adj[v].begin(); it != adj[v].end(); it++) {
            if (!visited[*it] && isCyclicUtil(*it, visited, recStack))
                return true;
            else if (recStack[*it])
                return true;
        }

        recStack[v] = false;
        return false;
    }

    bool isCyclic() {
        vector<bool>visited(V, false);
        vector<bool>recStack(V, false);

        for (int i = 0; i < V; i++) {
            if (isCyclicUtil(i, visited, recStack))
                return true;
        }

        return false;
    }
```

TOPO SORTING PART 2

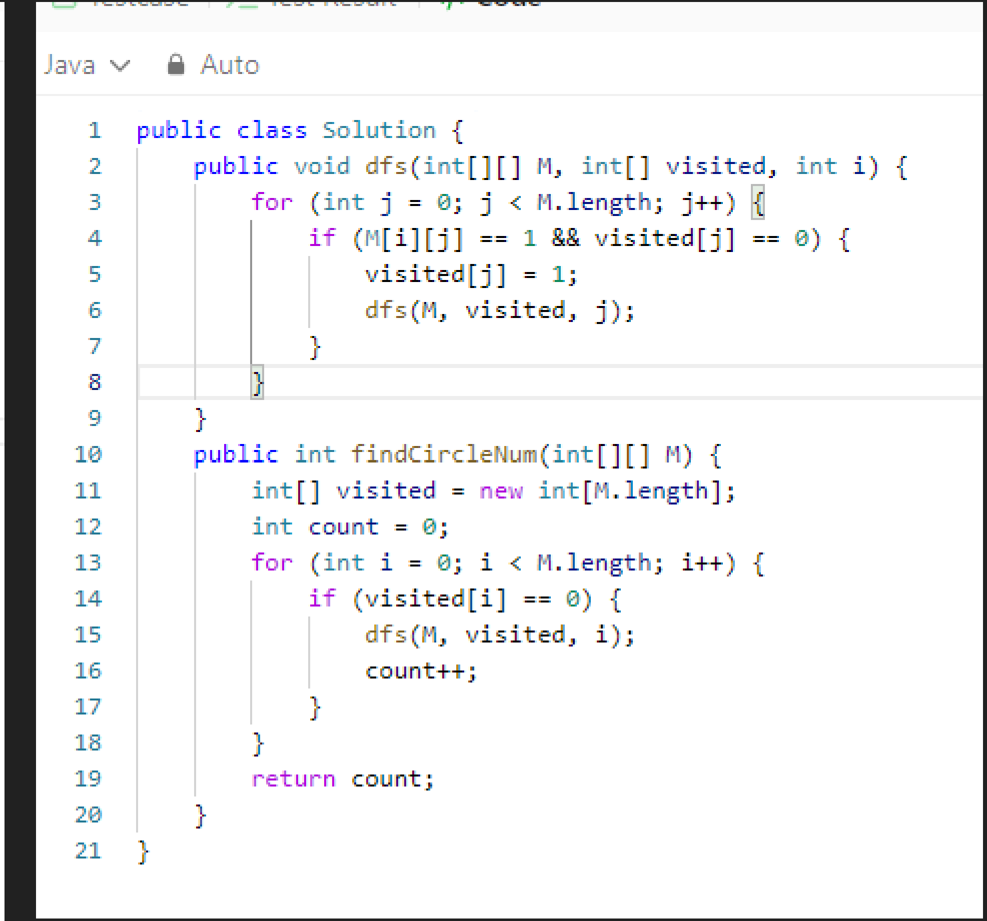


```
void topologicalSortUtil(int v, vector<bool>&visited, vector<int>& res) {
        visited[v] = true;

        for (auto it = adj[v].begin(); it != adj[v].end(); it++)
            if (!visited[*it])
                topologicalSortUtil(*it, visited, res);

        res.push_back(v);
    }

    vector<int>topologicalSort(int v) {
        vector<int>res;

        vector<bool>visited(V, false);
        topologicalSortUtil(v, visited, res);

        for (int i = 0; i < V; i++) {
            if (!visited[i])
                topologicalSortUtil(i, visited, res);
        }

        return res;
    }

    public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        V = numCourses;
        adj = new list<int>[V];

        unordered_map<int, vector<int>>hm;

        for (int i = 0; i < prerequisites.size(); i++) {
            adj[prerequisites[i][0]].push_back(prerequisites[i][1]);
            hm[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }

        if (isCyclic()) return vector<int>();

        int i = 0;
        for (i = 0; i < V; i++) {
            if (hm.find(i) == hm.end())
                break;
        }

        return topologicalSort(i);
    }
```

DFS - Iterative and recursive solutions



```java
import java.util.Stack;

public class Solution {
    public void dfs(int[][] M, int[] visited, int start) {
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int i = stack.pop();
            visited[i] = 1;

            for (int j = 0; j < M.length; j++) {
                if (M[i][j] == 1 && visited[j] == 0) {
                    stack.push(j);
                }
            }
        }
    }

    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```



```java
public class Solution {
    public void dfs(int[][] M, int[] visited, int i) {
        for (int j = 0; j < M.length; j++) {
            if (M[i][j] == 1 && visited[j] == 0) {
                visited[j] = 1;
                dfs(M, visited, j);
            }
        }
    }
    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

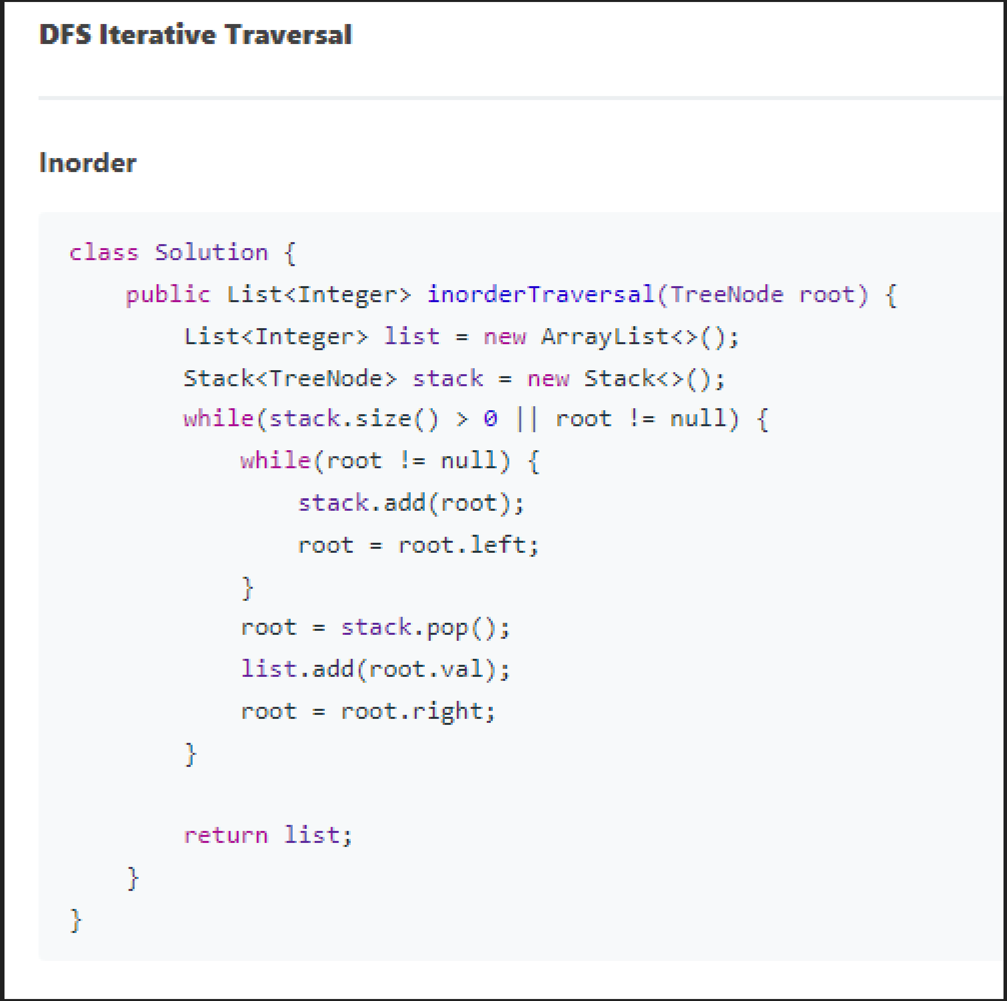
LEETCODE ARTICLES

https://leetcode.com/discuss/general-discussion/655708/Graph-For-Beginners-Problems-or-Pattern-or-Sample-Solutions

https://leetcode.com/discuss/study-guide/2360573/Become-Master-In-Graph

Trees - traversals

DFS iterative

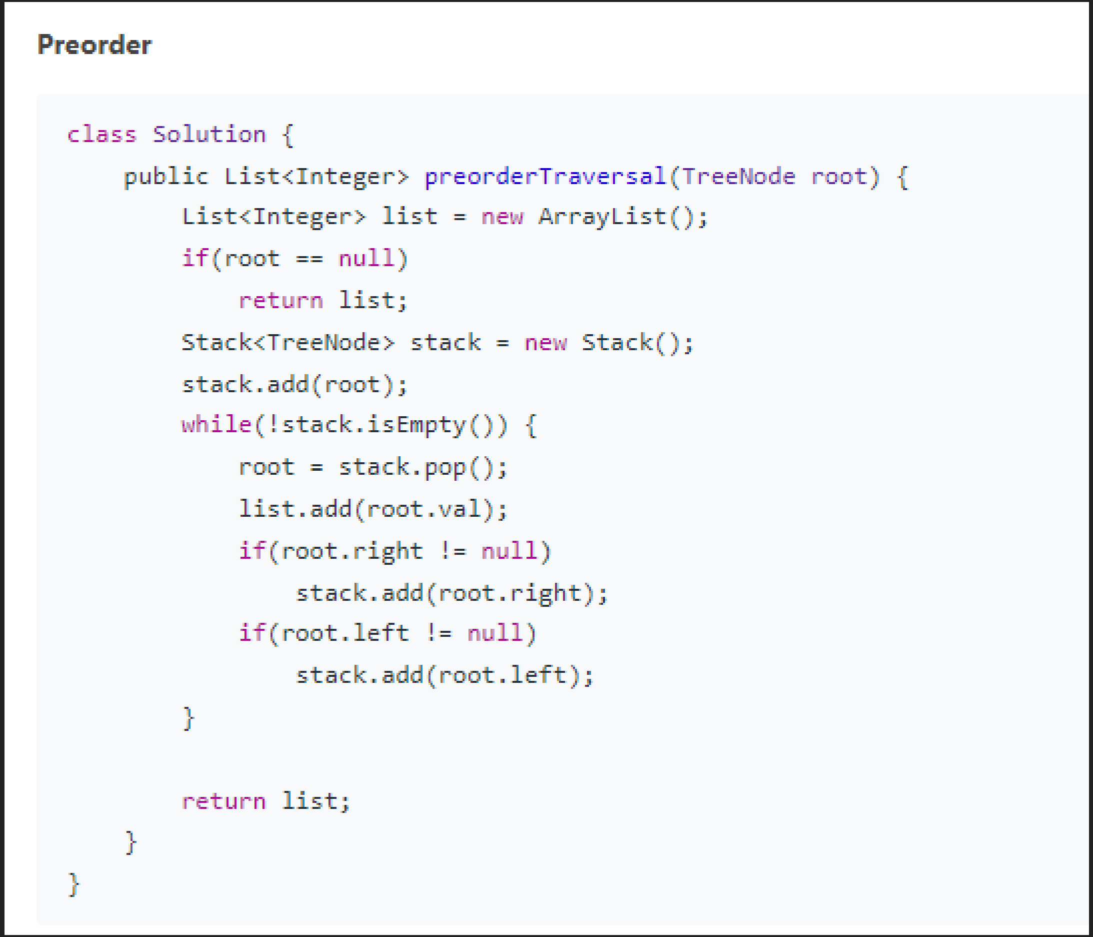


DFS Iterative Traversal

Inorder

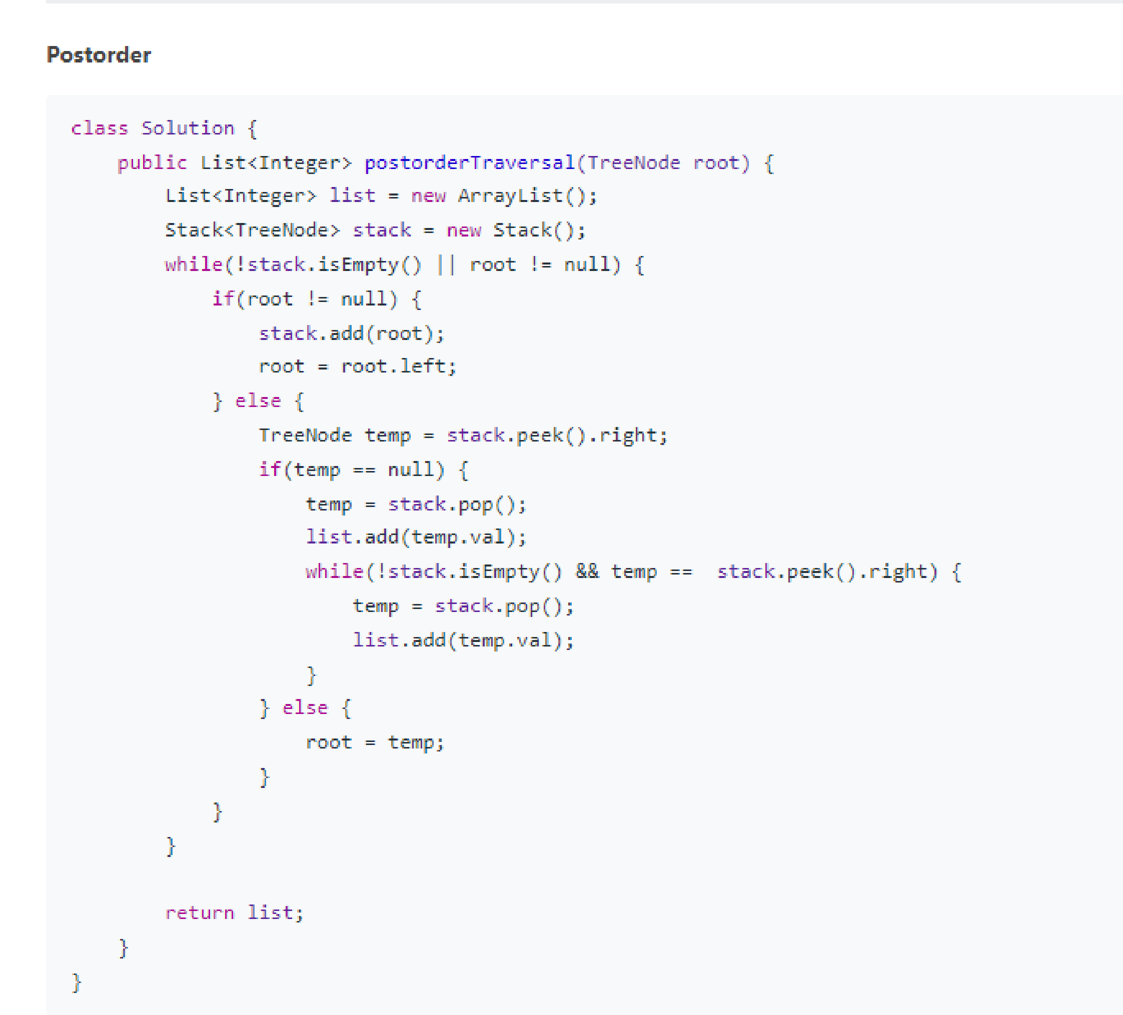
```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while(stack.size() > 0 || root != null) {
            while(root != null) {
                stack.add(root);
                root = root.left;
            }
            root = stack.pop();
            list.add(root.val);
            root = root.right;
        }

        return list;
    }
}
```

Preorder traversal



Preorder

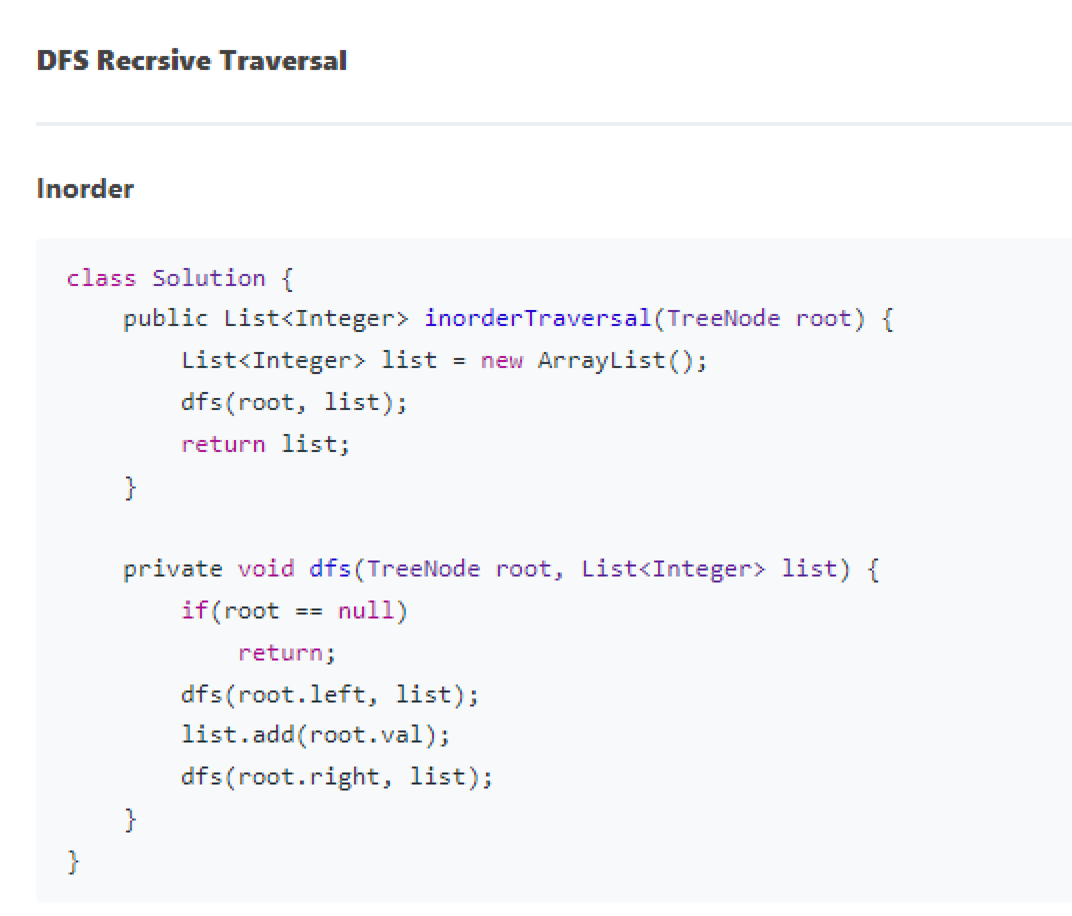
```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        if(root == null)
            return list;
        Stack<TreeNode> stack = new Stack();
        stack.add(root);
        while(!stack.isEmpty()) {
            root = stack.pop();
            list.add(root.val);
            if(root.right != null)
                stack.add(root.right);
            if(root.left != null)
                stack.add(root.left);
        }

        return list;
    }
}
```



Postorder

```
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        Stack<TreeNode> stack = new Stack();
        while(!stack.isEmpty() || root != null) {
            if(root != null) {
                stack.add(root);
                root = root.left;
            } else {
                TreeNode temp = stack.peek().right;
                if(temp == null) {
                    temp = stack.pop();
                    list.add(temp.val);
                    while(!stack.isEmpty() && temp ==  stack.peek().right) {
                        temp = stack.pop();
                        list.add(temp.val);
                    }
                } else {
                    root = temp;
                }
            }
        }

        return list;
    }
}
```

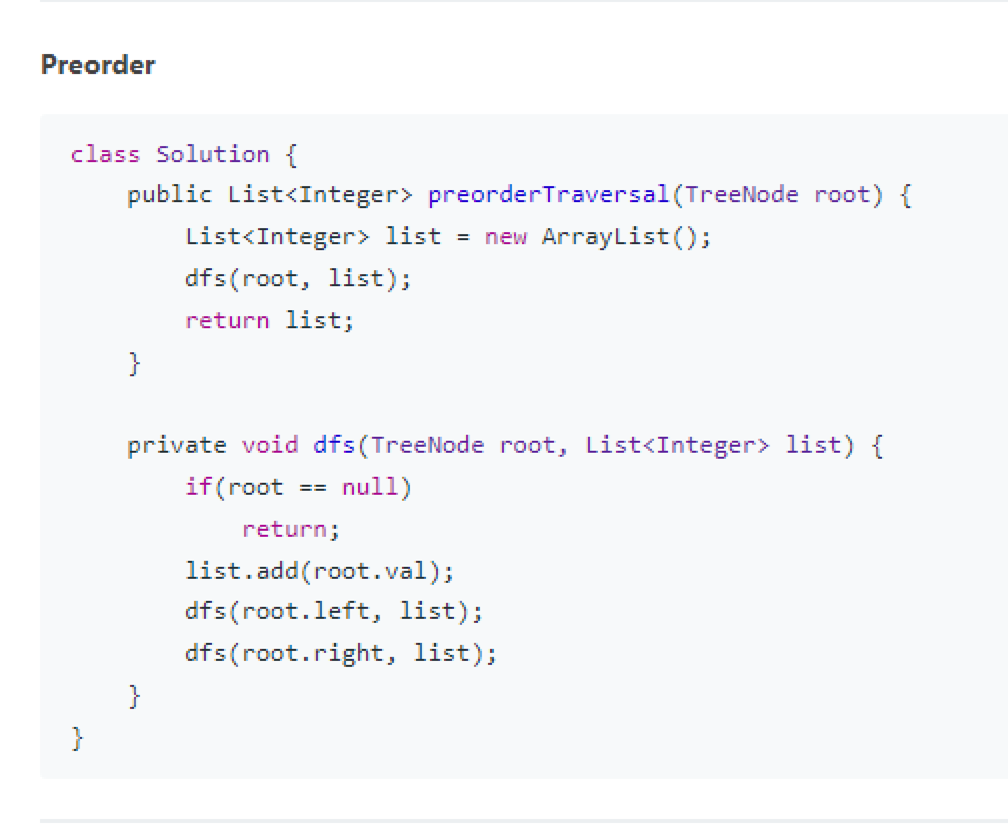
DFS recursive



DFS Recrsive Traversal

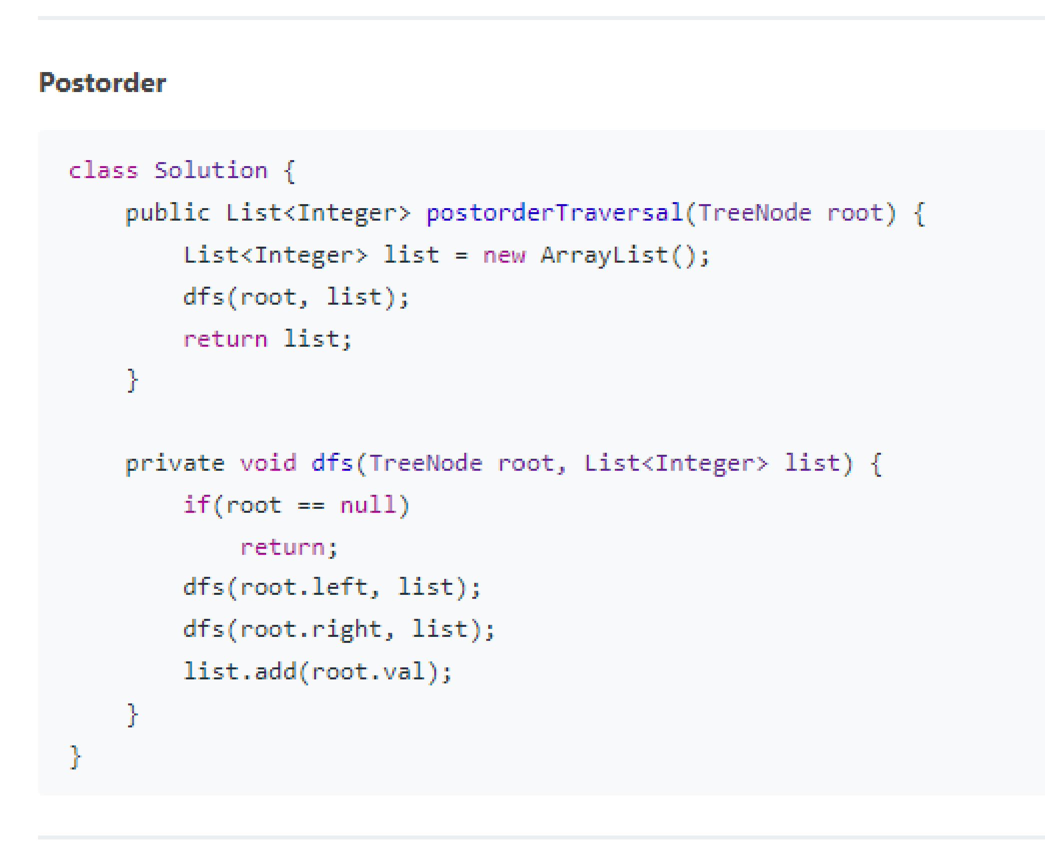
Inorder

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        dfs(root, list);
        return list;
    }

    private void dfs(TreeNode root, List<Integer> list) {
        if(root == null)
            return;
        dfs(root.left, list);
        list.add(root.val);
        dfs(root.right, list);
    }
}
```



Preorder

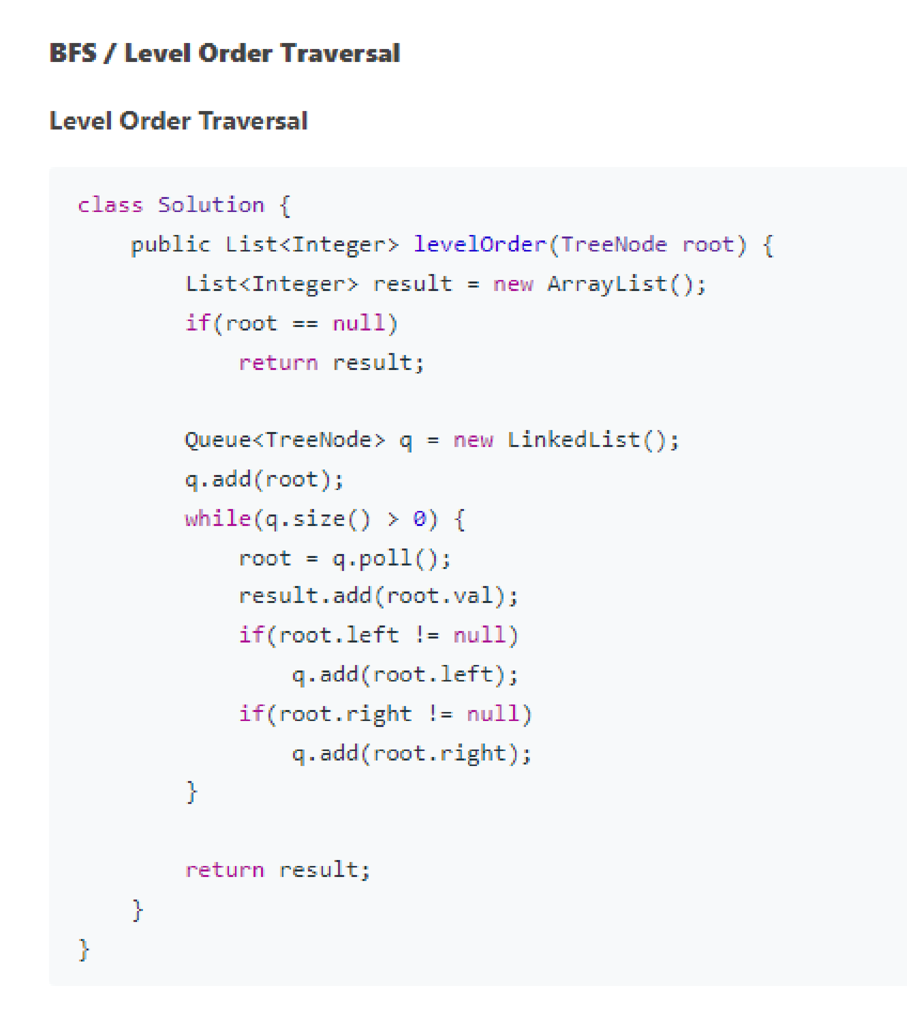
```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        dfs(root, list);
        return list;
    }

    private void dfs(TreeNode root, List<Integer> list) {
        if(root == null)
            return;
        list.add(root.val);
        dfs(root.left, list);
        dfs(root.right, list);
    }
}
```



Postorder

```
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList();
        dfs(root, list);
        return list;
    }

    private void dfs(TreeNode root, List<Integer> list) {
        if(root == null)
            return;
        dfs(root.left, list);
        dfs(root.right, list);
        list.add(root.val);
    }
}
```

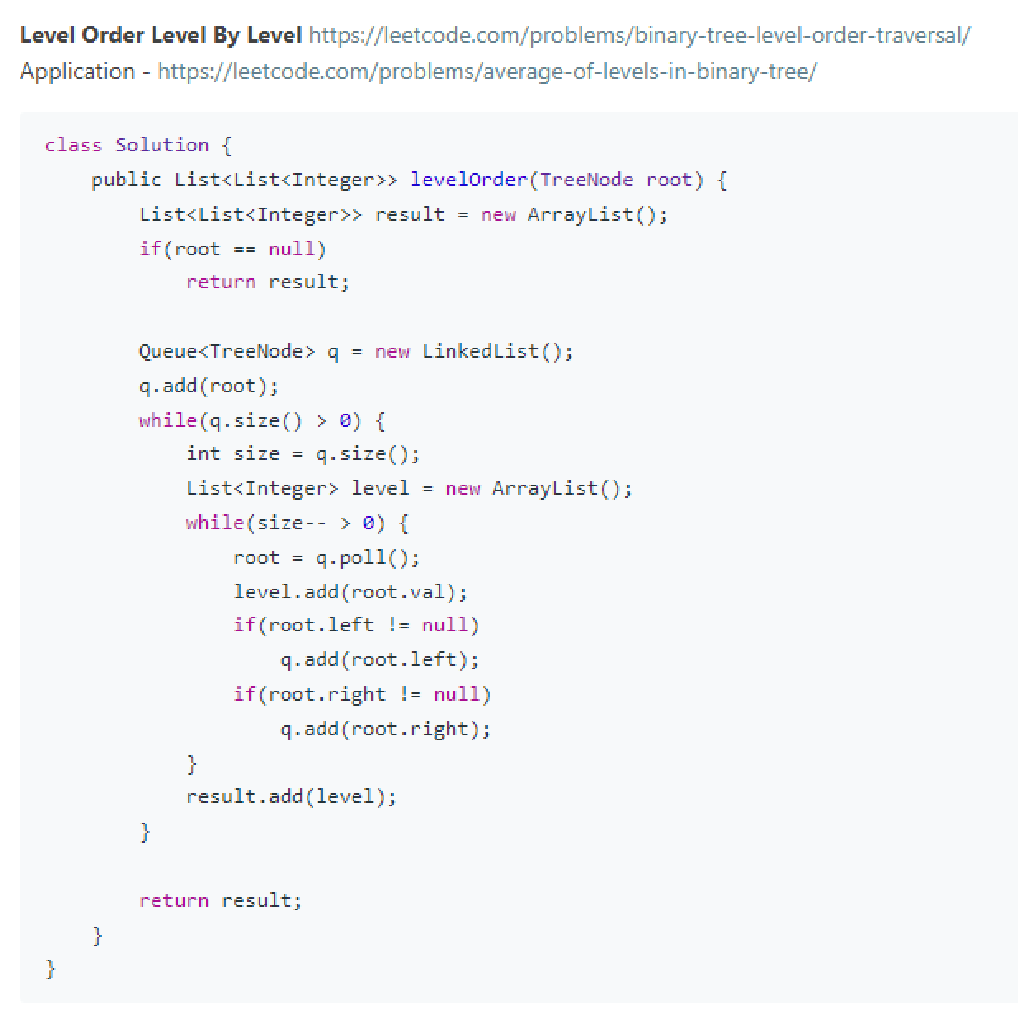
Other traversals



BFS / Level Order Traversal

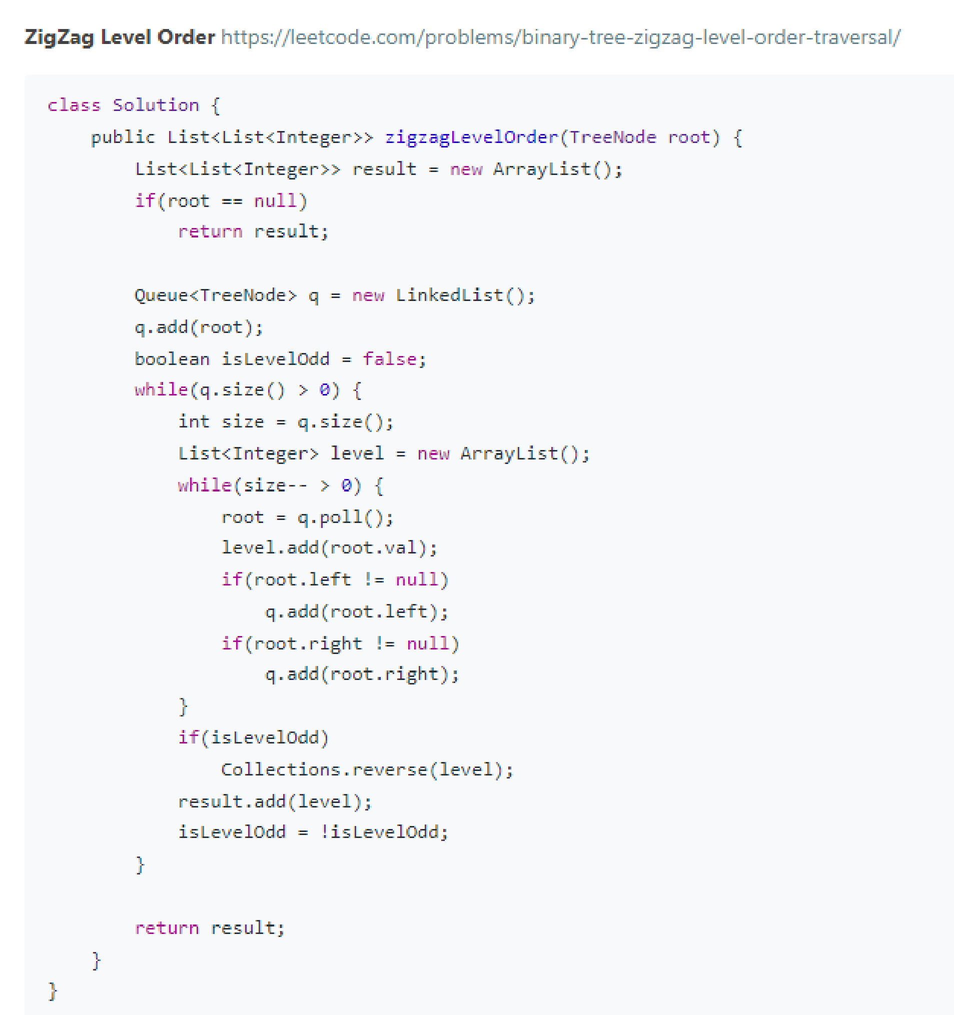
Level Order Traversal

```
class Solution {
    public List<Integer> levelOrder(TreeNode root) {
        List<Integer> result = new ArrayList();
        if(root == null)
            return result;

        Queue<TreeNode> q = new LinkedList();
        q.add(root);
        while(q.size() > 0) {
            root = q.poll();
            result.add(root.val);
            if(root.left != null)
                q.add(root.left);
            if(root.right != null)
                q.add(root.right);
        }

        return result;
    }
}
```



Level Order Level By Level https://leetcode.com/problems/binary-tree-level-order-traversal/
Application - https://leetcode.com/problems/average-of-levels-in-binary-tree/

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList();
        if(root == null)
            return result;

        Queue<TreeNode> q = new LinkedList();
        q.add(root);
        while(q.size() > 0) {
            int size = q.size();
            List<Integer> level = new ArrayList();
            while(size-- > 0) {
                root = q.poll();
                level.add(root.val);
                if(root.left != null)
                    q.add(root.left);
                if(root.right != null)
                    q.add(root.right);
            }
            result.add(level);
        }

        return result;
    }
}
```



ZigZag Level Order https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/

```java
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList();
        if(root == null)
            return result;

        Queue<TreeNode> q = new LinkedList();
        q.add(root);
        boolean isLevelOdd = false;
        while(q.size() > 0) {
            int size = q.size();
            List<Integer> level = new ArrayList();
            while(size-- > 0) {
                root = q.poll();
                level.add(root.val);
                if(root.left != null)
                    q.add(root.left);
                if(root.right != null)
                    q.add(root.right);
            }
            if(isLevelOdd)
                Collections.reverse(level);
            result.add(level);
            isLevelOdd = !isLevelOdd;
        }

        return result;
    }
}
```

Leetcode articles

https://leetcode.com/discuss/general-discussion/937307/iterative-recursive-dfs-bfs-tree-traversal-in-pre-post-levelorder-views

https://leetcode.com/discuss/study-guide/1212004/binary-trees-study-guide