Adventures & Explorations: LC

Guideline

- https://www.teamblind.com/post/New-Year-Gift---Curated-List-of-Top-75-LeetCode-Questions-to-Save-Your-Time-OaM1orEU
- Steps:
  - Keep calm and patient
  - Read question carefully and understand requirements fully: 5-10 mins
  - Set out strategy and get confirmation
  - Code main logic and handle edge cases (don’t hope for luck)

- Good advice/watching: https://www.youtube.com/c/NeetCode https://neetcode.io/
- udemy course:

https://www.udemy.com/course/datastructurescncpp/?referralCode=BD2EF8E61A98AB5E011D

Key steps:

- #1: Read Q clearly and understand requirements exactly, ask for clarification as needed, figure which category and which what data structure.
- #2: Discuss and confirm strategy
- #3: Code main logic first, and try to cover edge cases
- #4: don’t hope for luck that it works magically, make sure/ understand every step clearly.
- #5: Correct/Improve as it runs test cases

C++ STL: unordered_map

```
#include <unordered_map>
unordered_map<string, int> umap;
unordered_map<string, int>::iterator itr;
pair<string, int> elem;
umap[“test1”] = 1; umap[“test2”] = 2;
elem = make_pair(“test3”, 199); umap.insert(elem); // add

itr = umap.find(“mykey”); if (itr != umap.end()) printf(“found mykey”); //look up

for (itr =umap.begin(); itr != umap.end(); itr++) { // iterate thru  for (auto kv: umap) kv.first , kv.second
   printf(“key is %s , val is %d \n”, itr->first, itr->second);
umap.erase(“my key”); umap.erase(umap.begin());
Umap.count(“my key”); return zero if not found;
umap.size();  // return number of elements in the map
Umap.empty() ;//tell if there is anything
```

C++ STL unordered_set

```
#include <unordered_set>
unordered_set<string> set1;
unordered_set<string>::iterator itr1;
unordered_set<int> set2;
set1.insert("hello"); set1.insert("world");  // add
if (set1.find("myname") != set1.end()) { // found it … } // look up
for (auto itr = set1.begin(); itr != set1.end(); itr++) {  //iterate
      printf("%s", *itr);
}
set2.insert(10); set2.insert(20);  set2.erase(20); set2.erase(set2.find(10));
set2.count(key) => 0 or 1;   set2.size() how many elements?
if (set1.empty()) { // nothing inside …}
```

C++ STL Stack and Queue

```
#include <stack>
stack<int> st;
st.push(10); st.push(20); st.push(30);
while (!st.empty()) { printf("%d", st.top()); st.pop();}
st.size();

#include <queue>
queue<int> q;
q.push(100); q.push(200); q.push(300);
while (!q.empty()) { printf("%d, ", q.front()); q.pop();}
q.size();
```

C++ STL Vector: used as ArrayList in Java

```
#include <vector>
vector<int> g1;
for (int i = 1; i <= 10; i++)  g1.push_back(i * 10);
cout << "\n Reference operator [g] : g1[2] = " << g1[2];
cout << "\n Using at : g1.at(4) = " << g1.at(4);
cout << "\nfront() : g1.front() = " << g1.front();
cout << "\nback() : g1.back() = " << g1.back();
 int* pos = g1.data(); // pointer to the first element

 g1.push_back(15);  g1.pop_back();
g1.insert(g1.begin(), 100); g1.insert(g1.begin()+3, 300);
g1.erase(g1.begin() +3) ; //erase
int index = find(g1.begin(), g1.end(), key) – g1.begin();
#include <bits/stdc++.h>
vector<int> v1 { 1, 20, 3, 40, 5, 60};
sort(v1.begin(), v1.end(), less<int>()); // less is default { 1,3,5,20, 40, 60}
```

C++ STL string class

```
#include <string>
std::string s = "Hello";
std::string greet = s + " World"; //concatenation easy!
str.push_back('s');    str.length();  str.at(i)  is same as  str[i]
to_string(123); // converte integer to string
I = stoi("123"); // return interger 123;
str.find(subStr,0) == 0    => str is started with subStr
str.push_back('c');
string::append (size_type num, char c)

str.substr(pos, len);
const char *cp = str.c_str();
```

C++ STD priority_queue: min heap, adjust on the fly based on new value pq.push(), pq.pop()

- #include <queue>
- priority_queue<int, vector<int>, Cmp> pq;
  // without Cmp, big num# first pop(), Cmp=less<int>, called Max heap
- Comparator:

```
class Cmp {
 public:
  bool operator()(int i1, int i2) {
     return i1 > i2 ; //pq.top(): will be smallest number, min heap
}
```

- priority_queue<int, vector<int>, greater<int>> min_hep;

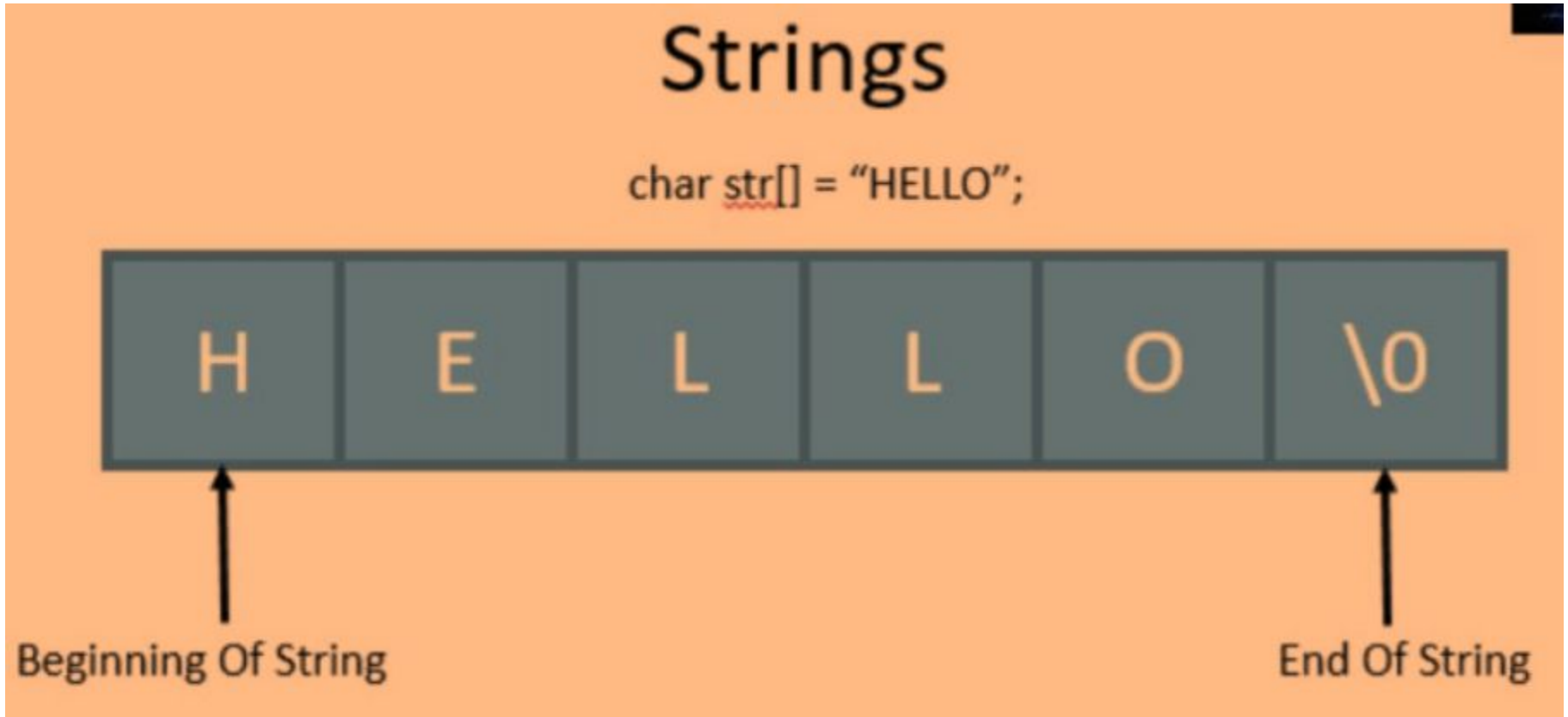
Binary search in vector

- sort: you can use binary search only on a *sorted* data, so you must guarantee that the data is sorted, before searching.
- lower_bound: this function returns an *iterator* to the first element that is **greater than or equal to** value.
- upper_bound: this function returns an *iterator* to the first element that is **greater than** value.
- binary_search:: this functions returns a *boolean*, whether the value is found or not (exactly as your program).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iostream>

typedef std::vector<int>::iterator iter;

int main() {

    std::vector<int> vec = {10, 20, 30, 30, 20, 10, 10, 20};

    // sort the data
    // the data will be:
    // 10, 10, 10, 20, 20, 20, 30, 30
    std::sort(vec.begin(), vec.end());

    // index of the first element, greater than or equal to 20
    iter low = std::lower_bound(vec.begin(), vec.end(), 20);

    // index of the first element, greater than 20
    iter high = std::upper_bound(vec.begin(), vec.end(), 20);

    std::cout << "index of first element, greater than or equal to 20 is: " << (low - vec.begin()) << '\n';

    std::cout << "index of first element, greater than to 20 is: " << (high - vec.begin()) << '\n';

    // classic binary search
    // check whether a givin value exists in the array or not
    if (std::binary_search(vec.begin(), vec.end(), 99)) {
        std::cout << "Found\n";
    } else {
        std::cout << "Not found\n";
    }

}
```

Return 3

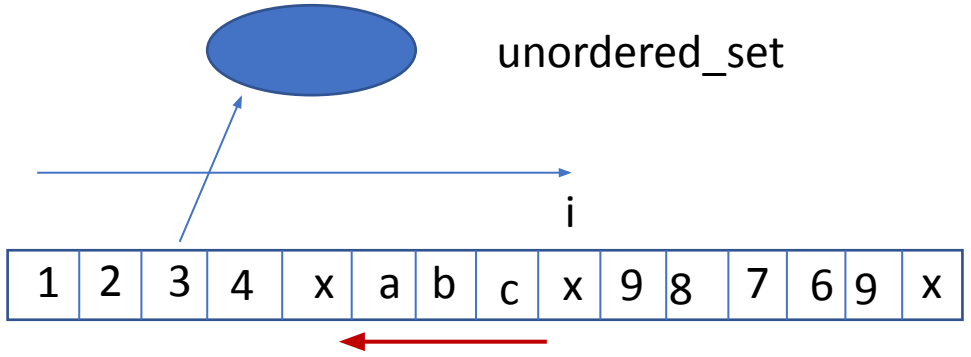
Return 6


Strings
char str[] = “HELLO”;
H
E
L
L
O
\0
Beginning Of String
End Of String

LC String#1: 3. Longest Substring Without Repeating Characters

```cpp
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int res = 0;
        int c = 0;
        if (s.length() < 2) return s.length();
        unordered_set<char> seen;
        for (int i = 0; i < s.length(); i++) {
            if (seen.find(s.at(i)) == seen.end()) {
                c++;
                res = max(c, res);
                seen.insert(s.at(i));
            } else {
                seen.clear();
                // search back for the last occurence of this character
                seen.insert(s.at(i));
                c = 1;
                for (int j = i-1; j >= 0; j--) {
                    if (s.at(j) != s.at(i)) {
                        seen.insert(s.at(j));
                        c++;
                    } else {
                        break;
                    }
                }
            }
        }
        return res;
    }
};
```

- Scan thru string, push each one into "set" named as "seen" if not seen, c++
- If it is seen before, reset count "c", and clear "seen", search back until current char is hit(where is l is adjusted to) add all back into "seen"



LC String#2: two pionters/ Caterpillar algorithm

424. Longest Repeating Character with K Replacement

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        int L = s.length();
        int res = 0;
        int l = 0, r = 0;
        int counters[26] = { 0 };
        bool adv_right = true;
        while (l <= r && r < L) {
            if (adv_right)
                counters[s.at(r)-'A']++;
            else
                counters[s.at(l-1)-'A']--;

            //find max freq of current window
            int max_freq = 0;
            for (int i = 0; i < 26; i++) {
                max_freq = max(max_freq, counters[i]);
            }
            int rc = r-l+1 - max_freq;
            if (rc <= k) {
                res = max(r-l+1, res);
                printf("\n res=%d, l=%d, r=%d", res, l,r);
                adv_right = true;
                r++;
            } else {
                printf("\n advance left l=%d", l+1);
                l++;
                adv_right = false;
            }
        }
        return res;
    }
};
```

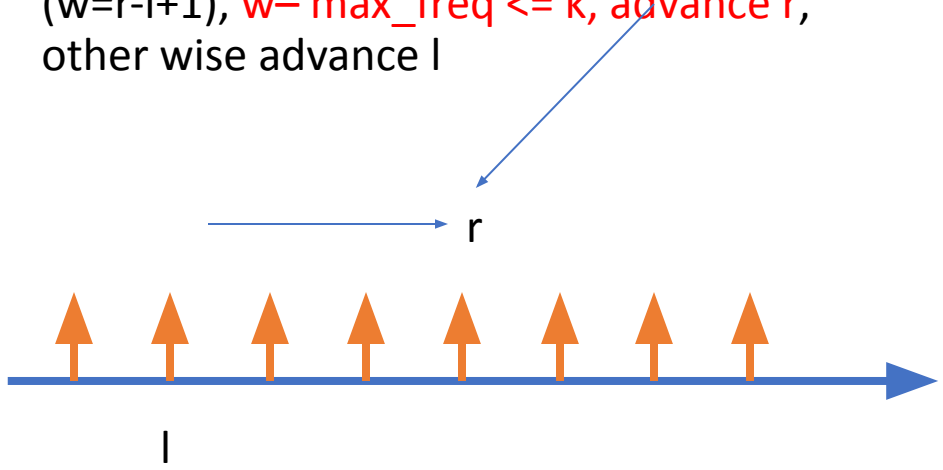
You are given a string `s` and an integer `k`. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most `k` times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations.*

Example 1:

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
```

- Use Sliding windows: l=0, r=0->L
- Use a counters[26] array to count each inside window's letters frequency when l/r is advanced
- Inside a sliding window, find the max_freq letter and its counter, window width (w=r-l+1), w– max_freq <= k, advance r, other wise advance l



LC string#3: two pionters/ Caterpillar algorithm

76. Minimum Window Substring: from S including T

Given two strings `s` and `t` of lengths `m` and `n` respectively, return *the* ***minimum window substring*** *of* `s` *such that every character in* `t` (***including duplicates***) *is included in the window. If there is no such substring, return the empty string* `""`.

The testcases will be generated such that the answer is **unique**.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
```

```cpp
class Solution {
public:
    string minWindow(string s, string t) {
        int l = 0, r = 0, res= INT_MAX, r_ = 0, l_ = 0;
        bool adv_r = true;
        int LEN = 'z' - 'A' + 1;
        int t_counters['z' - 'A' + 1] = { 0 };
        int s_counters['z' - 'A' + 1] = { 0 };

        for (int i = 0; i < t.length(); i++) t_counters[t.at(i)-'A']++;
        while (l <= r && r < s.length()) {
            if (adv_r)
                s_counters[s.at(r)-'A']++;
            else
                s_counters[s.at(l-1)-'A']--;

            //check if current window has all string t
            bool cover_t = true;
            for (int i = 0; i < LEN; i++) {
                if (t_counters[i] != 0 && s_counters[i] < t_counters[i]) {
                    cover_t = false;
                    break;
                }
            }
            if (cover_t) {
                if (r-l+1 < res) {
                    r_ = r; l_ = l;
                    res = r - l + 1;
                }
                adv_r = false;
                l++;
            } else {
                adv_r = true;
                r++;
            }
        }

        return res == INT_MAX? "":s.substr(l_, res);
    }
};
```

- Sliding window: expand R to cover T, write down W
- Advance L until not cover T, write down smaller W,
- Go back to first step until reach the end of string.

LC string#4: 242. Valid Anagram: s = "anagram", t = "nagaram“ => true

```cpp
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) return false;
        int s_c['z'-'a'+1] = {0};
        int t_c['z'-'a'+1] = {0};
        for (int i = 0; i < s.length(); i++) {
            s_c[s.at(i)-'a']++;
            t_c[t.at(i)-'a']++;
        }
        return memcmp(s_c, t_c, sizeof(int)*('z'-'a'+1)) == 0;
    }
};
```

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

LC string#5: 49. Group Anagrams:

Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> res;
        vector<vector<int>> counters;
        vector<bool> visited;
        for (int i = 0; i < strs.size(); i++) {
            string s = strs[i];
            vector<int> c(26,0);
            for (int j = 0; j < s.length(); j++)
                c[s.at(j) - 'a']++;
            counters.push_back(c);
            visited.push_back(false);
        }

        for (int i = 0; i < strs.size(); i++) {
            if (!visited[i]) {
                vector<string> ans;
                ans.push_back(strs[i]);
                visited[i] = true;
                for (int j = i+1; j < strs.size(); j++) {
                    if (counters[i] == counters[j]) {
                        ans.push_back(strs[j]);
                        visited[j] = true;
                    }
                }
                res.push_back(ans);
            }
        }
        return res;
    }
};
```

- Character counters: for each string vector<int> c(26,0)
- Visited[] Boolean to speed up
- Scan thru strs and counters[] vector compare: counters[i] == counters[j]

LC string#6: 20. Valid Parentheses:

Input: s = "()[]{}"
Output: true

Input: s = "([)]"
Output: false

```cpp
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;

        for (int i = 0; i < s.length(); i++) {
            if (s.at(i) == '(' || s.at(i) == '[' || s.at(i) == '{')
                st.push(s.at(i));
            else {
                if (st.empty()) return false;
                char c = st.top(); st.pop();
                if ((c == '{' && s.at(i) == '}') ||
                    (c == '[' && s.at(i) == ']') ||
                    (c == '(' && s.at(i) == ')')
                   ) {
                    continue;
                } else
                    return false;
            }
        }
        return st.empty();
    }
};
```

- Use stack: opening one, push, closing one pop and compare.

LC string#7: 125. Valid Palindrome:

Input: s = "A man, a plan, a canal: Panama"

Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

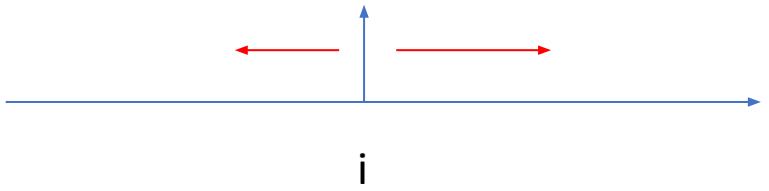
```
class Solution {
public:
    bool isAlphaNum(char c) {
        return ((c >= 'A' && c <= 'Z') ||
           (c >= 'a' && c <= 'z') ||
           (c >= '0' && c <= '9'));
    }
    bool equalIgnoreCase(char a, char b) {
        // converte to uppercase to compare
        if (a >= 'a' && a <= 'z') {
            a = 'A' + (a-'a');
        }
        if (b >= 'a' && b <= 'z') {
            b = 'A' + (b-'a');
        }
        return a == b;
    }
    bool isPalindrome(string s) {
        int l = 0;
        int r = s.length() -1;

        while (l < r) {
            while (l < s.length() && !isAlphaNum(s.at(l))) l++;
            while (r >= 0 && !isAlphaNum(s.at(r))) r--;
            if (l >= s.length() || r < 0 ) break;
            if (!equalIgnoreCase(s.at(l), s.at(r))) return false;
            l++;
            r--;
        }
        return true;
    }
};
```

LC string#8: 5. Longest Palindromic substring:

```
class Solution {
public:
    int expandHelper(string s, int l, int r) {
        while (l >= 0 && r <= s.length()-1 && s.at(l) == s.at(r) ) {
            l--; r++;
        }
        return r-l-1;
    }
    string longestPalindrome(string s) {
        int len = 1;
        int pos = 0;
        for (int i = 0; i < s.length(); i++) {
            int len_odd = expandHelper(s, i, i);
            int len_even = expandHelper(s, i, i+1);
            int max_len = max(len_odd, len_even);
            if (max_len > len) {
                len = max_len;
                pos = i - (len-1)/2;
            }
        }
        return s.substr(pos, len);
    }
};
```

Expand with I as center “odd”

Expand with i and i+1 as “even”
take longer one , record it along the way
to get len, pos = i-(len-1)/2



LC string#9: 647: Palindromic substring:

```
class Solution {
public:
    void expandHelper(string s, int l, int r, int & result) {
        while ( l >=0 && r < s.length() && s.at(l) == s.at(r)) {
            result++;
            l--; r++;
        }
    }
    int countSubstrings(string s) {
         int res = 0;

        for (int i = 0; i < s.length(); i++) {
            expandHelper(s, i, i, res);
            expandHelper(s, i, i+1, res);
        }

        return res;
    }
};
```

Given a string `s`, return *the number of* ***palindromic substrings*** *in it.*

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

```
Input: s = "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".
```

Expand with I as center “odd”

Expand with i and i+1 as “even”

i

LC string#10: 271: Encode and Decode strings:

```
class Codec {
public:

    // Encodes a list of strings to a single string.
    string encode(vector<string>& strs) {
        string res;
        for (int i = 0; i < strs.size(); i++) {
            string tmp = to_string(strs[i].length())+"@"+ strs[i];
            res.append(tmp);
        }
        return res;
    }

    // Decodes a single string to a list of strings.
    vector<string> decode(string s) {
        string dec_s = s;

        vector<string> res;
        int i = 0;
        while (i < dec_s.length()) {
            int l = 0;
            while (dec_s.at(i+l) != '@') {
                l++;
            }
            string num_s = dec_s.substr(i,l);
            int len = stoi(num_s);
            i += l+1; // skip @
            res.push_back(dec_s.substr(i, len));
            i += len;
        }
        return res;
    }
};
```

Design an algorithm to encode **a list of strings** to **a string**. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

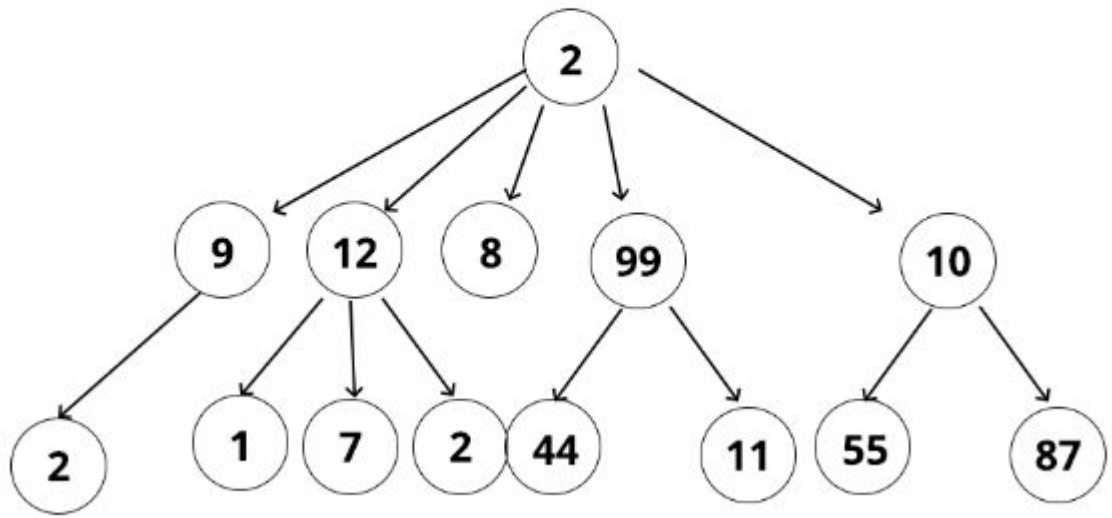
LC string#11: 13. Roman to Integer:

```cpp
class Solution {
public:
    int romanToInt(string s) {
        unordered_map<string, int> map;

        map["I"] = 1; map["V"] = 5; map["IV"] = 4; map["IX"] = 9;
        map["X"] = 10; map["L"] = 50; map["XL"] = 40; map["XC"] = 90;
        map["C"] = 100; map["D"] = 500; map["CD"] = 400; map["CM"] = 900;
        map["M"] = 1000;
        int i = 0;
        int res = 0;
        while (i < s.length()) {
            // check if it one or two characters
            if((i+1) < s.length()) {
                string dStr = s.substr(i,2);
                if (map.find(dStr) != map.end()) {
                    res += map[dStr];
                    i += 2;
                    continue;
                }
            }
            res += map[s.substr(i,1)];
            i++;
        }
        return res;
    }
};
```

12. Integer to Roman

```cpp
class Solution {
public:
    string intToRoman(int num) {
        vector<int> values { 1000, 900, 500, 400, 100,
                             90, 50, 40, 10, 9, 5, 4, 1};
        vector<string> syms { "M", "CM", "D", "CD", "C", "XC",
                              "L", "XL", "X", "IX", "V", "IV", "I"};
        string res = "";
        for (int i = 0; i < values.size() && num > 0; i++) {
            while (num >= values[i]) {
                num -= values[i];
                res.append(syms[i]);
            }
        }
        return res;
    }
};
```

TREES

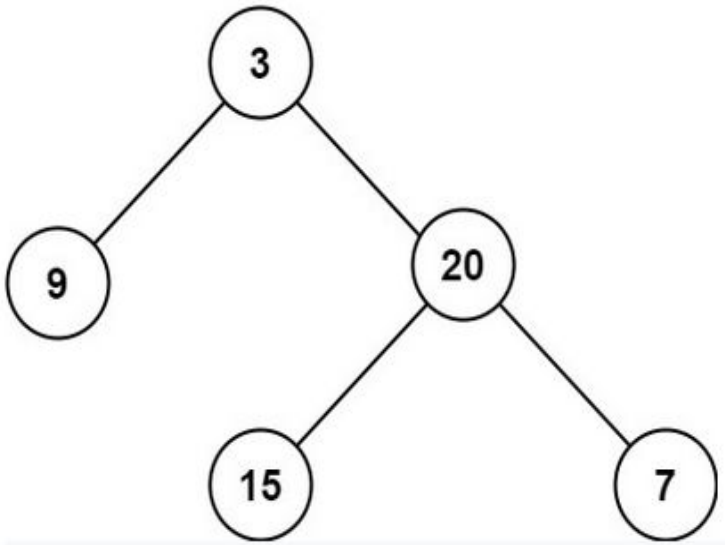


LC tree#1: 104. Maximum Depth of Binary Tree:

```cpp
class Solution {
public:
    int maxDepth(TreeNode* root) {
        int max_level = 0;

        if (!root) return max_level;
        queue<TreeNode*> Q;
        Q.push(root);

        while (!Q.empty()) {
            max_level++;
            int s = Q.size();
            for (int i = 0; i < s; i++) {
                TreeNode *n = Q.front(); Q.pop();
                if (n->left) Q.push(n->left);
                if (n->right) Q.push(n->right);
            }
        }

        return max_level;
    }
};
```

Use Q to do a level traversal and Counting levels

Example 1:



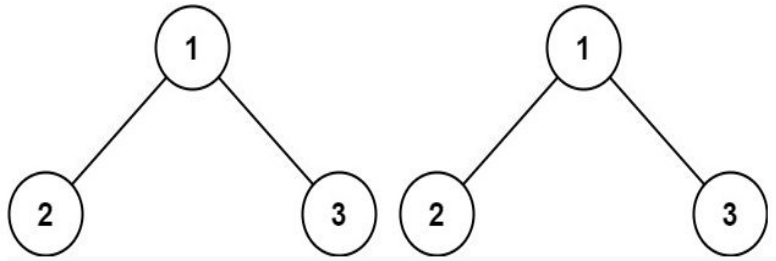

```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

LC tree#2: 100. Same Tree:

```
12  class Solution {
13  public:
14      void dfs(TreeNode *root, vector<int>& v) {
15          if (root == NULL) return;
16          TreeNode *dummy = new TreeNode(INT_MAX);
17
18          stack<TreeNode*> st;
19          st.push(root);
20          while (!st.empty()) {
21              TreeNode *n = st.top(); st.pop();
22              v.push_back(n->val);
23              if (n->val != INT_MAX) {
24                  st.push(n->left? n->left:dummy);
25                  st.push(n->right? n->right:dummy);
26              }
27          }
28          delete(dummy);
29      }
30      bool isSameTree(TreeNode* p, TreeNode* q) {
31          vector<int> p_vals, q_vals;
32          dfs(p, p_vals);
33          dfs(q, q_vals);
34          return p_vals == q_vals;
35      }
36  };
```

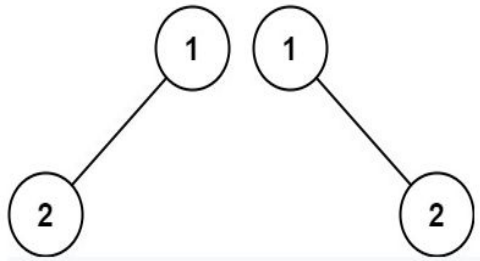
Do a DFS using stack for each tree, for absent node, Add a dummy node with INT_MAX as flag, which should do nothing except add INT_MAX into vector after it is popped out from stack

Example 1:




```
Input: p = [1,2,3], q = [1,2,3]
Output: true
```

Example 2:




```
Input: p = [1,2], q = [1,null,2]
Output: false
```

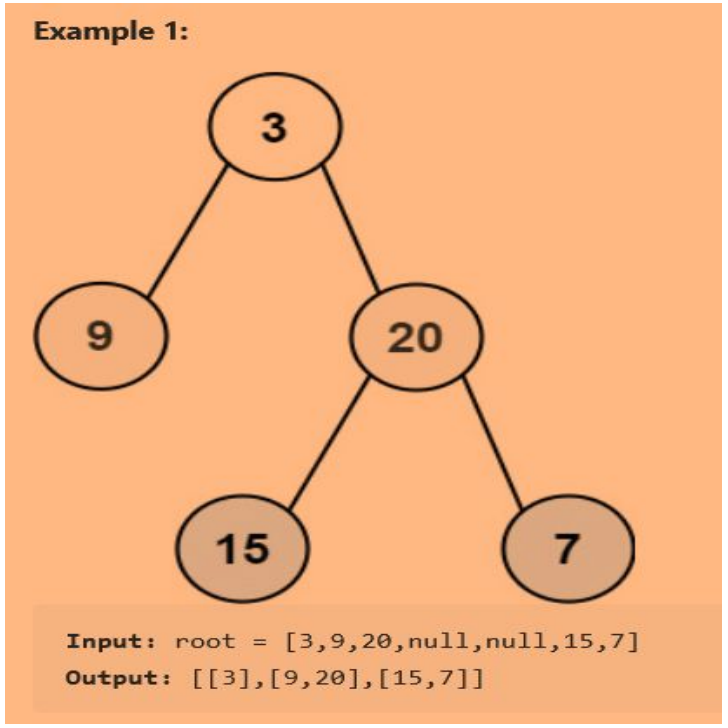
LC tree#3: 226. Invert Binary Tree:

```
12 ▾ class Solution {
13   public:
14 ▾     TreeNode* invertTree(TreeNode* root) {
15           if (root == NULL) return root;
16           TreeNode *t = root->left;
17           root->left = root->right;
18           root->right = t;
19           invertTree(root->left);
20           invertTree(root->right);
21
22           return root;
23       }
24   };
```

Exchange left with right node
Do recursion for left and right.

LC tree#4.1: 102. Binary Tree Level Order Traversal:

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if (root == NULL) return res;
        queue<TreeNode*> Q;
        Q.push(root);

        while (!Q.empty()) {
            int N = Q.size();
            vector<int> level;
            for (int i = 0; i < N; i++) {
                TreeNode *n = Q.front(); Q.pop();
                level.push_back(n->val);
                if (n->left) Q.push(n->left);
                if (n->right) Q.push(n->right);
            }
            res.push_back(level);
        }

        return res;
    }
};
```



Using Q to do level traversal
Check Q size in the beginning of loop,
which is number of nodes in that level.

LC string#4.2: 144. Binary Tree preOrder Traversal:

```
12  class Solution {
13  public:
14      vector<int> preorderTraversal(TreeNode* root) {
15          vector<int> res;
16          if (root == NULL) return res;
17  #ifdef RECURSIVE
18          res.push_back(root->val);
19          vector<int> left = preorderTraversal(root->left);
20          for (int i = 0; i < left.size(); i++)
21              res.push_back(left[i]);
22          vector<int> right = preorderTraversal(root->right);
23          for (int i = 0; i < right.size(); i++)
24              res.push_back(right[i]);
25  #endif
26          TreeNode *cur = root;
27          stack<TreeNode *> st;
28          while (cur != NULL || !st.empty()) {
29              if (cur != NULL) {
30                  res.push_back(cur->val);
31                  st.push(cur);
32                  cur= cur->left; // push down all the way to bottom most left node
33              } else {
34                  cur = st.top(); st.pop();
35                  cur = cur->right;
36              }
37
38          }
39          return res;
40      }
41  };
```

DFS using stack is cleanest!

```
42          TreeNode *cur = root;
43          stack<TreeNode *> st;
44          st.push(root);
45          while (!st.empty()) {
46             TreeNode *n = st.top(); st.pop();
47              res.push_back(n->val);
48              if (n->right) st.push(n->right);
49              if (n->left) st.push(n->left);
50          }
51          return res;
52      }
```

LC string#4.3: 94. Binary Tree Level InOrder Traversal:

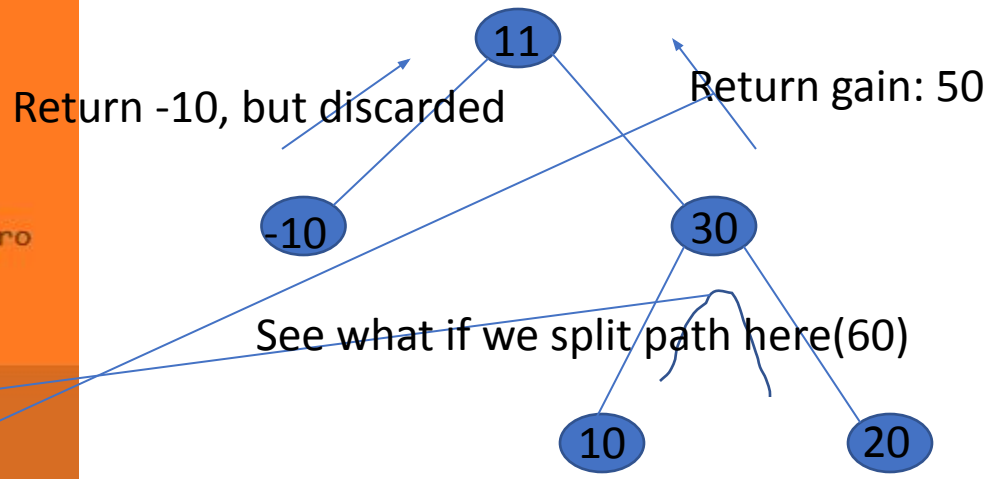
```
12  class Solution {
13  public:
14      vector<int> inorderTraversal(TreeNode* root) {
15          vector<int> res;
16          if (root == NULL) return res;
17  #ifdef RECURSIVE
18          vector<int> left = preorderTraversal(root->left);
19          for (int i = 0; i < left.size(); i++)
20              res.push_back(left[i]);
21          res.push_back(root->val);
22          vector<int> right = preorderTraversal(root->right);
23          for (int i = 0; i < right.size(); i++)
24              res.push_back(right[i]);
25  #endif
26          TreeNode *cur = root;
27          stack<TreeNode *> st;
28          while (cur != NULL || !st.empty()) {
29              if (cur != NULL) {
30                  st.push(cur);
31                  cur= cur->left; // push down all the way to bottom most left node
32              } else {
33                  cur = st.top(); st.pop();
34                  res.push_back(cur->val);
35                  cur = cur->right;
36              }
37
38          }
39          return res;
40      }
41  };
```

LC string#4.4: 102. Binary Tree PostOrder Traversal:

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == NULL) return res;
#ifdef RECURSIVE
        vector<int> left = preorderTraversal(root->left);
        for (int i = 0; i < left.size(); i++)
            res.push_back(left[i]);
        vector<int> right = preorderTraversal(root->right);
        res.push_back(root->val);
        for (int i = 0; i < right.size(); i++)
            res.push_back(right[i]);
#endif

        stack<TreeNode *> st;
        st.push(root);
        while (!st.empty()) {
            TreeNode* cur = st.top();
            st.pop();
            res.insert(res.begin(), cur->val); // this makes all existing one shift to right
            if (cur->left) st.push(cur->left); // left node first push
            if (cur->right) st.push(cur->right);
        }

        return res;
    }
};
```

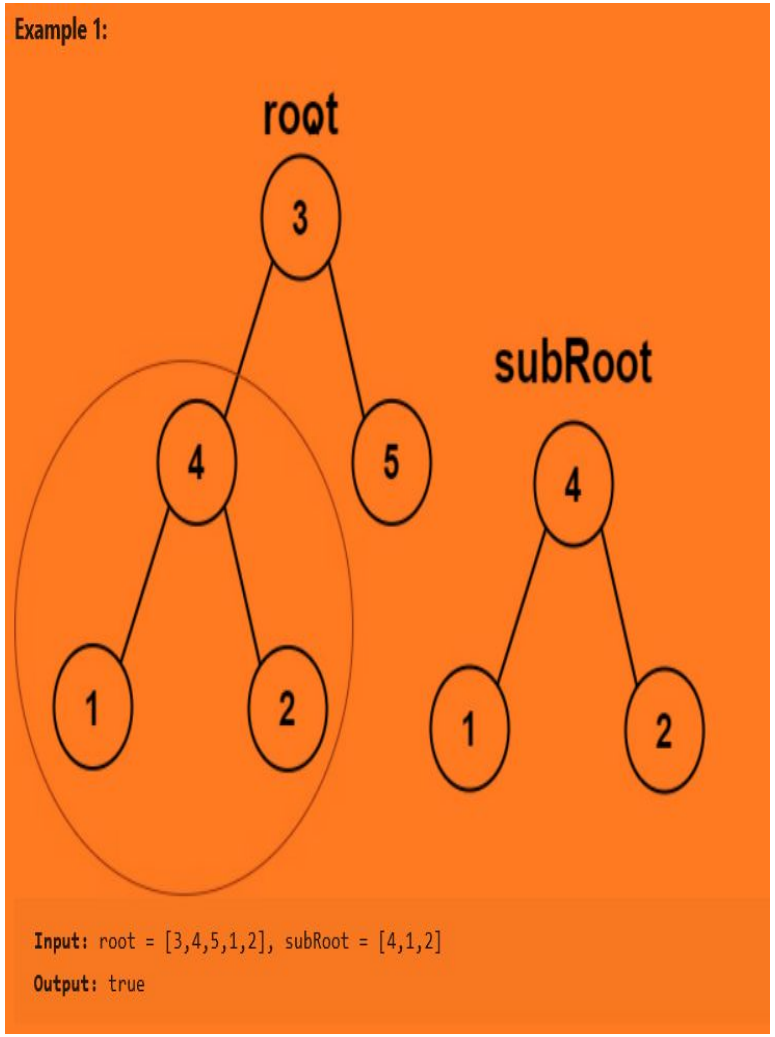
LC tree#5: 124. Binary Tree Maximum Path Sum(HARD):

```
12  class Solution {
13      int res = INT_MIN;
14  public:
15      // return gain of node without split path
16      int max_gain(TreeNode *n) {
17          if (n == NULL) return 0;
18
19          // if gain is negative, we will not take it, so set to zero
20          int left_gain = max(max_gain(n->left), 0);
21          int right_gain = max(max_gain(n->right), 0);
22
23          // check what is value if we split path at this node
24          if (n->val + left_gain + right_gain > res)
25              res = n->val + left_gain + right_gain;
26
27          return n->val + max(left_gain, right_gain);
28      }
29
30      int maxPathSum(TreeNode* root) {
31          if (root == NULL) return 0;
32
33          max_gain(root);
34
35          return res;
36      }
37  };
```



LC tree#6:

572. Subtree of Another Tree:



- Generate list of nodes starting from root
- Using each node to do same tree checking
- Same tree check using traversal to generate vector<int> to compare.

```
class Solution {
public:
    vector<int> levelTraver(TreeNode* root) {
        vector<int> res;
        if (root == NULL) return res;
        queue<TreeNode *> Q;
        Q.push(root);
        TreeNode *dummy = new TreeNode(INT_MAX);
        while (!Q.empty()) {
            TreeNode *n = Q.front(); Q.pop();
            res.push_back(n->val);
            if (n->val != INT_MAX) {
                Q.push(n->left ? n->left:dummy);
                Q.push(n->right ? n->right:dummy);
            }
        }
        return res;
    }
    bool sameTree(TreeNode *p, TreeNode *q) {
        vector<int> p_v = levelTraver(p);
        vector<int> q_v = levelTraver(q);
        return p_v == q_v;
    }
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
        vector<TreeNode *> nodes;
        if (root == NULL) return false;

        queue<TreeNode *> Q;
        Q.push(root);
        while (!Q.empty()) {
            TreeNode *n = Q.front(); Q.pop();
            nodes.push_back(n);
            if (n->left) Q.push(n->left);
            if (n->right) Q.push(n->right);
        }
        for (int i = 0; i < nodes.size(); i++) {
            if (nodes[i]->val == subRoot->val) {
                if (sameTree(nodes[i], subRoot))
                    return true;
            }
        }
        return false;
    }
};
```

LC tree#7: 105. Construct Binary Tree from Preorder and Inorder Traversal:

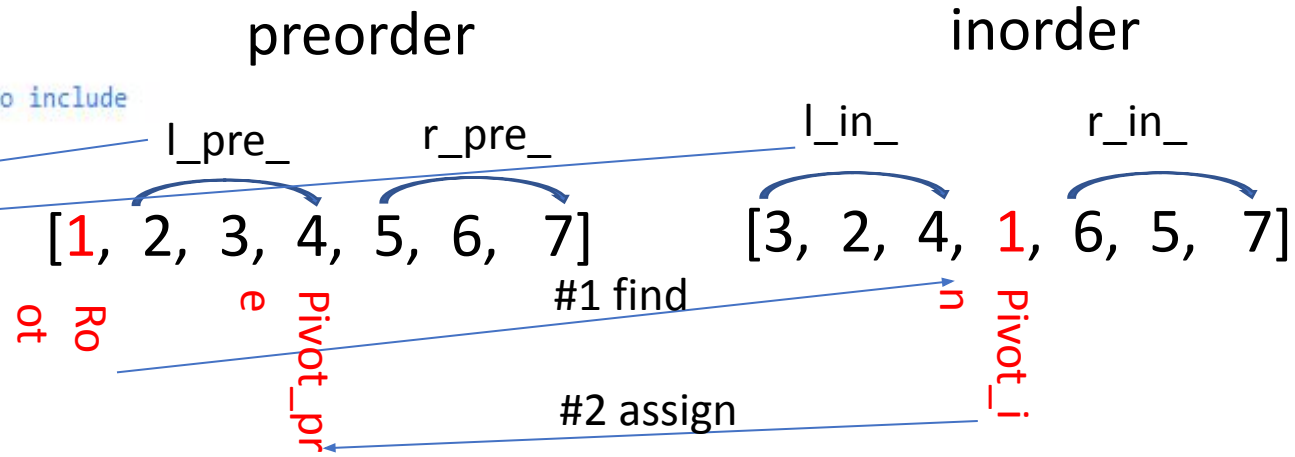
```
12  class Solution {
13  public:
14      TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
15          //first or preorder is for root
16          if (preorder.size() == 0) return NULL;
17
18          TreeNode *root = new TreeNode(preorder[0]); if (preorder.size() == 1) return root;
19          // find the pivot position of root node inside inorder
20          // (because of unique value) using root value
21          int pivot_in = find(inorder.begin(), inorder.end(), preorder[0]) - inorder.begin();
22          //partition preorder, use pivot_in (inclusive)
23          int pivot_pre = pivot_in;
24          vector<int> l_pre_, r_pre_;
25          for (int j = 1; j <= pivot_pre; j++)  // exclude root , start with 1
26              l_pre_.push_back(preorder[j]);
27          for (int j = pivot_pre+1  ; j < preorder.size(); j++)
28              r_pre_.push_back(preorder[j]);
29          //partition inorder, pivot is pivot_in
30          vector<int> l_in_, r_in_;
31          for (int j = 0; j < pivot_in; j++)  // pivot_in is root, not to include
32              l_in_.push_back(inorder[j]);
33          for (int j = pivot_in+1; j < inorder.size(); j++)
34              r_in_.push_back(inorder[j]);
35
36          root->left = buildTree(l_pre_, l_in_);
37          root->right = buildTree(r_pre_, r_in_);
38
39          return root;
40      }
41  };
```

#2

Given peorder and inorder, this precisesly define a tree
First value from preorder is ROOT, which can be used
To find the partition of LEFT and RIGHT!
If only preorder/inorder given, unless it is a full balanced tree
If postorder is given instead of preorder, find root using last!

Time & Space: O(n)

#1



LC tree#8: 226. serialize and deserialize binary tree:

```
class Codec {
public:

    // Encodes a tree to a single string following preorder sequence
    string serialize(TreeNode* root) {
        if (root == NULL) return "9999,"; // flag as end

        string res = to_string(root->val) + ",";
        res.append(serialize(root->left));
        res.append(serialize(root->right));
        return res;
    }
    int pos = 0; // use to track current proccessed node
    TreeNode * buildTree(vector<int>& vals) {
        if (vals[pos] == 9999) {
            pos++; return NULL;
        }
        TreeNode * root = new TreeNode(vals[pos]); pos++;
        root->left = buildTree(vals);
        root-> right = buildTree(vals);
        return root;
    }
    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        vector<int> vals;
        int s = 0, len = 0;
        for (int i = 0; i < data.size(); i++) {
            if (data.at(i) != ',')
                len++;
            else {
                vals.push_back(stoi(data.substr(s, len)));
                s = i+1;
                len = 0;
            }
        }
        return buildTree(vals);
    }
};
```

Use simple recursion/dfs to do preorder traversal
Using 9999 as NULL node.
Processing serialized string into vector<int> and use
Recursion to deserialize tree node.
Use variable “pos” to track position.

LC tree#9: 98. Validate Binary Search Tree:

```
12  class Solution {
13  public:
14      bool helper (TreeNode *root, int64_t low, int64_t high) {
15          if (root->val > low && root->val < high) {
16              if (root->left)
17                  if (!helper(root->left, low, root->val)) return false;
18              if (root->right)
19                  if (!helper(root->right, root->val, high)) return false;
20          } else {
21              return false;
22          }
23          return true;
24      }
25      bool isValidBST(TreeNode* root) {
26          if (root == NULL) return false;
27          int64_t LOW_INF = ((int64_t)INT_MIN)-1;
28          int64_t HIGH_INF = ((int64_t)INT_MAX) + 1;
29          return helper(root, LOW_INF, HIGH_INF);
30      }
31  };
```

- Use int64_t LOW_INF = (int64_t)INT_MIN-1
- Use int64_t HIGH_INF = (int64_t)INT_MAX+1
- To check left node, need pass down current node value as HIGH
- To check right node, need pass down current node value as LOW
- If there is any false return, return back all the way.

LC tree#10: 230. Kth Smallest Element in a BST:

```
12  class Solution {
13  public:
14
15      void inorderT(TreeNode * root, vector<int>& list) {
16          if (!root) return;
17          if (root->left) inorderT(root->left, list);
18          list.push_back(root->val);
19          if (root->right) inorderT(root->right, list);
20      }
21      int kthSmallest(TreeNode* root, int k) {
22          vector<int> list;
23          inorderT(root, list);
24          return list[k-1];
25      }
26  };
```

BST inorder traversal will give list of Value in ascending order.

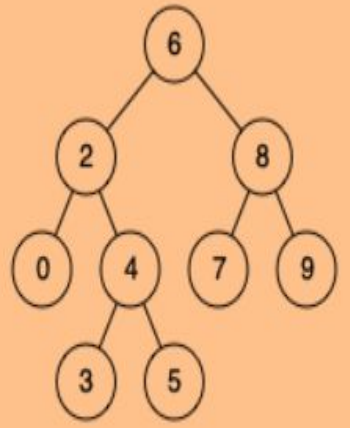
```
int hight_dfs(Node* node, int &res) {
    if (node == NULL) return -1;
    if (node->children.size() == 0) return 0;

    vector<int> heights;
    for (auto n: node->children) {
        heights.push_back(hight_dfs(n, res));
    }
    sort(heights.begin(), heights.end(), greater<int>());
    if (heights.size() >= 2) {
        res = max(res, heights[0] + heights[1] + 2); // add two: one for each side
    }
    return heights[0]+1;
}
```

LC tree#11: 235. Lowest Common Ancestor of a Binary Search Tree:

- Find p in BST into vector<int>
- Find q in BST into unordered_map<int, TreeNode*>

```
11  class Solution {
12  public:
13      void findNodeList(TreeNode * root, TreeNode *p, vector<int> &list) {
14          TreeNode *cur = root;
15          while (cur && cur->val != p->val) {
16           list.insert(list.begin(), cur->val);
17              (cur->val > p->val)? (cur = cur->left):(cur = cur->right);
18          }
19          list.insert(list.begin(), cur->val);
20      }
21      void findNodeMap(TreeNode * root, TreeNode *p, unordered_map<int, TreeNode*> &map) {
22          TreeNode *cur = root;
23          while (cur && cur->val != p->val) {
24              map[cur->val] = cur;
25              (cur->val > p->val)? (cur = cur->left):(cur = cur->right);
26          }
27          map[cur->val] = cur;
28      }
29      TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
30          vector<int> p_list;
31          unordered_map<int, TreeNode*> q_map;
32          findNodeList(root, p, p_list);
33          findNodeMap(root, q, q_map);
34          for (int i = 0; i < p_list.size(); i++){
35              if (q_map.find(p_list[i]) != q_map.end())
36                  return q_map[p_list[i]];
37          }
38          return NULL;
39      }
40  };
```

Example 1:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

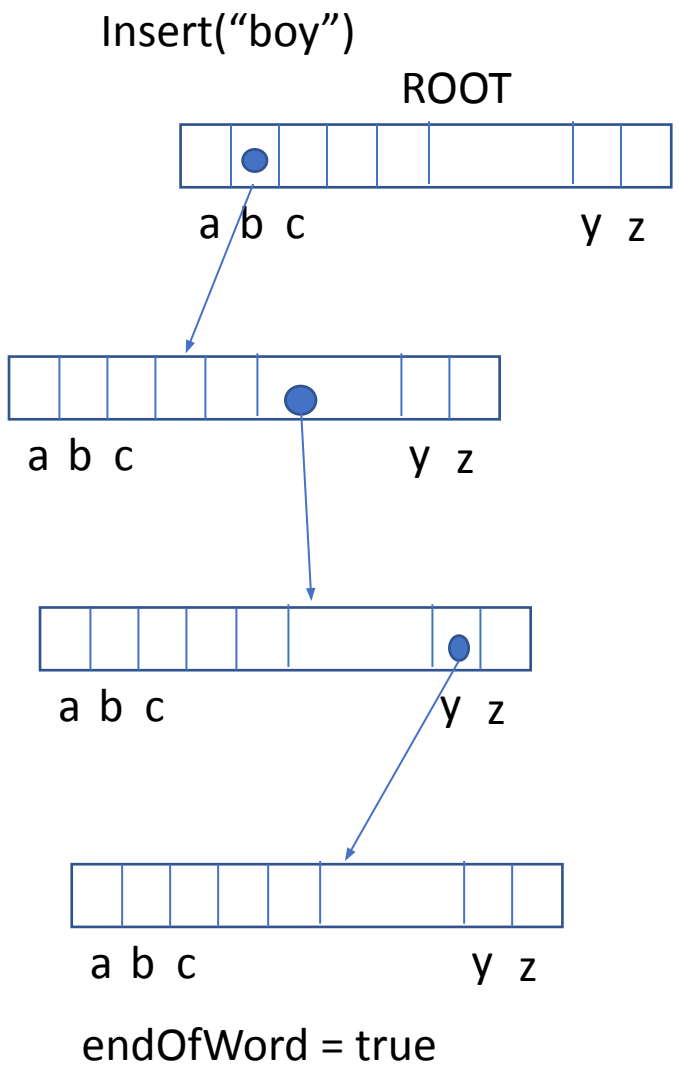
Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

236. Lowest Common Ancestor of a Binary Tree

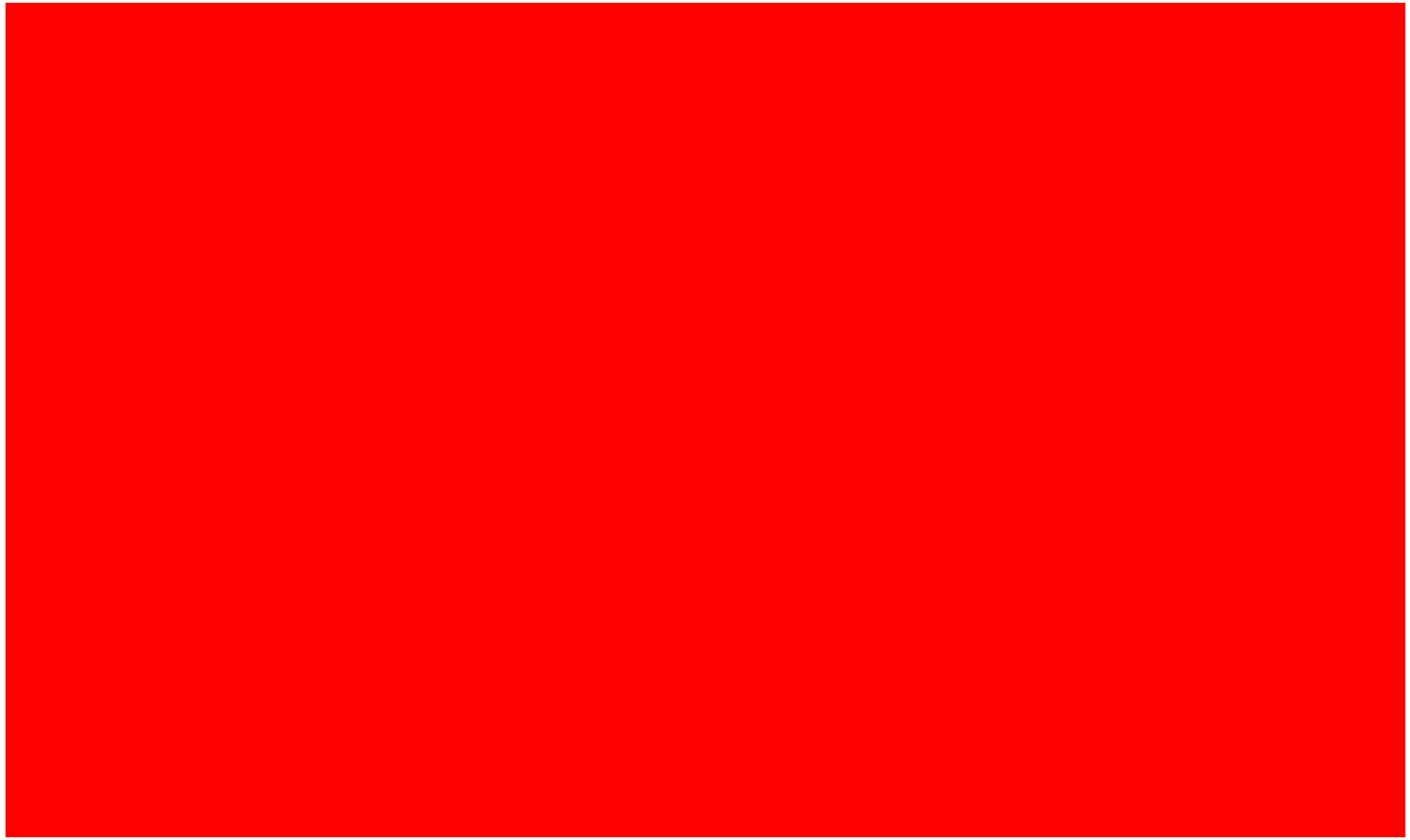
LC tree#12: 208. Implement Trie (Prefix Tree)

```
class Trie {
    Trie* children[26];
    bool endOfWord;

public:
    /** Initialize your data structure here. */
    Trie() {
        for (int i = 0; i < 26; i++) children[i] = NULL;
        endOfWord = false;
    }
    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie *p = this;
        for (int i = 0; i < word.length(); i++) {
            char c = word.at(i); // each character is used as key into children
            if (p->children[c-'a'] == NULL) {
                Trie *t = new Trie();
                p->children[c-'a'] = t;
            }
            p = p->children[c-'a'];
        }
        p->endOfWord = true;
    }
    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie *p= this;
        for (int i = 0; i < word.length(); i++) {
            char c = word.at(i);
            p = p->children[c-'a'];
            if (!p) return false;
        }
        return p->endOfWord;
    }
    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        Trie *p= this;
        for (int i = 0; i < prefix.length(); i++) {
            char c = prefix.at(i);
            p = p->children[c-'a'];
            if (!p) return false;
        }
        return (p != NULL);
    }
};
```



LC tree#13: 211. Design Add and Search Words Data Structure: (essential trie)

LC tree#14: 212. Word Search II: (HARD)



LC Array#1: 1. Two Sum (easy)

Using unordered_map<int, int> to record Index and value for search: O(n)

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> umap;
        unordered_map<int, int>::iterator itr;
        vector<int> res;

        for (int i = 0; i < nums.size(); i++) {
            int reminder = target - nums[i];
            itr = umap.find(reminder);
            if (itr != umap.end()) {
                res.push_back(i);
                res.push_back(itr->second);
                break;
            } else {
                umap[nums[i]] = i;
            }
        }
        return res;
    }
};
```

LC Array#2: 121. Best Time to Buy and Sell Stock (I), only one transaction 122: ((II) allow to buy and sell same day, no limit of transactions

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int cost = prices[0];
        int profit = 0;
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] >= cost) {
                profit = max(profit, prices[i] - cost);
            } else {
                cost = prices[i]; // starting over if we see lower co
            }
        }
        return profit;
    }
};
```

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int profit = 0;
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] > prices[i-1]) { // for any day, if we seen today price is higher
               profit += prices[i] - prices[i-1]; // we always say we bought it yesterday
            }           // if tomorrow is even higher, we said we bought it back yesterday
        }
        return profit;
    }
};
```

- a

LC Array#3: 217. Contains Duplicate (easy) 219: Contains Duplicate II

```cpp
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        unordered_set<int> seen;
        for (int i = 0; i < nums.size(); i++) {
            if (seen.find(nums[i]) == seen.end()) {
                seen.insert(nums[i]);
            } else {
                return true;
            }
        }
        return false;
    }
};
```

Example 1:

```
Input: nums = [1,2,3,1]
Output: true
```

Example 2:

```
Input: nums = [1,2,3,4]
Output: false
```

- Scan thru list, use hashSet to record if not seen

```cpp
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k) {
        unordered_map<int, int> map;
        for (int i = 0; i < nums.size(); i++) {
            if(map.find(nums[i]) == map.end()) {
                map[nums[i]] = i;
            } else {
                int j = map[nums[i]];
                if (i - j <= k)
                    return true;
                else
                    map[nums[i]] = i;
            }
        }
        return false;
    }
};
```

LC Array#4: 238. Product of Array Except Self

```cpp
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> left(nums.size()), right(nums.size());
        int tmp = 1;
        left[0] = tmp;
        for (int i=1; i < nums.size(); i++){
            tmp *= nums[i-1];
            left[i] = tmp;
        }
        tmp = 1;
        right[nums.size()-1] = tmp;
        for (int i = nums.size()-2; i >= 0; i--){
            tmp *= nums[i+1];
            right[i] = tmp;
        }

        vector<int> res(nums.size());
        for (int i= 0; i < nums.size(); i++) {
            res[i] = left[i]*right[i];
        }
        return res;
    }
};
```

- One pass starting from beginning to make LEFT product
- Another pass starting from end to make RIGHT product

LC Array#5: 53. Maximum Subarray

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxSum = INT_MIN;
        int curSum = 0;
        for (int i = 0; i < nums.size(); i++) {
            curSum += nums[i];
            maxSum = max(maxSum, curSum);
            if (curSum < 0) curSum = 0;
        }

        return maxSum;
    }
};
```

- Use two variables: curSum, maxSum
- Reset to curSum back to zero if it is negative

LC Array#6: 152. Maximum Product Subarray

```cpp
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int maxRes = INT_MIN;

        int product = 1;
        for (int i =0; i < nums.size(); i++) { //left to right
            product *= nums[i];
            maxRes = max(product, maxRes);
            if (product == 0) product = 1;
        }
        product = 1;
        for (int i = nums.size()-1; i >= 0; i--) { // right to left
            product *= nums[i];
            maxRes = max(product, maxRes);
            if (product == 0) product = 1;
        }

        return maxRes;
    }
};
```

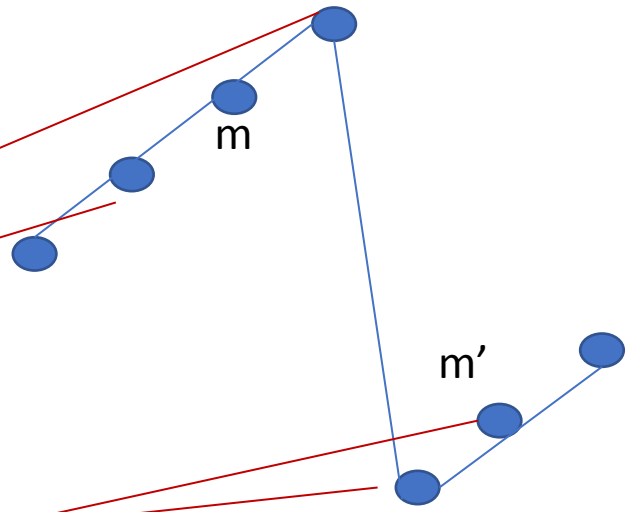
- Use two variables: maxRes, product
- Do product from left to right, track maxRes, if product becomes 0, reset to 1
- Do product from right to left, track maxRes, if product becomes 0, reset to 1

Binary search: three forms

int

LC Array#7: 153. Find Minimum in Rotated Sorted Array in O(log n) time.

```cpp
class Solution {
public:
    int findMin(vector<int>& nums) {
        if (nums.size() == 1) return nums[0];
        if (nums[0] < nums[nums.size()-1]) return nums[0];

        int l = 0, r= nums.size() - 1;
        while (l <= r) {
            if (r == l) return nums[l]; // special handling
            if (r-l ==1) return (min(nums[l], nums[r])); // special handling

            int m = l + (r-l)/2;
            if (nums[m] > nums[0]) {
                if (m+1 < nums.size() && nums[m] > nums[m+1]) {
                    return nums[m+1];
                } else {
                    l = m + 1;
                }
            } else {
                if (m-1 >= 0 && nums[m-1] > nums[m]) {
                    return nums[m];
                } else {
                    r = m -1;
                }
            }
        }
        return INT_MIN;
    }
};
```



- Special handling when r==l and r-l ==1 inside loop
- Special handling when nums[0] is minmum

LC Array#7.1: 162. Find Peak Element in O(log n) time.

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0, r = nums.size() - 1;
        if (r == 0) return 0;
        while (l < r) {
            if (r-l == 1) return nums[l] > nums[r]? l:r;
            int m = (l+r)/2;
            if (nums[m] < nums[m+1]) // climbing
                l = m;
            else
                r = m;
        }
        return l;
    }
};
```

Input: nums = [1,2,1,3,5,6,4]

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

A peak element is an element that is strictly greater than its neighbors.

Given an integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that `nums[-1] = nums[n] = -∞`.

You must write an algorithm that runs in `O(log n)` time.

LC Array#8: 33. Search in Rotated Sorted Array using O(log N)

```cpp
class Solution {
public:
    int bst(vector<int>& nums, int l, int r, int target) {
        int res = -1;
        while (l <=r) {
            int m = (l+r)/2;
            if (nums[m] == target) {
                res = m; break;
            } else {
                (nums[m] < target) ? (l=m+1):(r=m-1);
            }
        }
        return res;
    }
    int search(vector<int>& nums, int target) {
        int pivot,  l = 0 , r = nums.size()-1;
        if (nums.size() == 1)
            return nums[0] == target? 0:-1;
        if (nums[l] < nums[r]) return bst(nums, l, r, target);
        while (l<=r) {
            if (l == r) {
                pivot = l; break;
            }
            if (r -l == 1) {
                pivot = nums[l] < nums[r]? l:r;
                break;
            }
            int m = (l+r)/2;
            if (m> 0 && (m+1) < (nums.size()) && nums[m+1] > nums[m] && nums[m-1] > nums[m]) {
                pivot = m;
                break;
            }
            if (m>0 && (m+1) < (nums.size()) && nums[m-1] < nums[m] && nums[m] > nums[m+1]) {
                pivot = m+1;
                break;
            }
            if (nums[m] > nums[0])
                l = m+1;
            else
                r = m -1;
        }
        if (target >= nums[0])
            return bst(nums, 0, pivot-1, target);
        else
            return bst(nums, pivot, nums.size()-1, target);
    }
};
```

- If the array size is 1, return result
- If array is fully sorted, do binary search
- Otherwise find pivot, then decide which segment do perform binary search.

LC Array#9: 15. 3Sum.

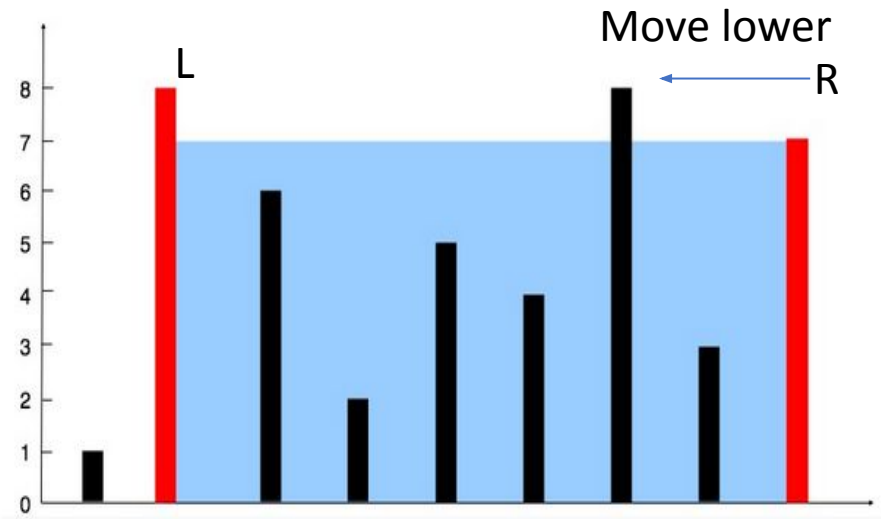
```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> res;
        if (nums.size() < 3) return res;
        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size()-2; i++) {
            if (nums[i] > 0) continue;
            if (i > 0 && (nums[i] == nums[i-1])) continue; // avoid dup
            int target = -nums[i];
            int l = i+1; int r = nums.size()-1;
            while (l < r) {
                if (nums[l] + nums[r] == target) {
                    vector<int> ans;
                    ans.push_back(nums[i]);
                    ans.push_back(nums[l]);
                    ans.push_back(nums[r]);
                    res.push_back(ans);
                    // don't break, maybe more answer
                }
                // advance l or r
                if (nums[l] + nums[r] <= target) {
                    int ll = l;
                    while ((l < nums.size() -1) && (nums[ll] == nums[l])) l++;
                } else {
                    int rr = r;
                    while ((r > 0) && (nums[rr] == nums[r])) r--;
                }
            }
        }
        return res;
    }
};
```

- Sort array
- Go thru for i, target is –nums[i]
- Using two pointers between i+1 to size()-1 to find other two numbers
- O(n*n)

LC Array#10: 11. Container With Most Water.

```cpp
class Solution {
public:
    int maxArea(vector<int>& height) {
        int res = INT_MIN;

        #ifdef BRUTEFORCE
        for (int i = 0; i < height.size() -1; i++) {
            for (int j = i+1; j < height.size(); j++) {
                int area = (j-i)*min(height[i], height[j]);
                res = max(res, area);
            }
        }
        #endif
        // two pointers
        int l = 0, r = height.size() -1;
        while (l < r) {
            int a = (r-l)*min(height[l], height[r]);
            res = max(res, a);
            if (height[l] < height[r]) {
                l++;
            } else {
                r--;
            }
        }
        return res;
    }
};
```

Example 1:



Input: height = [1,8,6,2,5,4,8,3,7]

Output: 49

Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

LC#528 PrefixSum

528. Random Pick with Weight

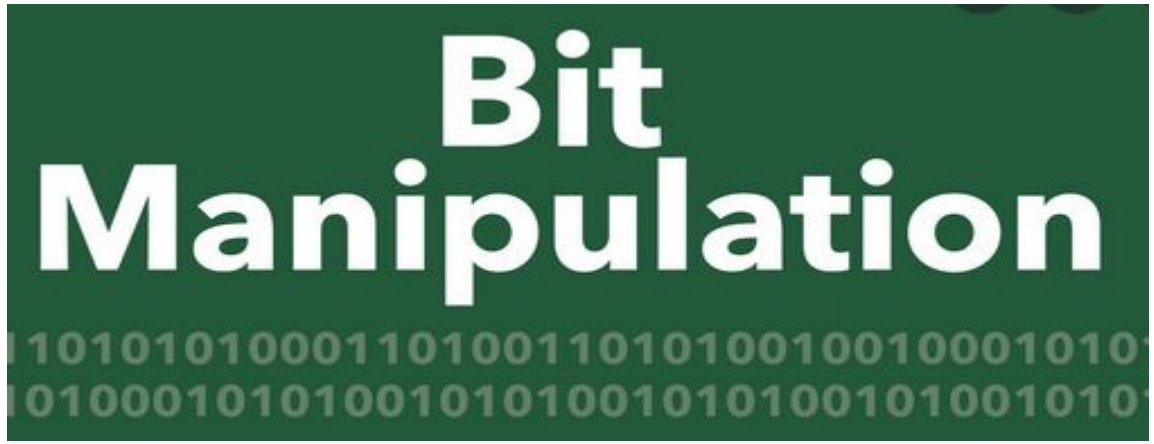
Medium 1494 3131 Add to List Share

You are given an array of positive integers `w` where `w[i]` describes the weight of `i th` index (0-indexed).

We need to call the function `pickIndex()` which **randomly** returns an integer in the range `[0, w.length - 1]`. `pickIndex()` should return the integer proportional to its weight in the `w` array. For example, for `w = [1, 3]`, the probability of picking the index `0` is `1 / (1 + 3) = 0.25` (i.e 25%) while the probability of picking the index `1` is `3 / (1 + 3) = 0.75` (i.e 75%).

More formally, the probability of picking index `i` is `w[i] / sum(w)`.

```cpp
class Solution {
    vector<int> prefixSum;
    int N;
public:
    Solution(vector<int>& w) {
        N = w.size();
        prefixSum.push_back(w[0]);
        for (int i = 1; i < N; i++) {
            prefixSum.push_back(prefixSum.back()+w[i]);
        }
    }

    int pickIndex() {
        float r = ((float)rand()) / RAND_MAX; // get a number between 0-1
        int target = (int) (r * prefixSum[N-1]);
        for (int i = 0; i < N; i++) {
            if (target < prefixSum[i]) return i;
        }
        return N-1;
    }
};
```



Types of Bitwise Operators

Operator	Name	Example	Result
&	Bitwise AND	6 & 3	2
|	Bitwise OR	10 | 10	10
^	Bitwise XOR	2^2	0
~	Bitwise 1's complement	~9	-10
<<	Left-Shift	10<<2	40
>>	Right-Shift	10>>2	2

LC BITS#1: 371. Sum of Two Integers.

```
1  class Solution {
2  public:
3      int getSum(int a, int b) {
4          long mask = 0xFFFFFFFF;
5          while (b) {
6              int sum = a ^ b;
7              int carry = ((a & b)&mask) << 1; // avoid negative# shift error
8              a = sum;
9              b = carry;
10         }
11         return a;
12     }
13 };
14
15
```

- (A & B) << 1 => carry, A^B => answer
- Use **long** mask=0xFFFF,FFFF

LC BITS#2: 268. Missing Number.

```cpp
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int res = nums.size();
        for (int i = 0; i < nums.size(); i++) {
            res ^=i;
            res ^=nums[i];
        }
        return res;
    }
};
```

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return *the only number in the range that is missing from the array.*

Follow up: Could you implement a solution using only `O(1)` extra space complexity and `O(n)` runtime complexity?

Example 1:

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.
```

- (A & B) << 1 => carry, A^B => answer

LC BITS#3: 190. Reverse Bits.

```cpp
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        unsigned int l_mask = 1 << 31, r_mask = 1;
        for (int i = 0; i < 16; i++) {
            int r_ = (n & r_mask);
            int l_ = (n & l_mask);
            n = (n & ~l_mask) | (r_?l_mask:0);
            n = (n & ~r_mask) | (l_?r_mask:0);
            l_mask = l_mask >>1 ;
            r_mask = r_mask << 1;
        }
        return n;
    }
};
```

- a

LC BITS#4: counting 1 bit of an interger.

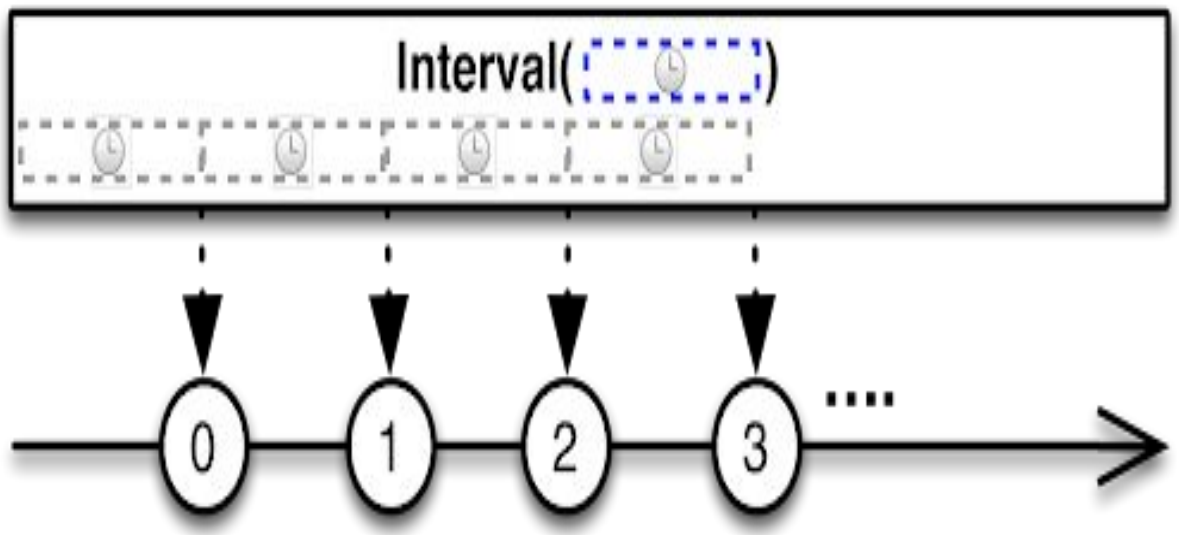
```
while (n) { counter++; n &=(n-1);}
__builtin_popcount(n);
```

- __builtin_popcount(): count number of 1 bit, leverage “popcnt” instruction
- __builtin_clz(): count leading zero, “lea” instruction
- __builtin_ctz(): count trailing zero, “tzcnt” instruction
- __builtin_parity(): parity check

LC BITS#5: check if a number is power of 4.

```
bool isPowerOfFour(int n) {
    return !(n&(n-1)) && (n&0x55555555);
    //check the 1-bit location;
}
```

- Check it has only one bit and bit location is 0x0101, 0101, 0101, 0101,0101,0101,0101,0101
- If it is power of 8, which is (2*2*2)N, bit location: 0x0100,1001,0010,0100,1001,0010,0100,1001 0x49249249


Interval( )
0
1
2
3
....

LC INTERVAL#1: 57. Insert Interval.

```cpp
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
        //insert newInterval into intervals to keep ascending order
        bool inserted = false;
        for (int i = 0; i < intervals.size(); i++) {
            if (newInterval[0] <= intervals[i][0]) {
                intervals.insert(intervals.begin()+i, newInterval);
                inserted = true;
                break;
            }
        }
        if (!inserted) intervals.push_back(newInterval);

        stack<vector<int>> st;
        st.push(intervals[0]);
        for (int i = 1; i < intervals.size(); i++) {
            vector<int> top = st.top();
            if (intervals[i][0] > top[1]) { // no-overlap,
                st.push(intervals[i]);
            } else {
                if (intervals[i][1] > top[1]) { // bigger than current interval
                    top[1] = intervals[i][1];   // extend current interval
                    st.pop();
                    st.push(top);
                }
            }
        }
        vector<vector<int>> res;
        while (!st.empty()) {
            res.insert(res.begin(), st.top());
            st.pop();
        }
        return res;
    }
};
```

Example 1:

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

- Given interval list is in ascending order
- Insert newInterval into given list in ascending order
- Merge all intervals in the list using STACK

LC INTERVAL#2: 56. Merge Intervals

```cpp
class Solution {
public:
    static bool cmp(vector<int> v1, vector<int> v2) {
        return v1[0] < v2[0];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.size() == 0) return res;
        sort(intervals.begin(), intervals.end(), cmp);

        vector<vector<int>> res;
        stack<vector<int>> st;
        st.push(intervals[0]);
        for (int i = 1; i < intervals.size(); i++) {
            vector<int> top = st.top();
            int c_s = top[0];
            int c_e = top[1];
            int n_s = intervals[i][0];
            int n_e = intervals[i][1];
            if (n_s <= c_e) {
                // merge
                top[1] = n_e > c_e ? n_e: c_e;
                st.pop();
                st.push(top);
            } else {
                // no overlap, simply push
                st.push(intervals[i]);
            }
        }

        while (!st.empty()) {
            res.insert(res.begin(), st.top());
            st.pop();
        }
        return res;
    }
};
```

Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

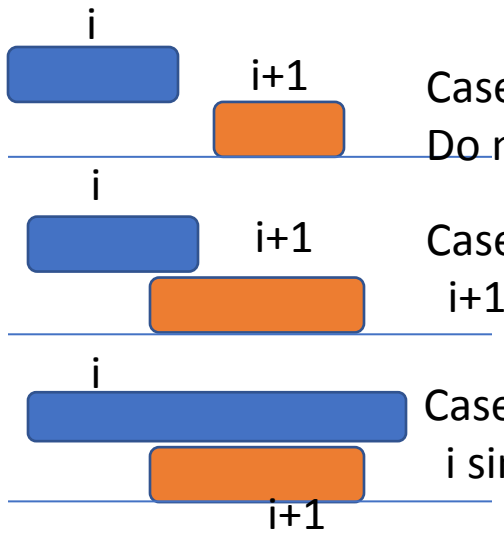
- First sort intervals by using start value in ascending order
- Use stack to merge any overlapped intervals

LC INTERVAL#3: 435. Non-overlapping Intervals.

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

```
class Solution {
public:
    static bool cmp(vector<int> v1, vector<int> v2) {
        return v1[0] < v2[0];
    }
    // firt sort the intervals based on starting value in ascending order
    // if interval[i] overlaps with interval[i+1]
    //    if interval[i+1] end is greater than interval[i], remove i+1
    //     else remove interval[i], which  is longer
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.size() < 2) return 0;

        sort(intervals.begin(), intervals.end(), cmp);
        vector<int> prev = intervals[0];
        int removals = 0;
        for (int i = 1; i < intervals.size(); i++) {
            vector<int> cur = intervals[i];
            if (cur[0] >= prev[1]) {
                prev = cur;
                continue; // non overlap
            } else {
                if (cur[1] > prev[1]) {
                    // remove cur
                    removals++;
                    // prev pointer stays
                } else {
                    prev = cur;
                    removals++;
                }
            }
        }
        return removals;
    }
};
```

- Sort all interval lists in ascending order
- Go thru interval list, there are three cases below:



LC INTERVAL#4: 252. Meeting Rooms.

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

```cpp
class Solution {
public:
    static bool cmp(vector<int> v1, vector<int> v2) {
        return v1[0]<v2[0];
    }

    bool canAttendMeetings(vector<vector<int>>& intervals) {
        if (intervals.size() < 2) return true;

        sort(intervals.begin(), intervals.end(), cmp);

        vector<int> prev = intervals[0];
        for (int i = 1; i < intervals.size(); i++) {
            vector<int> cur = intervals[i];
            if (prev[1] > cur[0])
                return false;
            else {
                prev = cur;
            }
        }

        return true;
    }
};
```

Example 1:

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: false
```

- Sort interval by starting time in ascending order
- Check if there is any overlap

LC INTERVAL#5: 253: Meeting Rooms II.

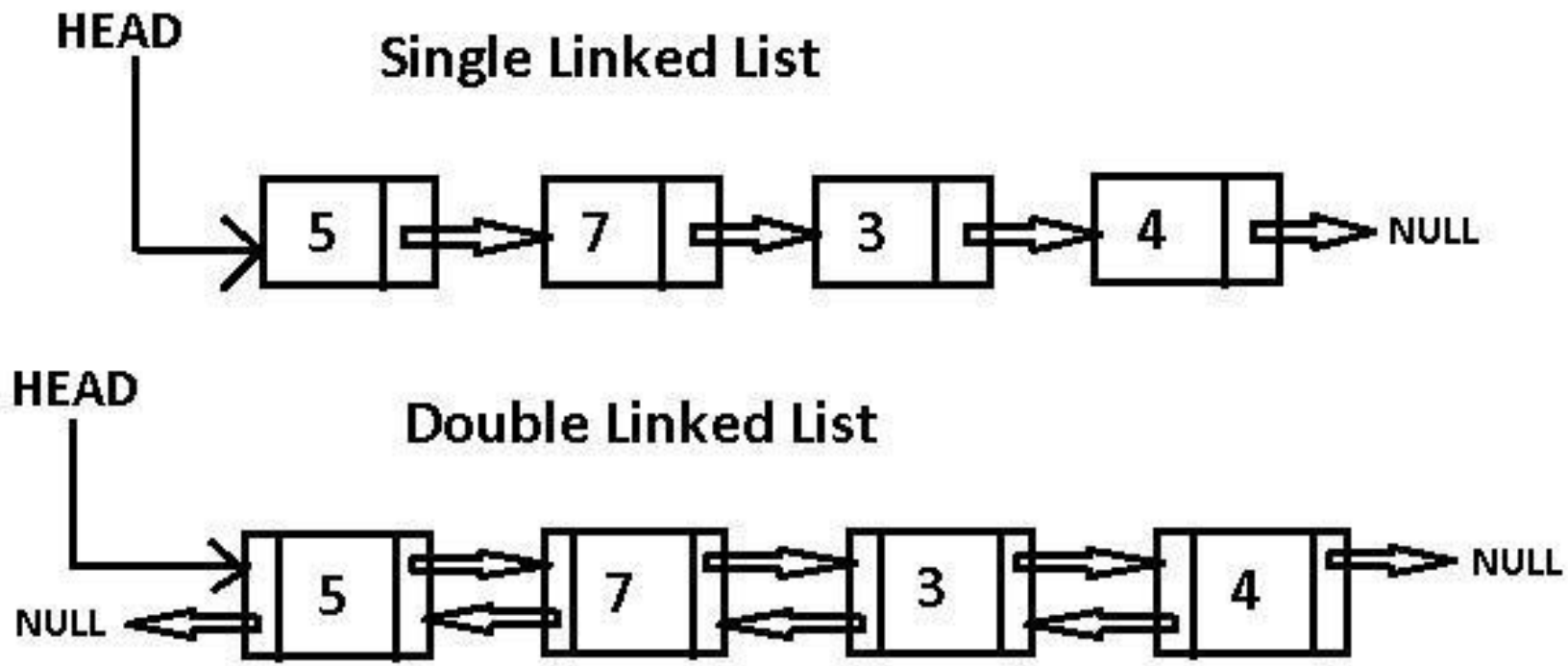
Example 1:

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
```

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return the minimum number of conference rooms required.

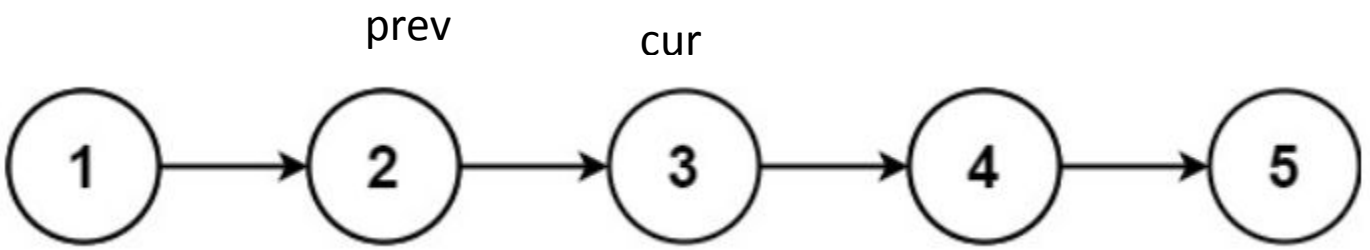
```
class Solution {
public:
    static bool cmp(int i1, int i2) {
        if (abs(i1) == abs(i2)) {
            return i1<i2;
        }
        return abs(i1) < abs(i2);
    }

    int minMeetingRooms(vector<vector<int>>& intervals) {
        vector<int> times;
        for (int i =0; i < intervals.size(); i++) {
            times.push_back(intervals[i][0]);
            times.push_back(-intervals[i][1]);
        }
        sort(times.begin(), times.end(), cmp);
        int res = 0;
        int rooms = 0;
        for (int i = 0; i < times.size(); i++) {
            if (times[i] >= 0) {
                rooms++;
                res = rooms > res? rooms:res;
            } else {
                rooms--;
            }
        }
        return res;
    }
};
```

- Merge all starting time and ending time into one single vector, but ending time as negative value
- Sort vector using absolute value, if it is equal, negative first
- Go thru vector list, see positive time, increase 1,
- See negative time, decrease one.
- Record the max rooms


HEAD
Single Linked List
5
7
3
4
NULL
HEAD
Double Linked List
NULL
5
7
3
4
NULL

LC LinkedList#1: 206. Reverse Linked List

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *prev = NULL, *cur = head;
        while (cur) {
            ListNode *tmp = cur->next;
            cur->next = prev;
            prev = cur;
            cur = tmp;
        }
        return prev;
    }
};
```



LC LinkedList#2: 141. Linked List Cycle

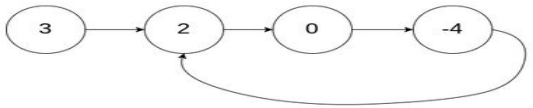
```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *s1 = head, *s2;
        if (!s1) return false;
        if (s1->next) s2 = s1->next;
        bool cycle = false;
        while (s1 && s2) {
            if (s1 == s2) {
                cycle = true;
                break;
            }
            s1 = s1->next;
            s2 = s2->next? s2->next->next:NULL;
        }
        return cycle;
    }
};
```

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

Return `true` *if there is a cycle in the linked list*. Otherwise, return `false`.

Example 1:



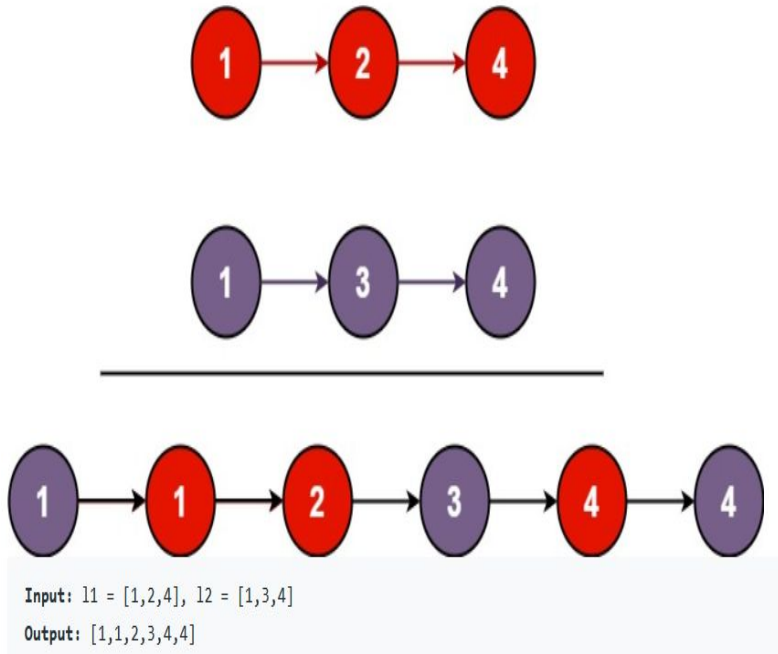
- Use two pointers: s1 move with one step, s2 move two steps
- If there is loop, s1 and s2 will never NULL and will be s1==s2 eventually.
- Or use **hashSet "seen"**

```cpp
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (head == NULL) return NULL;
        unordered_set<ListNode *> seen;
        ListNode *s1 = head;
        while (s1) {
            if (seen.find(s1) != seen.end()) return s1;
            seen.insert(s1);
            s1 = s1->next;
        }
        return NULL;
    }
};
```

LC LinkedList#3: 21. Merge Two Sorted Lists.

```
11  class Solution {
12  public:
13      ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
14          ListNode *head = NULL, *tmp, *cur;
15          if (!l1) return l2;
16          if (!l2) return l1;
17
18          while (l1 && l2) {
19              if (l1->val <= l2->val) {
20                  tmp = l1;
21                  l1 = l1->next;
22              } else {
23                  tmp = l2;
24                  l2 = l2->next;
25              }
26              if (!head) {
27                  head = tmp;
28                  cur = tmp;
29              } else {
30                  cur->next = tmp;
31                  cur = tmp;
32              }
33          }
34          if (l1) {
35              cur->next = l1;
36          }
37          if (l2) {
38              cur->next = l2;
39          }
40
41          return head;
42      }
43  };
```

Example 1:



Input: l1 = [1,2,4], l2 = [1,3,4]

Output: [1,1,2,3,4,4]

- Use “cur” to hold current latest merged node
- Use “head” to hold return value
- Compare l1 and l2, advance the smaller value pointer
- If one becomes NULL, simply connect the other remaining

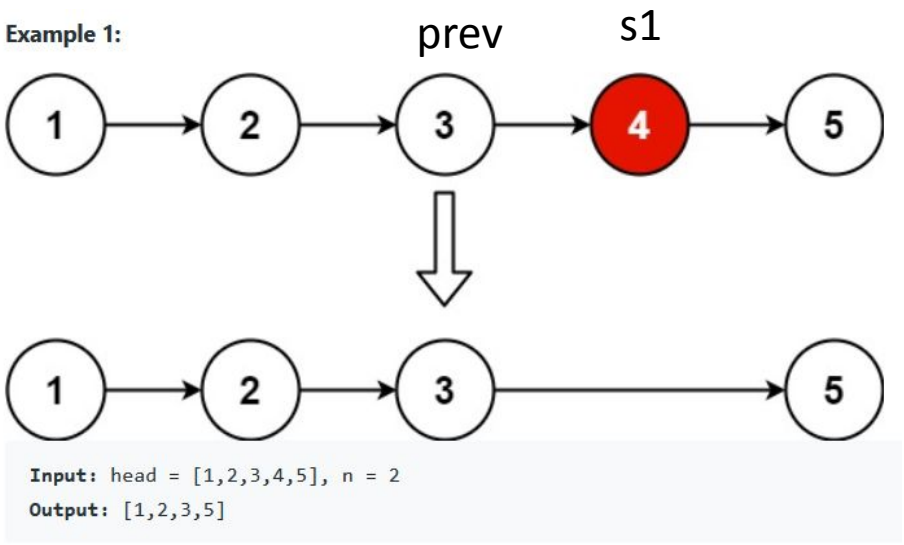
LC LinkedList#4: 23. Merge k Sorted Lists.(HARD)

```cpp
class Cmp {
    public:
    bool operator()(ListNode* n1, ListNode* n2) {
        return n1->val > n2->val; // ascending order
    }
};
class Solution {
public:
    static bool cmp(ListNode* n1, ListNode *n2) {
        return n1->val < n2->val;
    }

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode *, vector<ListNode *>, Cmp> miniHeap;
        for (int i = 0; i < lists.size(); i++) {
            ListNode *head = lists[i];
            while (head) {
                miniHeap.push(head);
                head = head->next;
            }
        }
        ListNode *root = NULL, *cur = NULL, *tmp;
        while (!miniHeap.empty()) {
            ListNode *tmp = miniHeap.top();
            tmp->next = NULL;
            if (!root) {
                root = tmp;
                cur = root;
            } else {
                cur->next = tmp;
                cur = tmp;
            }
            miniHeap.pop();
        }
        return root;
    }
};
```

- Create a miniHeap using priority_queue (or simply use vector sorting)
- Traversal all K lists, add all nodes into minheap
- Pop minheap node one by one and form a new list to return

LC LinkedList#5: 19. Remove Nth Node From End of List.

```
11  class Solution {
12  public:
13      ListNode* removeNthFromEnd(ListNode* head, int n) {
14          ListNode *s1 = head, *s2 = head, *prev = NULL;
15          for (int i = 0; i < n; i++) {
16              s2 = s2->next;
17          }
18          while (s1 && s2) {
19             prev = s1;
20              s1 = s1->next;
21              s2 = s2->next;
22          }
23          if (prev)
24              prev->next = s1->next;
25          else
26              return s1->next;
27
28          return head;
29      }
30  };
```



- Use two pointers: s2 is N step ahead of s1
- When s2 becomes NULL, S1 is the node to be removed
- Use “prev” hold node before s1

LC LinkedList#6: 143. Reorder List.

```
class Solution {
public:
    void reorderList(ListNode* head) {
        vector<ListNode *> nodes;
        ListNode* cur = head;
        while (cur) { // put all nodes into vector list with next set to NUL
            ListNode *tmp = cur->next;
            cur->next = NULL;
            nodes.push_back(cur);
            cur = tmp;
        }
        ////////////////////////////
        int N = nodes.size() -1;
        cur = NULL;
        for (int i = 0; i < N; i++) {
            if (i < (N-i)) {
                nodes[i]->next = nodes[N-i];
                if (cur)
                    cur->next = nodes[i];
                cur = nodes[N-i];
            } else if (i== (N-i)) { // point to same node
                cur->next = nodes[i]; // last node
            } else {
                break;
            }
        }
    }
};
```

You are given the head of a singly linked-list. The list can be represented as:

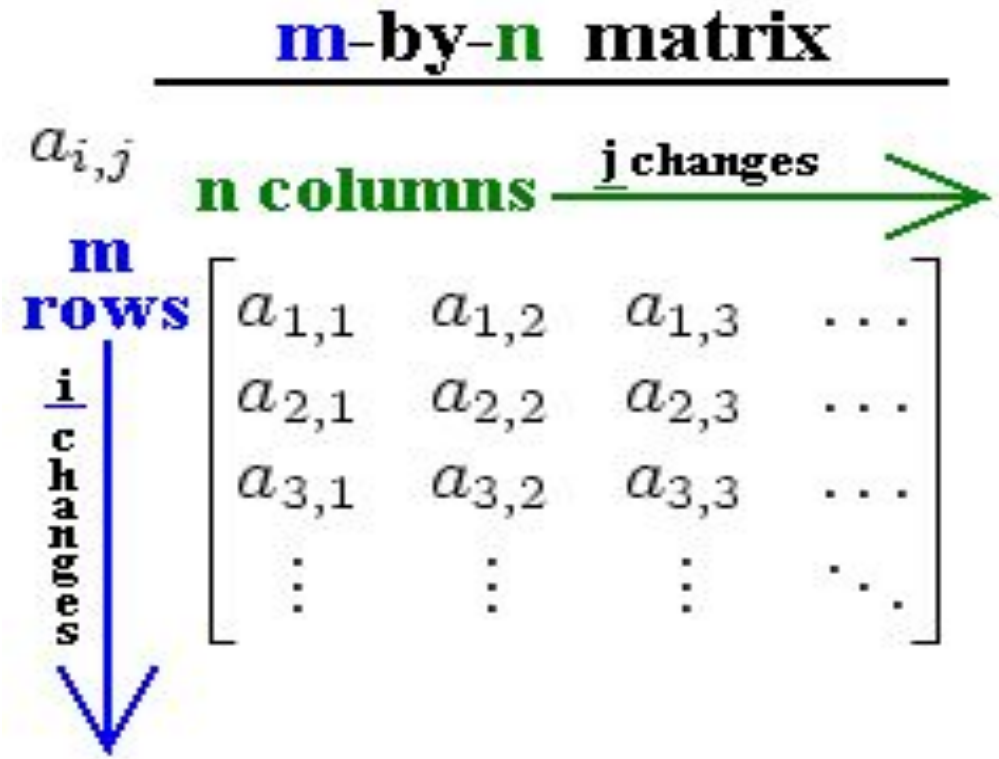
$$L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$$

Reorder the list to be on the following form:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

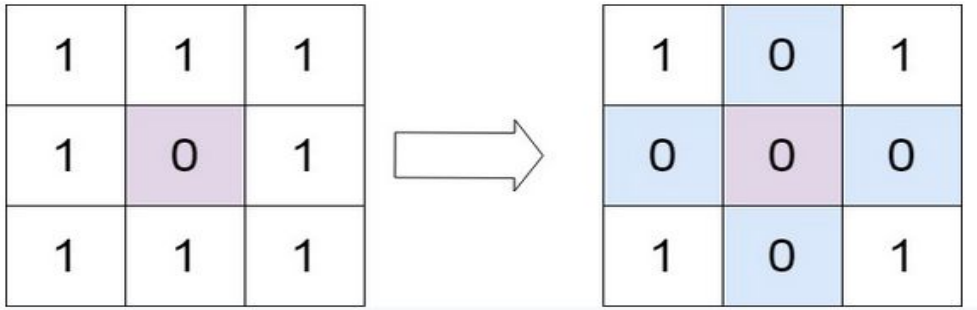
- Push all ListNode into vector list
- Take out node from vector as required to form a new list.


m-by-n matrix
$a_{i,j}$
n columns
j changes
m
rows
i
changes
$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots \\ a_{3,1} & a_{3,2} & a_{3,3} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$

LC Matrix#1: 73. Set Matrix Zeroes.

```cpp
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int R = matrix.size();
        int C = matrix[0].size();
        unordered_set<int> r_zero;
        unordered_set<int> c_zero;

        for (int r = 0; r < R; r++) {
            vector<int> row = matrix[r];
            for (int c = 0; c < C; c++) {
                if (matrix[r][c] == 0) {
                    if (r_zero.find(r) == r_zero.end()) {
                        // set this row to zero
                        r_zero.insert(r);
                    }
                    if (c_zero.find(c) == c_zero.end()) {
                        // set this column to zero
                        c_zero.insert(c);
                    }
                }
            }
        }
        for (int r = 0; r < R; r++) {
            vector<int> row = matrix[r];
            for (int c = 0; c < C; c++) {
                if (r_zero.find(r) != r_zero.end()
                   || c_zero.find(c) != c_zero.end()) {
                    matrix[r][c] = 0;
                }
            }
        }
    }
};
```

Example 1:



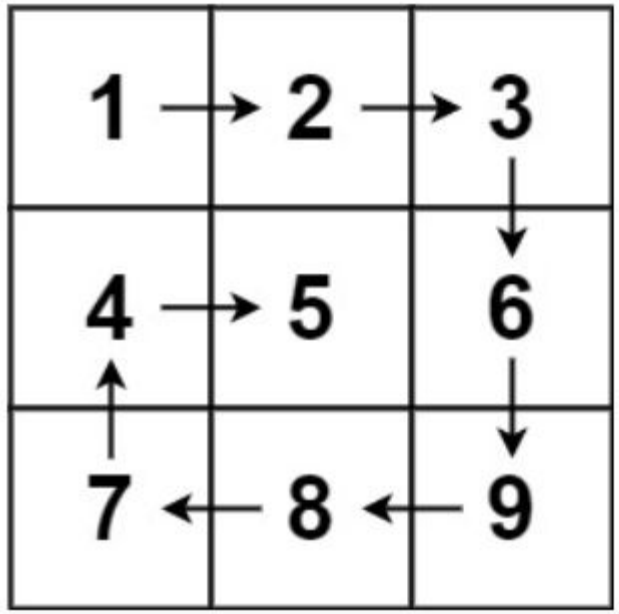
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]

- Scan thru matrix to mark down rows and columns (into hashset) to be set to zero
- Second time go thru matrix again and check hashSets , either r or c in the its set, set matrix cell to be zero

LC Matrix#2: 54. Spiral Matrix.

```cpp
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> res;
        vector<vector<int>> DIRS {{0, 1}, {1, 0}, {0, -1}, {-1,0}};
        int R = matrix.size();
        int C = matrix[0].size();
        int r = 0, c = 0, mov_dir = 0;
        int VISITED = INT_MAX;
        while (1) {
            int r_off = DIRS[mov_dir][0];
            int c_off = DIRS[mov_dir][1];
            res.push_back(matrix[r][c]);
            matrix[r][c] = VISITED;
            int nr = r + r_off;
            int nc = c+ c_off;
            if (nr < 0 || nr >= R || nc < 0 || nc >= C || matrix[nr][nc] == INT_MAX) {
                // time to change direction
                mov_dir = (mov_dir +1)%4;
                r = r+DIRS[mov_dir][0];
                c = c+DIRS[mov_dir][1];
                if (r < 0 || r >= R || c < 0 || c >= C) break;
                if (matrix[r][c] == INT_MAX) break;
            } else {
                r = nr;
                c = nc;
            }
        }
        return res;
    }
};
```

Example 1:



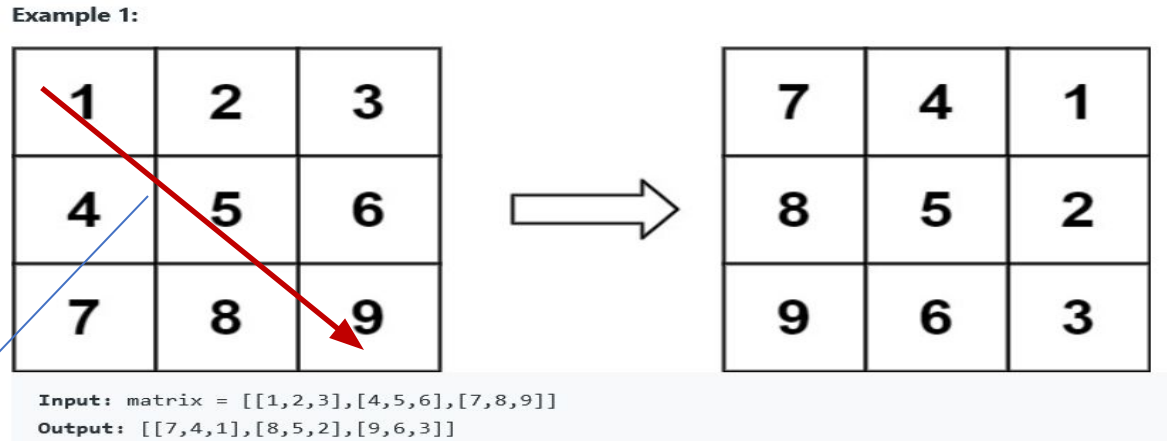
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [1,2,3,6,9,8,7,4,5]

- Set up four directions: L->R, U->D, R->L, D->U with offset
- Starts with [0][0], each visited node filled with INT_MAX
- When we move node, change direction when either reach out of boundary or VISTED node

LC Matrix#3: 48. Rotate Image.

You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).




```cpp
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int N = matrix.size();
        for (int r = 0; r < N; r++) {
            for (int c = r; c < N; c++) { // column starts with r !!!
                int t = matrix[r][c];
                matrix[r][c] = matrix[c][r];
                matrix[c][r] = t;
            }
        }

        // roate clockwise , reverse each row
        for (int r = 0; r < N; r++) {
            int l_ = 0, r_ = N-1;
            while (l_ < r_) {
                int t = matrix[r][l_];
                matrix[r][l_] = matrix[r][r_];
                matrix[r][r_] = t;

                l_++; r_--;
            }
        }
        #ifdef ROTATE_COUNTERCLOCK
        // if rotate counterlockwise, reverse each column
        for (int c = 0; c < N; c++) {
            int t_ = 0, b_ = N-1;
            while (t_ < b_) {
                int t = matrix[t_][c];
                matrix[t_][c] = matrix[b_][c];
                matrix[b_][c] = t;

                t_++; b_--;
            }
        }
        #endif
    }
};
```

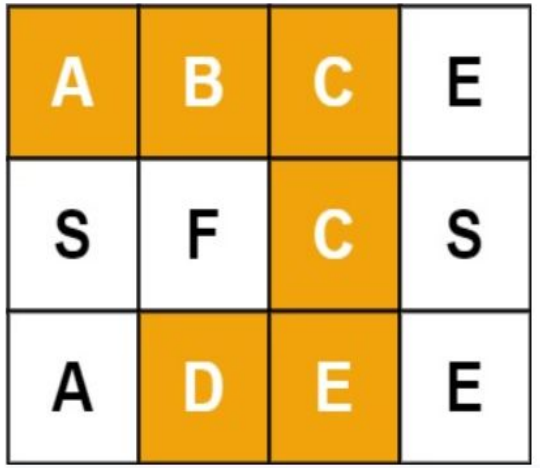
- Flip value diagnose
- Reverse each row for clockwise
- (Reverse each column for counter-clockwise)

LC Matrix#4: 79. Word Search.

- Key points:

```cpp
class Solution {
public:
    int R = 0;
    int C = 0;
    //vector<vector<bool>> map;

    bool backtrack(int r, int c, int pos, string word, vector<vector<char>>& board , vector<vector<bool>> map) {
        if (pos == word.length()-1) return true;

        if (map[r][c]) return false;

        map[r][c] = true;
        // check around neighbors (T, D, L, R) if it is matches next char, then call recursively
        vector<vector<int>> DIRS { {-1, 0}, {1, 0}, {0,1}, {0,-1} };
        for (int i = 0; i < DIRS.size(); i++) {
            int nr = DIRS[i][0] + r;
            int nc = DIRS[i][1] + c;
            if (nr < 0 || nr >= R || nc < 0 || nc >= C || map[nr][nc]) continue;

            if (board[nr][nc] == word.at(pos+1)) {
                vector<vector<bool>> nmap = map; // pass new map down
                if (backtrack(nr, nc, pos+1, word, board, nmap))
                    return true;
            }
        }
        return false;
    }

    bool exist(vector<vector<char>>& board, string word) {
        int r = 0, c = 0, pos = 0;
        R = board.size();
        C = board[0].size();

        vector<vector<bool>> map;
        for (int i = 0; i < R; i++) {
            vector<bool> t(C, false);
            map.push_back(t);
        }

        // find all possible first character to get started
        for (int r = 0; r < R; r++) {
            for (int c = 0; c < C; c++) {
                if (board[r][c] == word.at(0))
                    if (backtrack(r, c, pos, word, board, map))
                        return true;
            }
        }
        return false;
    }
};
```

Example 1:



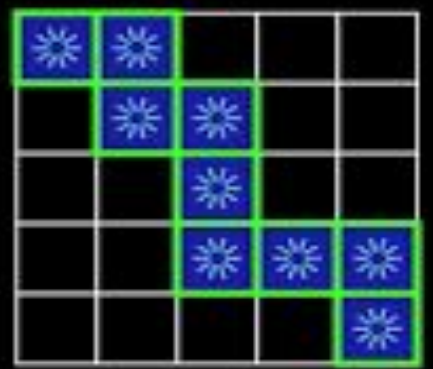
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

Output: true

LC Matrix#5: TBA.

- (A & B) << 1 => carry, A^B => answer

Dynamic Programming



LC DP#1: 70. Climbing Stairs.

```
class Solution {
public:
    vector<int> memo;
    bool init = true;
    int climbStairs(int n) {
        if (n <=2) return n;
        if (init) {
            memo = vector<int>(n+1, 0);
            init = false;
        }
        if (memo[n] > 0) return memo[n];
        int res = climbStairs(n-1) + climbStairs(n-2);
        memo[n] = res;
        return res;
    }
```

- F(n) = F(n-1) + F(n-2) with F(1) = 1, F(2) = 2

LC DP#2: 322. Coin change.

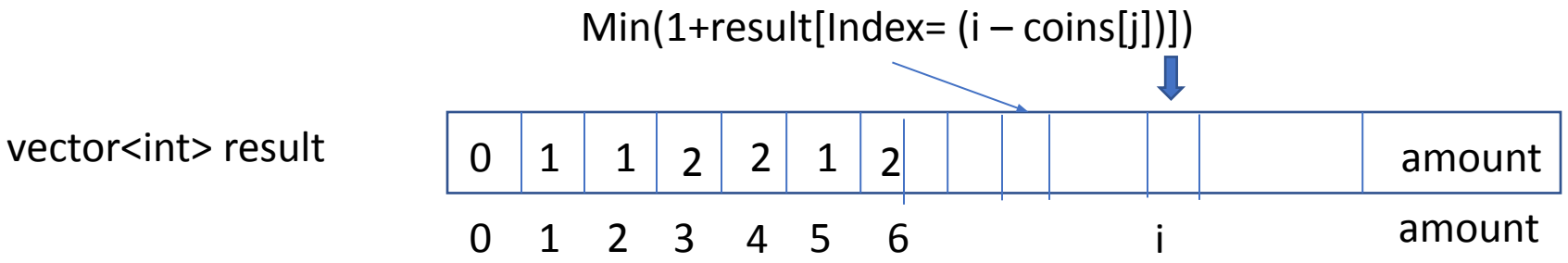
```
class Solution {
    public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> result(amount+1, INT_MAX);
        result[0] = 0;
        for (int i = 1; i <= amount; i++) {
            int ans = INT_MAX-1;
            for (int j = 0; j < coins.size(); j++) {
                int index = i - coins[j];
                if (index >= 0) {
                    ans = min(ans, 1 + result[index]);
                }
            }
            result[i] = ans;
        }
        return result[amount] >= INT_MAX-1? -1: result[amount];
    }
```

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

- for each amount, check for each coin
- since coin is always positive, if substract coin[i] is positive, we use it as index to get a result from that amount, plus 1 (current coin)
- we do for all coins to find the mini number

- Same nature of problem is perfect square LC#279, which is number of [1, 4, 9, 16…]



LC DP#3: 300. Longest Increasing Subsequence(LIS) of given array.

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> res(nums.size(), 1);

        for (int i = 1; i < nums.size(); i++) {
            int ans = 1;
            for (int j = 0; j <= i; j++) { // going thru all values before me,
                if (nums[i] > nums[j]) { // for all values I'm greater
                    int t = res[j] + 1;
                    ans = max(ans, t);   // keep max number
                }
            }
            res[i] = ans;
        }
        int ret = 0;
        for (int i = 0; i < nums.size(); i++) ret = max(ret, res[i]);
        return ret;
    }
};
```

- Start from beginning, for each position, check all values before me
- If it is smaller than me, take its result +1
- Keep max value as result
- If it is Longest Decreasing Subsequence: starting from the end.

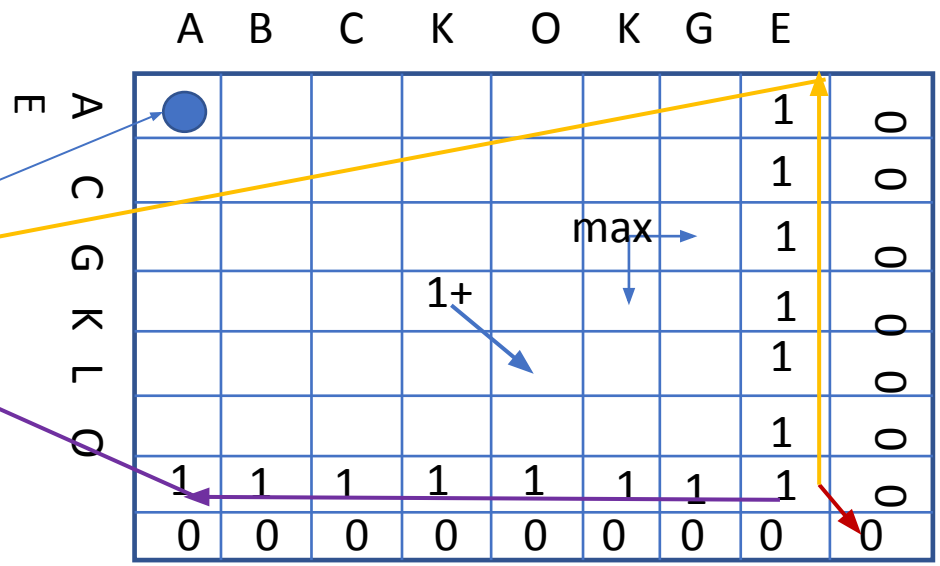
LC DP#4: 1143. Longest Common Sequence(LCS) of two strings.

```cpp
class Solution {
public:
    unordered_map<string, int> dp;

#ifdef RECURSION
    int longestCommonSubsequence(string text1, string text2) {
        if (text1.length() == 0 || text2.length() == 0) return 0;
        if (dp.find(text1+ "-" + text2) != dp.end()) {
            return dp[text1 + "-" + text2];
        }
        int res = 0;

        string t1 = text1.substr(1);
        string t2 = text2.substr(1);
        if (text1.at(0) == text2.at(0)) {
            res = 1 + longestCommonSubsequence(t1, t2);
        } else {
            int lcs1 = longestCommonSubsequence(text1, t2);
            int lcs2 = longestCommonSubsequence(t1, text2);
            res = max(lcs1, lcs2);
        }
        dp[text1 + "-" + text2] = res;
        return res;
    }
#endif
    int longestCommonSubsequence(string text1, string text2) {
        vector<vector<int>> dp(text1.length()+1, vector<int>(text2.length()+1,0));
        int r = text1.length()-1;
        int c = text2.length()-1;

        while (r >= 0 && c >= 0) {
            for (int c_ = c; c_ >= 0; c_--) {
                if (text2.at(c_) == text1.at(r))
                    dp[r][c_] = 1 + dp[r+1][c_+1];
                else
                    dp[r][c_] = max(dp[r+1][c_], dp[r][c_+1]);
            }
            for (int r_ = r; r_ >= 0; r_--) {
                if (text2.at(c) == text1.at(r_))
                    dp[r_][c] = 1 + dp[r_+1][c+1];
                else
                    dp[r_][c] = max(dp[r_+1][c], dp[r_][c+1]);
            }
            r--; c--;
        }
        return dp[0][0];
    }
};
```

- Use recursion: straightforward, but it takes much more time O(2 power min(N, M))
- Basically below plus memo:

```
if (text1.at(0) == text2.at(0))
   return 1 + LCS(text1.substr(1), text2.substr(1);
else
   return (max(LCS(text1, text2.substr(1)),
                LCS(text1.substr(1), text2)));
```

- Use dp approach: it is O(m*n)



LC DP#5: 139. word break: given sentence and word dictionary, tell if it can be break using word from dictionary.

- std::string::find(substr, pos): find first occurrence substr starting index of pos
- std::string::rfind(substr, pos): find last occurrence substr starting from pos
- Std::string::find_first_of(char c, int pos)

```
1  class Solution {
2  public:
3      unordered_map<string, bool> dp;
4
5      bool wordBreak(string s, vector<string>& wordDict) {
6          if (s.length() == 0) return true;
7          if (dp.find(s) != dp.end()) return dp[s];
8
9          // check if any matched word in dict,
10         for (int i= 0; i < wordDict.size(); i++) {
11             string word=wordDict[i];
12             if (s.find(word, 0) == 0) {
13                 if(wordBreak(s.substr(word.length()), wordDict)) {
14                     dp[s] = true;
15                     return true;
16                 }
17             }
18         }
19         dp[s] = false;
20         return false;
21     }
22 };
```

- Go thru given dictionary, for each word, use s.find(word, 0) to see if given string starts with word, if yes, recursive call of substring (after remove word)
- Use memo to improve performance

Backtracking recipe

```
void Backtrack(res, args)
    if ( GOAL REACHED )
        add solution to res
        return
    for ( int i = 0; i < NB_CHOICES; i++ )
        if ( CHOICES[i] is valid )
            make choices[i]
            Backtrack(res, args)
            undo choices[i]
```

LC DP#6.0 permutation

```
3       // https://www.youtube.com/watch?v=Nabbpl7y4Lo
4       // three steps of backtrack:
5       // #1 check if reaches goal, if yes, save result and return
6       // #2 among all possible choices
7       //     - take valid choice, update in input list
8       //     -- perform bacKtrack
9       //     - undo the choice, go for next choice
10      //
11      void backtrack(vector<int>& nums, vector<int>& perm, vector<vector<int>>& res, vector<bool>& used) {
12          if (perm.size() == nums.size()) { // check if meets goal
13              res.push_back(perm);           // yes, save result
14              return;
15          }
16          for (int i = 0; i < nums.size(); i++) { // all possible choices
17              if (!used[i]) {         // only take valid one
18                  used[i] = true;        // make the choice
19                  perm.push_back(nums[i]);  // update parameters
20                  backtrack(nums, perm, res, used);
21                  perm.pop_back();      // undo the choice
22                  used[i] = false;      // undo the choice
23              }
24          }
25
26      }
27      vector<vector<int>> permute(vector<int>& nums) {
28          vector<vector<int>> res;
29          vector<int> perm;
30          vector<bool> used(nums.size(), false);
31          backtrack(nums, perm, res, used);
32          return res;
33      }
34  };
```

LC DP#6: 39. combination sum I

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all* ***unique combinations*** *of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

```
1  class Solution {
2  public:
3      // reference: https://www.youtube.com/watch?v=yFfv03AE_vA
4      void backTrack(vector<int>& candidates, int start, vector<int> list, vector<vector<int>>& result, int target) {
5          if (target < 0) return;
6          if (target == 0) {
7              if (list.size() > 0) {
8                  vector<int> res1(list);
9                  result.push_back(res1);
10             }
11             return;
12         }
13         for (int i = start; i < candidates.size(); i++) {// start is used to prevent duplicate
14             list.push_back(candidates[i]);
15             backTrack(candidates, i, list, result, target-candidates[i]);
16             list.pop_back();
17         }
18     }
19
20     vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
21         vector<vector<int>> result;
22         vector<int> list;
23
24         backTrack(candidates, 0, list, result, target);
25
26         return result;
27     }
28 };
```

Allow same number to be re-used

- Numbers are distinct
- backtrack allow same number to be re-used

LC DP#6: 40. combination sum I I

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target` .

Each number in `candidates` may only be used **once** in the combination.

```
1  class Solution {
2  public:
3      void backTrack(vector<int>& candidates, int start, int target, vector<int> list, vector<vector<int>>& result) {
4          if (target < 0) return;
5          if (target == 0) {
6              vector<int> ans(list);
7              result.push_back(ans);
8              return;
9          }
10         for (int i = start; i < candidates.size(); i++) {
11             // this makes sure number is used only once
12             if (i > start && candidates[i] == candidates[i-1]) continue;
13             if (target-candidates[i] < 0) break; // since it is sorted
14             list.push_back(candidates[i]);
15             backTrack(candidates, i+1, target-candidates[i], list, result); // i+1 to avoid re-use the same number
16             list.pop_back();
17         }
18     }
19
20     vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
21         vector<vector<int>> result;
22         vector<int> list;
23         sort(candidates.begin(), candidates.end()); // sort in ascending order
24         backTrack(candidates, 0, target, list, result);
25         return result;
26     }
27 };
```

- Values are random, could duplicate
- Sort them in ascending order
- each occurrence of number is used only once
- **Skip the following numbers with same value**

LC DP#6: 216. combination sum III

Find all valid combinations of `k` numbers that sum up to `n` such that the following conditions are true:

- Only numbers `1` through `9` are used.
- Each number is used **at most once**.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

```
class Solution {
public:
    void backtrack(int k, int start, int target, vector<int> list, vector<vector<int>> &result) {
        if (target < 0) return;
        if (target ==0) {
            if (list.size() == k) { // result only allow K numbers
                vector<int> ans(list);
                result.push_back(ans);
            }
            return;
        }
        if (list.size() >= k) return; // too many stop here
        for (int i = start; i < 10; i++) { // only use the digits 1-9
            list.push_back(i);
            backtrack(k, i+1, target-i, list, result);
            list.pop_back();
        }
    }

    vector<vector<int>> combinationSum3(int k, int n) {
        vector<vector<int>> result;
        vector<int> list;
        backtrack(k, 1, n, list, result);
        return result;
    }
};
```

- simpliedfied version of II

LC DP#6: 377. combination sum IV

Given an array of **distinct** integers `nums` and a target integer `target`, return *the number of possible combinations that add up to* `target`.

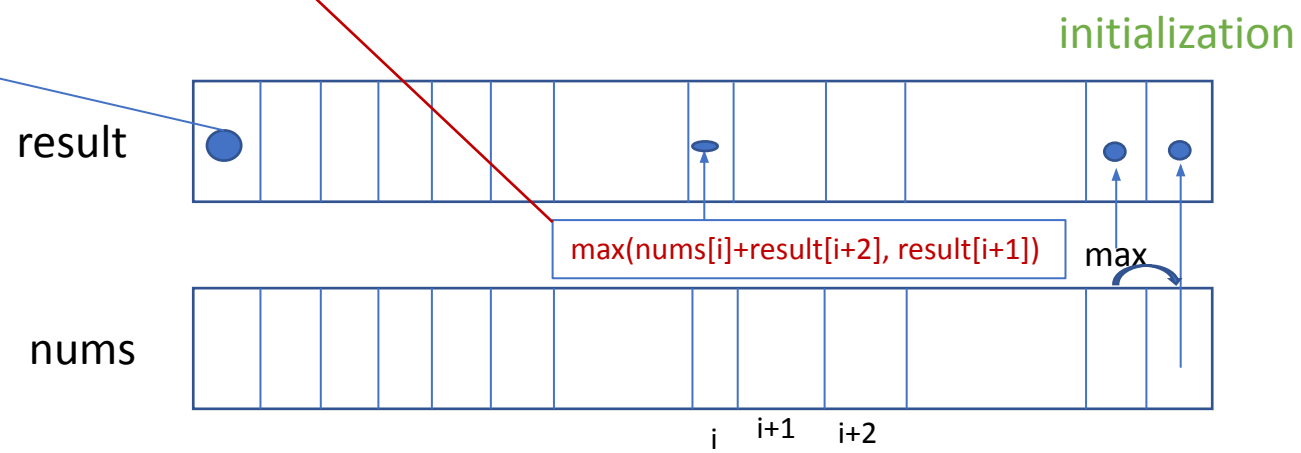
```
class Solution {
public:
    unordered_map<string, int> memo;
    int backtrack(vector<int>& nums, int pos, int target) {
        int res = 0;
        if (target < 0) return 0;
        if (target == 0) {
            return 1;
        }
        string key = to_string(pos) + "---" + to_string(target);
        if (memo.find(key) != memo.end()) return memo[key];

        for (int i = pos; i < nums.size(); i++) {
            if (target - nums[i] < 0) break;
            res += backtrack(nums, pos, target - nums[i]);
        }
        memo[key] = res;
        return res;
    }

    int combinationSum4(vector<int>& nums, int target) {
        int result = 0;
        sort(nums.begin(), nums.end());
        result = backtrack(nums, 0, target);
        return result;
    }
};
```

- backtrack with memo of pos+target

LC DP#7: 198. Rob house: Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 1) return nums[0];

        vector<int> result(nums.size(), 0);
        result[nums.size()-1] = nums[nums.size()-1];
        result[nums.size()-2] = max(nums[nums.size()-1],nums[nums.size()-2]) ;
        for (int i = nums.size()-3; i >= 0; i--) {
            result[i] = max(result[i+2]+nums[i], result[i+1]);
        }
        return result[0];
    }
};
```

- Since we don't know what is ahead to make it maximum, we start from the end: last one is it self, last second one is if it is greater last, then it is self, otherwise would be last one
- For any given house position "index", compare "its value + next_next" with "next", take the bigger value



LC DP#8: 91. decode way. 226 => BBF, BZ, VF

A message containing letters from `A-Z` can be **encoded** into numbers using the following mapping:

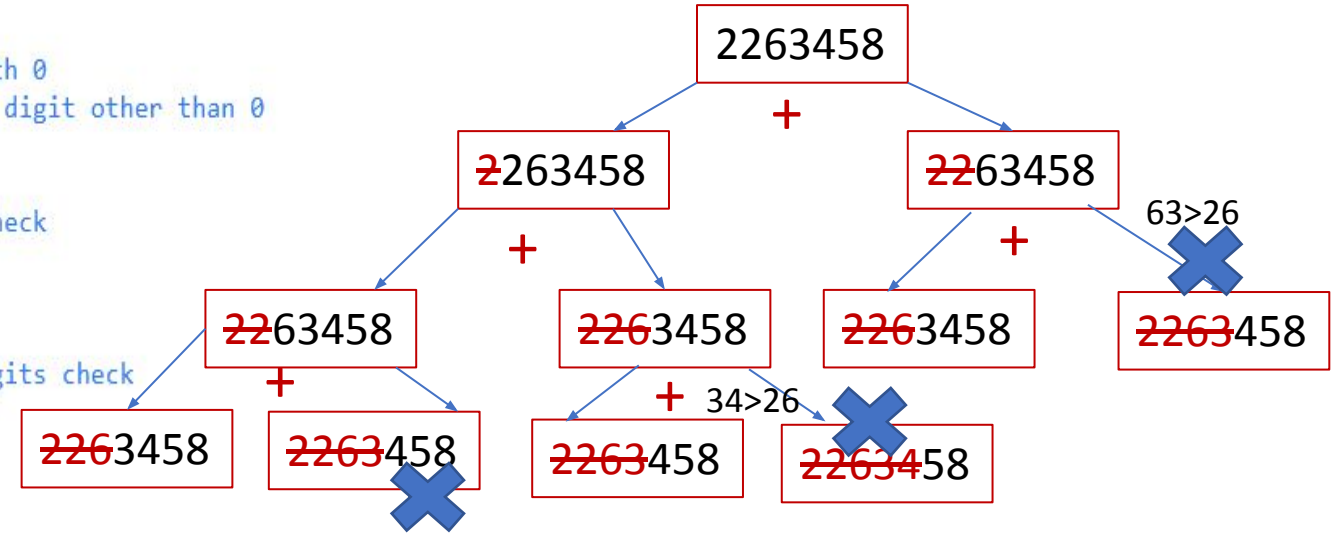
```
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
```

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, `"11106"` can be mapped into:

- `"AAJF"` with the grouping `(1 1 10 6)`
- `"KJF"` with the grouping `(11 10 6)`

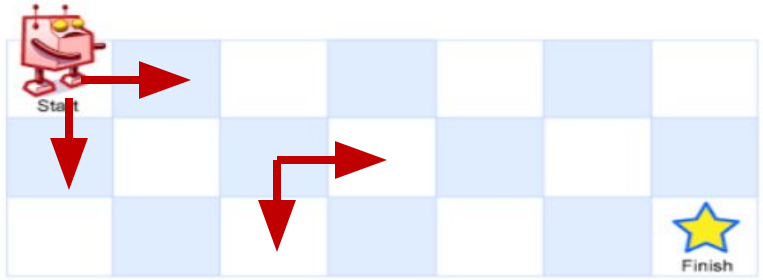
```
3  class Solution {
4  private :
5  public:
6      unordered_map<string, int> memo;
7
8      int numDecodings(string s) {
9          if (s.length() == 0) return 1;
10         if (s.at(0) == '0') return 0; // can' decode starting with 0
11         if (s.length() == 1) return 1; // good if it is only one digit other than 0
12         if (memo.find(s) != memo.end()) return memo[s];
13
14         int res = numDecodings(s.substr(1)); // spin one digit check
15         if (s.length()>=2) {
16             int val = stoi(s.substr(0,2));
17             if (val <=26) {
18                 res += numDecodings(s.substr(2)); // spin two digits check
19             }
20         }
21         memo[s] = res;
22         return res;
23     }
24 };
25
```

- Check special cases:
  - end of string: good (return 1)
  - Starts with 0, can't decode
  - One digit other than 0, good (return 1)
- Check memo cache, if it exists, return result
- Otherwise try to explore with one digit and two digits if possible



LC DP#9: 62. unique paths/grid walk

Example 1:



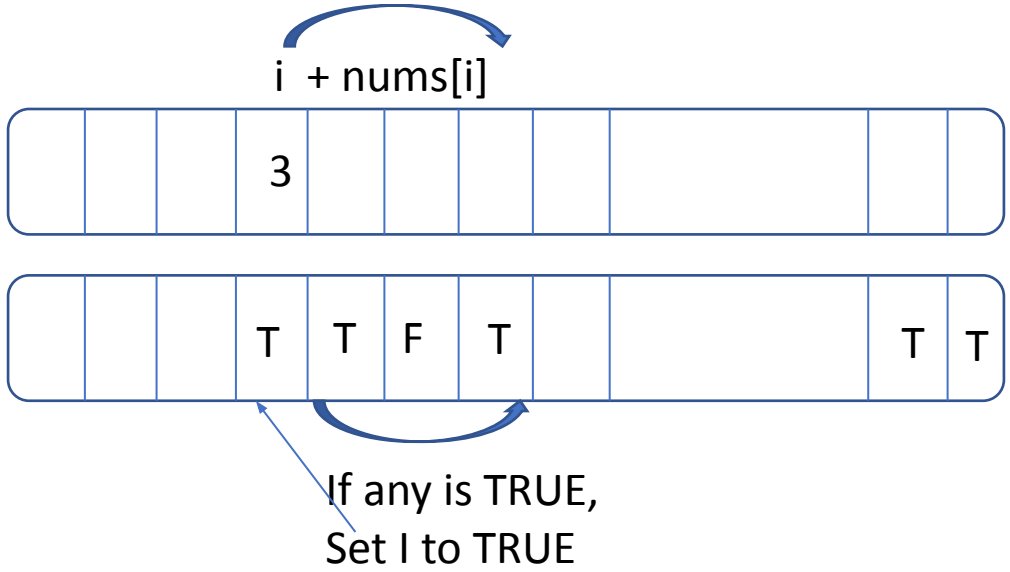

```
Input: m = 3, n = 7
Output: 28
```

```
1  class Solution {
2  private:
3          unordered_map<string, int> memo;
4  public:
5
6      int uniquePaths(int m, int n) {
7
8          if (m < 1 || n < 1) return 0;
9          if (m == 1 || n == 1) return 1; // walk bottom or right
10         string key = to_string(m) + "---" + to_string(n);
11         if (memo.find(key) != memo.end()) {
12             return memo[key];
13         }
14         int res = uniquePaths(m-1, n) + uniquePaths(m, n-1);
15         memo[key] = res;
16
17         return res;
18     }
19 };
```

- Res [m,n] = res[m-1, n] + res[m, n-1]
- Use a unordered_map<string, int> as memo
- Key is " "r"==="c"

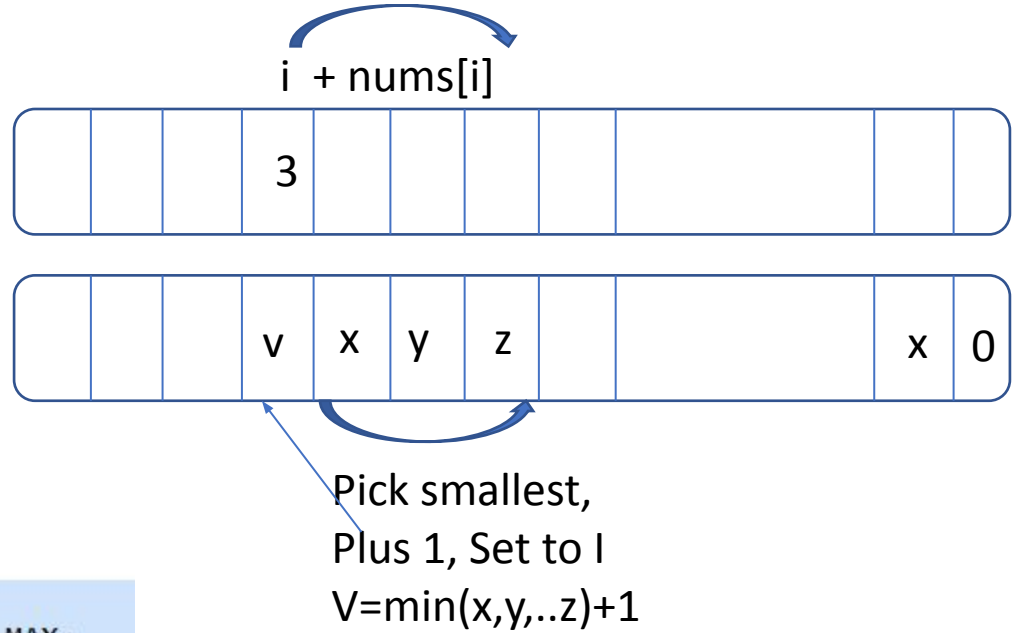
LC DP#10: 55. Jump Game.

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int N = nums.size();
        vector<bool> status(N, false);

        status[N-1] = true; // last stop is true
        for (int i = N - 2; i >= 0; i--) {
            int furestJump = min(i + nums[i], N-1);
            for (int j = i+1; j <= furestJump; j++) {
                if (status[j] == true) { // within reach, if any is good
                    status[i] = true;  // we are good.
                    break;
                }
            }
        }
        return status[0];
    }
};
```



LC DP#11: 45. Jump Game II.

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int N = nums.size();
        vector<int> jumps(N);
        jumps[N-1] = 0;

        for (int i = N-2; i >= 0; i--) {
            int maxPos = min(N-1, i+nums[i]);
            int ans = INT_MAX;
            for (int j=i+1; j <= maxPos; j++) {
                int tmp = jumps[j] != INT_MAX ? (jumps[j] + 1):INT_MAX;
                ans = min(ans, tmp);
            }
            jumps[i] = ans;
        }
        return jumps[0];
    }
};
```



LC DP#12: 1306. Jump Game III

Given an array of non-negative integers `arr`, you are initially positioned at `start` index of the array. When you are at index `i`, you can jump to `i + arr[i]` or `i - arr[i]`, check if you can reach to **any** index with value 0.

```cpp
class Solution {
public:
    bool helper(vector<int>& arr, int start, vector<bool>& v) {
        if (arr[start] == 0) return true; // reach desired place
        if (v[start] == true) return false; // found loop

        v[start] = true; // mark this is visited
        vector<bool> nv = v;
        if (start + arr[start] < arr.size()) { // try next round
            if (helper(arr, start + arr[start], nv)) return true;
        }
        if (start - arr[start] >= 0) {
            if (helper(arr, start - arr[start], nv)) return true;
        }
        return false;
    }

    bool canReach(vector<int>& arr, int start) {
        vector<bool> v(arr.size(), false);

        return helper(arr, start, v);
    }
};
```

- Recursive call forward and backward if it inbound
- Use a visited map, if reach visited pos, loop detected, NOT possible

Example 1:

Input: arr = [4,2,3,0,3,1,2], start = 5

Output: true

Explanation:

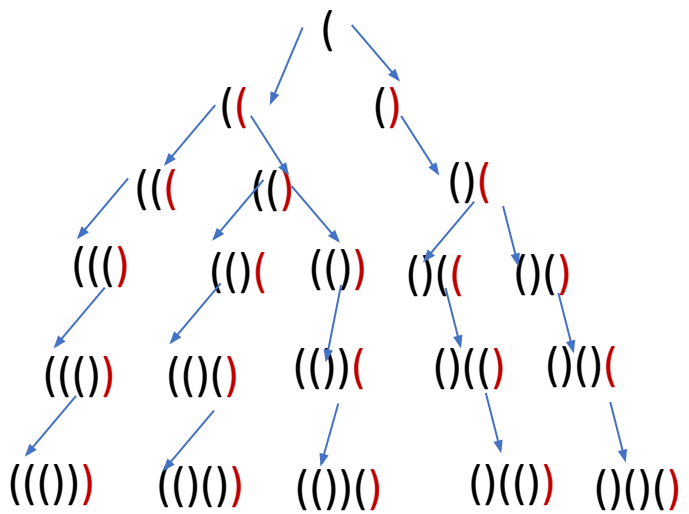
All possible ways to reach at index 3 with value 0 are:

index 5 -> index 4 -> index 1 -> index 3

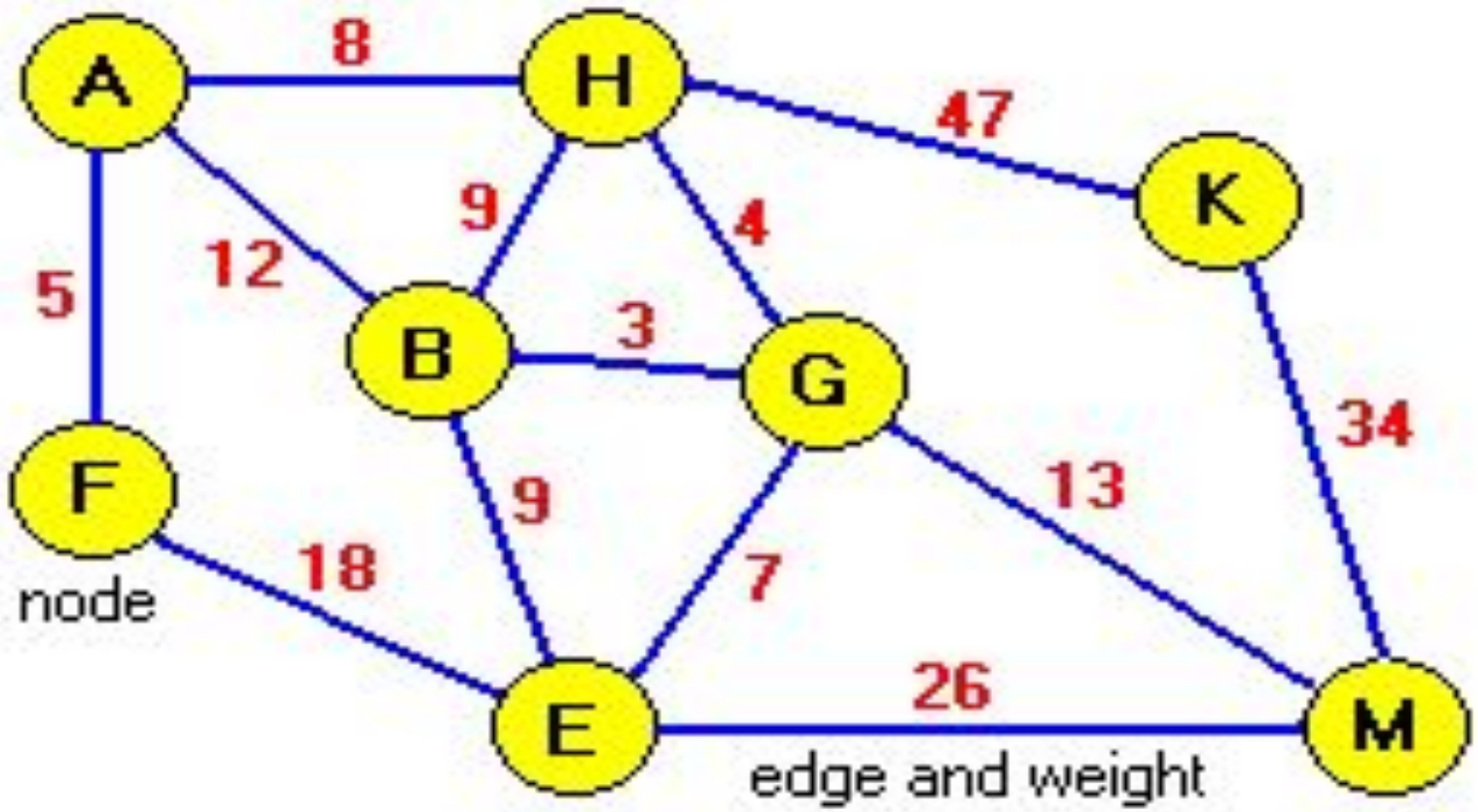
index 5 -> index 6 -> index 4 -> index 1 -> index 3

LC DP#13: 22: Generate parentheses

```
class Solution {
public:
    int N;
    void backTrack(vector<string>& res, vector<char> cur, int open, int close) {
        if (cur.size() == 2*N) {
            string ans = "";
            for (int i = 0; i < 2*N; i++) ans.append(1,cur[i]);
            res.push_back(ans);
            return;
        }
        if (open < N) {
            cur.push_back('(');
            backTrack(res, cur, open+1, close);
            cur.pop_back();
        }
        if (open > close) {
           cur.push_back(')');
           backTrack(res, cur, open, close+1);
           cur.pop_back();
        }
    }
    vector<string> generateParenthesis(int n) {
        vector<string> res;
        vector<char> cur;
        N = n;
        backTrack(res, cur, 0, 0);
        return res;
    }
};
```



Graph



LC Graph#1: 133. clone graph.

```
class Solution {
public:
    Node* cloneGraph(Node* node) {
        Node *root = NULL;
        queue<Node *> q;
        unordered_map<int, Node *> newNodes;
        unordered_set<int> seen;
        if (node == NULL) return NULL;

        q.push(node);

        while (!q.empty()) {
            Node *t = q.front();
            q.pop();
            Node *n = NULL;
            if (newNodes.find(t->val) == newNodes.end()) {
                n = new Node(t->val);
                if (root == NULL) root = n;
                newNodes[t->val] = n;
            } else {
                n = newNodes[t->val];
            }

            for (int i = 0; i < t->neighbors.size(); i++) {
                Node *nei = t->neighbors[i];
                Node *nnei = NULL;
                if (newNodes.find(nei->val) == newNodes.end()) {
                    nnei = new Node(nei->val);
                    newNodes[nnei->val] = nnei;
                } else {
                    nnei = newNodes[nei->val];
                }
                if (find(n->neighbors.begin(), n->neighbors.end(), nnei) == n->neighbors.end())
                    n->neighbors.push_back(nnei);
                if (seen.find(nei->val) == seen.end())
                    q.push(nei);
            }
            seen.insert(n->val);
        }
        return root;
    }
};
```

```
class Node {
public:
    int val;
    vector<Node*> neighbors;
    Node() {
        val = 0;
        neighbors = vector<Node*>();
    }
    Node(int _val) {
        val = _val;
        neighbors = vector<Node*>();
    }
```

- Use BFS (i.e. use Q) to traversal Graph with help of unordered_set<int>
- Create a unordered_map<int, Node*> to track all created cloned Node, key is node#
- When traverse a node,
  - check cloned node in map, if it doesn't exist, create it.
  - find its all neighbor nodes, check if its cloned node is in map, if not created,
  - Add all cloned neighbors under cloned node

LC Graph#2: 207. Course Schedule.

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [a_i, b_i]` indicates that you **must** take course `b_i` first if you want to take course `a_i`. Return `true` if you can finish all courses. Otherwise, return `false`.

```
class Solution {
    const int INIT = 0;
    const int DONE = 1;
    const int PROCESSING = 2;
    unordered_map<int, vector<int>> preq_list; // essentially adjancy list of graph
    vector<int> state;
public:
    bool hasCycle(int course) {
        if (state[course] == PROCESSING) return true; // cycle detected!
        state[course] = PROCESSING;
        if (preq_list.find(course) != preq_list.end()) {
            vector<int> preqs = preq_list[course];
            for (int j = 0; j < preqs.size(); j++) {
                if (state[preqs[j]] != DONE) {
                    if (hasCycle(preqs[j]))
                        return true;
                }
            }
        }
        state[course] = DONE;
        return false;
    }
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        // go thru to establish preq for each course (node)
        for (int i = 0; i < prerequisites.size(); i++) {
            vector<int> elem = prerequisites[i];
            int course = elem[0];
            int pre = elem[1];
            if (preq_list.find(course) != preq_list.end()) {
                vector<int> courses = preq_list[course];
                courses.push_back(pre);
                preq_list[course] = courses; // update
            } else {
                vector<int> preq { pre }; //preq.push_back(pre);
                preq_list[course] = preq;
            }
        }
        // now go thru each course, to make sure there is no loop in the preq_list
        state = vector<int>(numCourses, INIT); // essentially it creates an array, init to zero
        for (int i = 0; i < numCourses; i++) {
            if ( state[i] == INIT) {  // if this course was never initiated, start detect
                if (hasCycle(i))
                    return false;
            }
        }
        return true;
    }
};
```

- Go thru given pre-requisites to establish preq_list unordered_map<int, vector<int>>: key is course#, value is its pre_requsite vector<int>
- Init state for each course to INIT
- Go thru each course, using GRAPH COLOR to detect if there is any cycle.

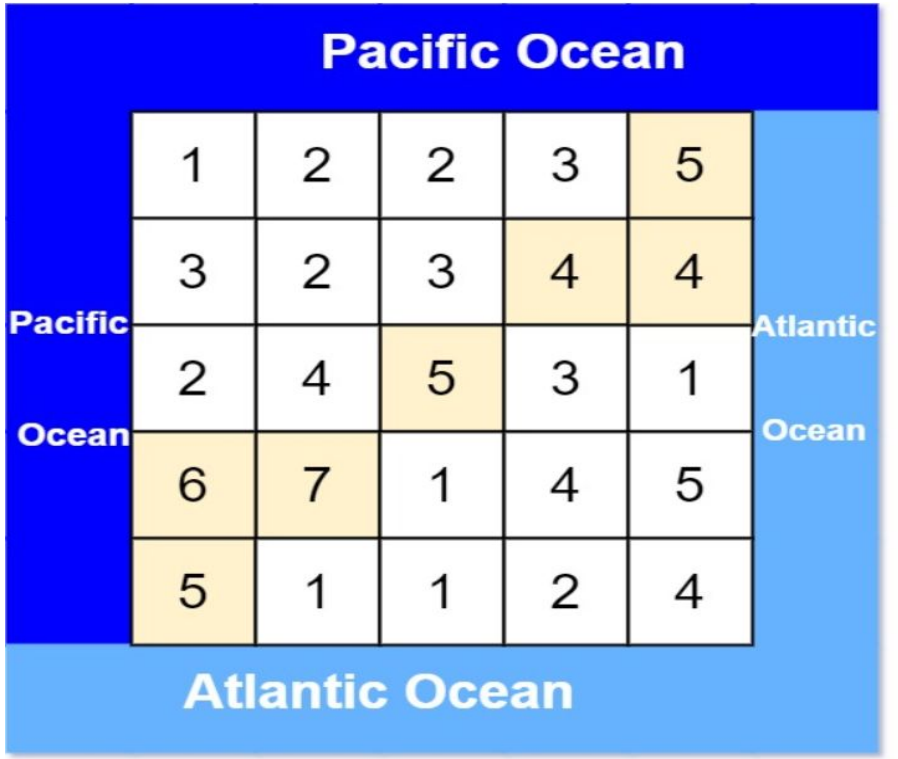
LC Graph#3: 210. Course Schedule II (topological sort).

```cpp
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        vector<int> res;
        vector<vector<int>> preq_list(numCourses);
        // establish indegree for each course: number of pre-requsites
        vector<int> indegrees(numCourses, 0);
        for (int i = 0; i < prerequisites.size(); i++) {
            int target = prerequisites[i][0];
            int condition = prerequisites[i][1];
            preq_list[condition].push_back(target); // collect who requires/needs me
            indegrees[target] +=1; // indegree table for each course
        }

        // go thru indegrees table to find initial courses with zero indegree
        queue<int> q;
        unordered_set<int> visited;
        for (int i = 0; i < numCourses; i++) {
            if (indegrees[i] == 0) {
                q.push(i);
                visited.insert(i);
            }
        }
        while (!q.empty()) {
            int c = q.front(); q.pop();
            res.push_back(c);

            // find who needs me, decrease indegrees by 1 ,
            vector<int> targets = preq_list[c];
            for (int j = 0; j < targets.size(); j++) {
                indegrees[targets[j]] -=1;
            }
            // and add the member with zero into q if it is not seen before
            for (int i = 0; i < numCourses; i++) {
                if (indegrees[i] == 0 && visited.find(i) == visited.end()) {
                    q.push(i);
                    visited.insert(i);
                }
            }
        }
        if (res.size() != numCourses) res.clear();
        return res;
    }
};
```

- Go thru given pre-requisites list to establish indegree[target] table and preq_list<cond, targets> (for a given course, all other courses require/need it)
- Add all course with 0 in indegree table into Q
- For each course in Q,
  - dequeue it,
  - Add it into result list
  - decrease 1 in indegree table for all other courses which require it.
  - Add any new courses with 0 indegree.
- Repeat until Q is empty

LC Graph#4: 417. Pacific Atlantic Water Flow.

```cpp
class Solution {
    const int PAC = 0x1;
    const int ATL = 0x2;

public:
    void dfs(int r, int c, vector<vector<int>>& h, int flag, vector<vector<int>>& map) {
        vector<vector<int>> DIRS;
        vector<int> up {-1, 0}, down {1,0}, left{0,-1}, right{0,1};
        DIRS.push_back(up); DIRS.push_back(down); DIRS.push_back(left); DIRS.push_back(right);
        int R = h.size();
        int C = h[0].size();
        map[r][c] |= flag;
        // DFS explore thru 4 directions
        for (auto d: DIRS) {
            int nr = r + d[0];
            int nc = c + d[1];
            if (nr < 0 || nr >= R || nc < 0 || nc >= C ) continue; // out of space
            if ((map[nr][nc] & flag) == flag) continue; // reached already
            if (h[nr][nc] >= h[r][c]) dfs(nr, nc, h, flag, map); // if neighbor is not lower, explore
        }
    }
    vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
        int R = heights.size();
        int C = heights[0].size();
        vector<vector<int>> map(R, vector<int>(C,0));

        // do DFS traversal starting from known cells:
        // pacific: top and left
        // altalantic: bottolw and right
        for (int r = 0; r < R; r++) {
            dfs(r, 0, heights, PAC, map); // left side
            dfs(r, C-1, heights, ATL, map); // right side
        }
        for (int c = 0; c < C; c++) {
            dfs(0, c, heights, PAC, map); // top side
            dfs(R-1, c, heights, ATL, map); // bottom side
        }
        vector<vector<int>> res;
        for (int r =0; r < R; r++) {
            for (int c =0; c < C; c++) {
                if (map[r][c] == (PAC|ATL)) {
                    vector<int> ans {r,c};
                    res.push_back(ans);
                }
            }
        }
        return res;
    }
};
```



Input: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

- Use DFS find all cells (4 directions) which have equal or higher HEIGHT
- Pacific starts top & left cells; Atlantic starts bottom and right cells
- Use a vector<vector<int>> map to record DFS result
- Check each cell result in map, if both PAC and ATL set, add into result.

LC Graph#5: 200. Number of Islands.

```
class Solution {
public:
    int R, C;
    void dfs(int r, int c, vector<vector<char>>& grid) {
        vector<vector<int>> DIRS { {-1,0}, {1,0}, {0,-1}, {0,1}};
        grid[r][c] = 0; //visit and zero out
        for (auto d: DIRS) {
            int nr = r + d[0];
            int nc = c + d[1];
            if (nr < 0 || nc < 0 || nr >= R || nc >= C) continue;
            if (grid[nr][nc] == '1') dfs(nr, nc, grid);
        }
    }

    int numIslands(vector<vector<char>>& grid) {
        R = grid.size();
        C = grid[0].size();
        int res = 0;

        for (int r = 0; r < R; r++) {
            for (int c = 0; c < C; c++) {
                if (grid[r][c]=='1') {
                    res++;
                    dfs(r, c, grid);
                }
            }
        }
        return res;
    }
};
```

- Search thru grid, find 1,
- Increment result
- Perform DFS to zero out all neighbors(UP, BOTTOM, LEFT, RIGHT)
- Repeat above

LC Graph#6: 128. Longest Consecutive Sequence. Must be O(n)

```cpp
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        if (nums.size() == 0) return 0;
        unordered_set<int> allNums(nums.begin(), nums.end());
        unordered_map<int, bool> checked; // to ensure O(n) check
        int res = 1;
        for (int i = 1; i < nums.size(); i++) {
            int n = nums[i];
            if (checked.find(n) == checked.end()) {
                int counter = 1;
                int target = n -1; // search downwards
                while (allNums.find(target) != allNums.end()) {
                    counter++;
                    checked[target] = true;
                    target--;
                }
                target = n+1; // search upwords
                while (allNums.find(target) != allNums.end()) {
                    counter++;
                    checked[target] = true;
                    target++;
                }
                res = max(counter, res);
                checked[n] = true;
            }
        }
        return res;
    }
};
```

- Add all numbers into a set
- Go thru each number, search downward and upwards consecutively
- Counter all of them
- Mark them checked in HashMap
- Return max counter

LC Graph#7: 261. Graph Valid Tree: i.e. all connected without loop.

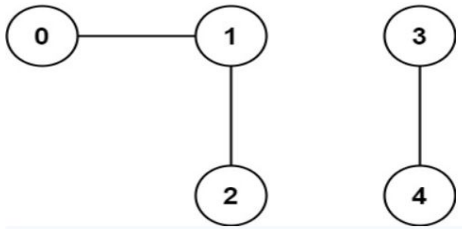
```cpp
class Solution {
public:
    bool validTree(int n, vector<vector<int>>& edges) {
        // establish adjList
        vector<vector<int>> adj(n);
        for (int i= 0; i < edges.size(); i++) {
            int a = edges[i][0]; // use a, b make more sense since this is undirected graph
            int b = edges[i][1];
            adj[a].push_back(b); // every edge, we have two edges save, need remove
            adj[b].push_back(a); // one edge during traverse the first one
        }
        unordered_set<int> seen; // save the traversed nodes as result
        stack<int> st;
        st.push(0); // start with first node
        seen.insert(0);
        while (!st.empty()) {
            int n = st.top(); st.pop();
            for (auto nei: adj[n]) {
                if (seen.find(nei) != seen.end()) {
                    return false; // there is cycle
                }
                seen.insert(nei);
                st.push(nei);

                // remove edge from nei->n since this is undirected graph
                vector<int> nodes_2_nei = adj[nei];
                nodes_2_nei.erase(find(nodes_2_nei.begin(), nodes_2_nei.end(), n));
                adj[nei] = nodes_2_nei; // need update back!!!
            }
        }

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

- A graph is a valid tree: all nodes are connected without loop
- Establish adjancey list for each node by going thru edge list: by directions
- DFS using STACK: Starts with first node, use a “seen” set to track visited node.
- While add each neighbor into stack, remove current node from neighbor’s neighbor list!

LC Graph#8: 323. Number of Connected Components in an Undirected Graph.

```
class Solution {
    vector<vector<int>> G;
    vector<bool> visited;
public:
    void dfs(int i) {
        visited[i] = true;
        for (int j= 0; j < G.size(); j++) {
            if (G[i][j] && visited[j] == false) dfs(j);
        }
    }

    int countComponents(int n, vector<vector<int>>& edges) {
        visited = vector<bool>(n, false);
        G = vector<vector<int>>(n, vector<int>(n,0));
        for (int i = 0; i < edges.size(); i++) {
            G[edges[i][0]][edges[i][1]] = 1;
            G[edges[i][1]][edges[i][0]] = 1;
        }

        int res = 0;
        for (int i = 0; i < n; i++) {
            if (visited[i] == false) {
                dfs(i);
                res++;
            }
        }

        return res;
    }
};
```

Example 1:



Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2

- Transform edges into G
- Use vector<bool> visited(n, false) to track if node is visited by dfs
- Start dfs with node 0, every return of dfs, increment res++;

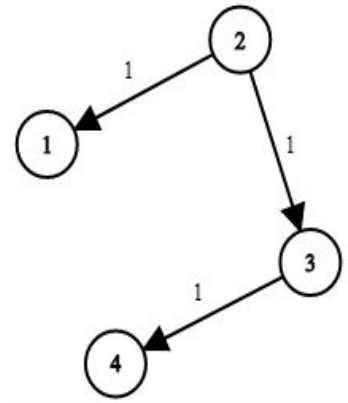
LC Graph#9:

743: network delay time.

You are given a network of `n` nodes, labeled from `1` to `n`. You are also given `times`, a list of travel times as directed edges `times[i] = (u_i, v_i, w_i)`, where `u_i` is the source node, `v_i` is the target node, and `w_i` is the time it takes for a signal to travel from source to target.

We will send a signal from a given node `k`. Return the time it takes for all the `n` nodes to receive the signal. If it is impossible for all the `n` nodes to receive the signal, return `-1`.

Example 1:




```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2
```

```cpp
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        // establish G and adj list for each node
        int N = n + 1;
        vector<vector<int>> G(N, vector<int>(N,0));
        vector<vector<int>> adj(N); // holds neighbors for easy access
        for (int i = 0; i < times.size(); i++) {
            int u = times[i][0];
            int v = times[i][1];
            int w = times[i][2];
            G[u][v] = w;
            vector<int> neis1 = adj[u];
            neis1.push_back(v);
            adj[u] = neis1;
        }

        vector<int> time(n+1, INT_MAX) ; // hold the final result
        vector<bool> visited(n+1, false);
        visited[0] = true; // since we are not using 0 position, exlcude it
        time[k] = 0; // set the given node K as starting point

        while (1) {
            // find the node u with mini time from not visited nodes
            // make u visited
            int u = 0;
            int minTime = INT_MAX;
            for (int i = 1; i < N; i++) {
                if (visited[i] == false && time[i] < minTime) {
                    u = i; minTime = time[i];
                }
            }
            if (minTime == INT_MAX) break; // all nodes are covered
            visited[u] = true;
            // find all neighbors of u, named it v, and update time based on givem ti
            vector<int> u_neis = adj[u];
            for (int j = 0; j < u_neis.size(); j++) {
                int v = u_neis[j];
                if (visited[v] == false) {
                    int w = G[u][v];
                    time[v] = min(time[u] + w, time[v]);
                }
            }
        }
        int result = 0;
        for (int i =1; i < time.size(); i++)
            result = max(result,time[i]);

        return result == INT_MAX? -1:result;
    }
};
```

visited

T		F	T		F	F	F	F	F

time

			0				INF	INF	

k

- Among NOT visited nodes, find node with mini time as node u
- Find all neighbors of u, name as v, min(time[v], time[u]+w), make node u visited.
- Repeat above until all nodes are visited or TIME=INF(not reachable)

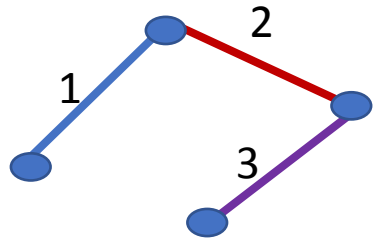
LC Graph#10: 787: cheapest flights within K stops.

- Key points

LC# 1135. Connecting Cities With Minimum Cost (Prims algorithm)

```cpp
class Solution {
public:
    // use prims algorithm:
    // slect the smallest edge, but makse sure one node is in node hashset, the other one is not
    // until N nodes in hashset
    static bool cmp(vector<int> c1, vector<int> c2) {
        return c1[2] < c2[2];
    }

    int minimumCost(int n, vector<vector<int>>& connections) {
        int res = 0;
        unordered_set<int> nodes;
        sort(connections.begin(), connections.end(), cmp); // nLogN
        //take the first minimum cost edge
        vector<int> first = connections[0];
        res = first[2];
        nodes.insert(first[0]);
        nodes.insert(first[1]);
        connections.erase(connections.begin());
        while (nodes.size() < n) {
            bool foundConnection = false;
            for (int i = 0; i < connections.size(); i++) { // N*N
                vector<int> c = connections[i];
                bool node1InSet = (nodes.find(c[0]) != nodes.end());
                bool node2InSet = (nodes.find(c[1]) != nodes.end());
                if ((node1InSet && !node2InSet) || (!node1InSet && node2InSet)) {
                    res += c[2];
                    if (!node1InSet) nodes.insert(c[0]);
                    if (!node2InSet) nodes.insert(c[1]);
                    connections.erase(connections.begin()+i);
                    foundConnection = true;
                    break;
                }
            }
            if (!foundConnection && (nodes.size() < n)) return -1;
        }
        return res;
    }
};
```

- Sort connection first
  - Add first mini cost connection, remove It from connections, add nodes into set
  - Find mini cost connection with only one node in set until all nodes are in.



Misc: LRU cache, LC#146

Tree Height, Prime number and Inorder Traversal

```cpp
int hight_dfs(Node* node, int &res) {
    if (node == NULL) return -1;
    if (node->children.size() == 0) return 0;

    vector<int> heights;
    for (auto n: node->children) {
        heights.push_back(hight_dfs(n, res));
    }
    sort(heights.begin(), heights.end(), greater<int>());
    if (heights.size() >= 2) {
        res = max(res, heights[0] + heights[1] + 2); // add two: one for each side
    }
    return heights[0]+1;
}
```

```cpp
class Solution {
public:
    int countPrimes(int n) {
        if (n <= 2) return 0;

        vector<bool> prime_nums(n, true);;
        for ( int p = 2; p*p < n; p++) {
            if (prime_nums[p]) {
                for (int i=p*p; i < n; i +=p)
                    prime_nums[i] = false;
            }
        }
        int res = 0;
        for (int i = 2; i < n; i++)
            if (prime_nums[i]) res++;

        return res;
    }
};
```

```cpp
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == NULL) return res;
#ifdef RECURSIVE
        vector<int> left = preorderTraversal(root->left);
        for (int i = 0; i < left.size(); i++)
            res.push_back(left[i]);
        res.push_back(root->val);
        vector<int> right = preorderTraversal(root->right);
        for (int i = 0; i < right.size(); i++)
            res.push_back(right[i]);
#endif
        TreeNode *cur = root;
        stack<TreeNode *> st;
        while (cur != NULL || !st.empty()) {
            if (cur != NULL) {
                st.push(cur);
                cur= cur->left; // push down all the way to bottom most left node
            } else {
                cur = st.top(); st.pop();
                res.push_back(cur->val);
                cur = cur->right;
            }

        }
        return res;
    }
```

From Excel

- 937s, 408s, 65h,242s, 49m, 1062m, 1092m,14m,12m,527h
- 752m(open lock, BFS), 56s, 80m, 88s, 238s, 16m, 41m, 128m, 35s,
- 69s, 34s, 153m, 162m(peak), 278m, 33m, 81m, 34m, 50m, 96m,
- 762h, 136s, 137m, 169s, 229s, 134m(gas), 179m, 402m, 55m, 45m,
- 31m,19s,21s,82s,83s,86s,206s,92s,61m,109m,
- 138m,141s,142m,143m,148m,538s,110s(balanced tree),102s,107s(Tree Level T),144s (Tree Preorder T),
- 104s,105m(binT from Pre & inroder),297m,98m,285m,510m,366m,156m(BT upDown),39m,207m(course schedule),
- 51m,52m,90m,78m,47m,46m, 63s, 70s, 53s,152m(max product subarray),
- 72m,115h,120m(trangle min sum),139m(word break, recur with memo), 140h (word break II, backtrack), 97m,91m, 128m(int array LongConSeq), 23m,
- 232m, 155m,263m,264m(ugly number), 212h, 79h, 295h,84h, 438s,311m,288m,
- 981m(time based key store), 706m(Design HashMap)

Java: HashMap and HashSet

```
HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
/* Add new entry */
map.put(1, 100); map.put(2, 200); if (map.size() == 2) System.out.println("Cool");

/* Get/check using key */
Integer val = map.get(1);
/* Loop thru key */ for (Integer key: map. keyset()) System.out.println("key is " + key.intValue());
/* Loop thru value */
for (Integer val: map.values()) System.out.println("val is " + val.intValue();

/* Iterator using Map.Entry*/
for (Map.Entry<Integer, Integer> ent: map.entrySet())
    System.out.println("Key is "+  ent.getKey() + ", value is " +  ent.getValue());

HashSet<String> set = new HashSet<String>();
set.add("xyz"); if (set.contains("abc")) System.out.println("abc is not found");
```

Java: Stack and Queue

```
Stack<Integer> st = new Stack<Integer>;
st.push(1); st.push(2); st.pop(); st.peek(); st.size();
while (!st.isEmpty()) System.out.println("pop: " + st.pop());

Queue<Integer> q = new LinkedList<Integer>();
q.add(1); q.add(2); q.add(3);
System.out.println("Q size is " + q.size());
while (!q.isEmpty()) System.out.println("Q remove : " + q.remove()); // remove() throw exp if empty
Integer val = q.poll(); //remove() method returns the head of the queue and removes it.
                        // It returns null if the queue is empty.
Vector<Integer> v = new Vector<Integer>(); // thread safe
v.add(1); v.add(2); Integer val = v.get(1);
v.remove(0); v.remove(new Integer(2));
for (int i = 0; i < v.size(); i++) System.out.print(v.get(i) + " ");

ArrayList<Integer> al = new ArrayList <Integer>(); // not synchronized, faster
al.add(1); al.add(2); Integer val = al.get(1);
al.remove(0); al.remove(new Integer(2));
for (int i = 0; i < al.size(); i++) System.out.print(al.get(i) + " ");
```

Java: util.Arrays

- Java.util.Arrays.binarySearch(int[] arr, int key):
- Java.util.Arrays.copyOf(int[] arr, int newLen)
- Java.util.Arrays.sort(int[] arr) or
  Java.util.Arrays.sort(int[] arr, Collections.reverseOrder())
- Class MyCmp implements Comparator<Student> {
  public int compare(Student a, Student b) { return a.id – b.id;} }
- Java.util.Arrays.sort(Student[] arr, new MyCmp())

Python: list (just like vector in C++ std)

- L1 = [1,2,3] L2 = list()
- L1.append(4) L2.append(“abc”)
- L1.insert(0, 100) # add 100 at the beginning (index, val)
- val1 = L1.pop(0) # remove first element and return
- Val2 = L1.pop() # pop last element, for this reason, list can be used as stack
- L1.remove(2) # remove the first occurrence of value 2
- L1.reverse()
- L1.sort()
- L1.count(2) // tell how many occurrence of value 2
- L1.index(3) L1[3]// return value at given index
- Size = len(L1) # return how many elements in the list

Python dict: (unordered_map in C++ std)

- D1 = { 1:”abc”, 2:”xyz} D2 = dict()
- D1[100] = “odfag” D2[“xyz”] = 987 // add new element
- len(D1) # get the total elements in dictionary
- del D1[2] #remove element using key
- D1.pop(2) # another way to remove element using key
- 2 in D1: # tell if key exists
- “abc” in D1.values(): # tell if value exists
- for k in D1: print(k); print(D1[k]) # iteratre thru all elements

Python queue & stack

```
from queue import Queue
q = Queue(maxsize = 3)
q.put('a'); q.put('b'); q.put('c')
print(q.get())
q.empty()
q.full()
q.qsize()
```

```
from queue import LifoQueue
st = LifoQueue(maxsize = 3)
st.put('a'); st.put('b'); st.put('c')
print(st.get())
st.empty()
st.full()
st.qsize()
```