Technical Interview Workshops

Day II @USC

Yongwhan Lim @THH 202
https://www.yongwhan.io
Saturday, November 19, 2022

Day II: Format

- **I** read the question aloud.
- **I** go through the example.
- **You** think about the question for **5** minutes.
- **You** will provide solution for **5** minutes.
- **I** discuss the model solution.

- We divide Day II into **8 one-hour block**s.
- We pause until the next hour mark after finishing a problem set!

Problem Set #1

#1-1

[LeetCode 127] Word Ladder (Hard)

- A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s`$_1$ `-> s`$_2$ `-> ... -> s`$_k$ such that:
  - Every adjacent pair of words differs by a single letter.
  - Every `s`$_i$ for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
  - `s`$_k$ `== endWord`
- Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the **number of words** in the **shortest transformation sequence** from* `beginWord` *to* `endWord`, *or* `0` *if no such sequence exists*.

#1-1

[LeetCode 127] Word Ladder (Hard)

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

#1-1

Time: O(NL log(NL))
Space: O(NL)

```
int ladderLength(string beginWord, string endWord,
vector<string>& wordList) {
    set<string> wordDict(wordList.begin(),wordList.end());
    map<string,int> dist;
    int sz=beginWord.size();
    queue<string> q; q.push(beginWord); dist[beginWord]=1;
    while(!q.empty()) {
        // ...
    }
    return 0;
}
```

#1-1

Time: O(NL log(NL))
Space: O(NL)

```
string cur=q.front(); q.pop();
if(cur==endWord) return dist[endWord];
for (int i=0; i<sz; i++) {
    string nxt=cur;
    for (char ch='a'; ch<='z'; ch++) {
        nxt[i]=ch;
        if(!wordDict.count(nxt)) continue;
        if(dist.count(nxt)) continue;
        dist[nxt]=dist[cur]+1;
        q.push(nxt);
    }
}
```

#1-2

[LeetCode 859] Buddy Strings (Easy)

- Given two strings `s` and `goal`, return `true` *if you can swap two letters in* `s` *so the result is equal to* `goal`*, otherwise, return* `false`.
- Swapping letters is defined as taking two indices `i` and `j` (0-indexed) such that `i != j` and swapping the characters at `s[i]` and `s[j]`.
    - For example, swapping at indices `0` and 2 in `"abcd"` results in `"cbad"`.

#1-2

[LeetCode 859] Buddy Strings (Easy)

- **Example 1:**
  - **Input**: `s = "ab", goal = "ba"`
  - **Output**: `true`
- **Example 2:**
  - **Input**: `s = "ab", goal = "ab"`
  - **Output**: `false`
- **Example 3:**
  - **Input**: `s = "aa", goal = "aa"`
  - **Output**: `true`

#1-2

Time: O(L log(L))
Space: O(L)

```
bool buddyStrings(string a, string b) {
    if(a.size()!=b.size()) return false;
    string x,y;
    for (int i=0; i<a.size(); i++)
        if(a[i]!=b[i]) x+=a[i], y+=b[i];
    if(x.size()==2) { swap(y[0],y[1]); return x==y; }
    if(!x.empty()) return false;
    set<char> st(a.begin(),a.end());
    return (st.size()!=a.size());
}
```

#1-3

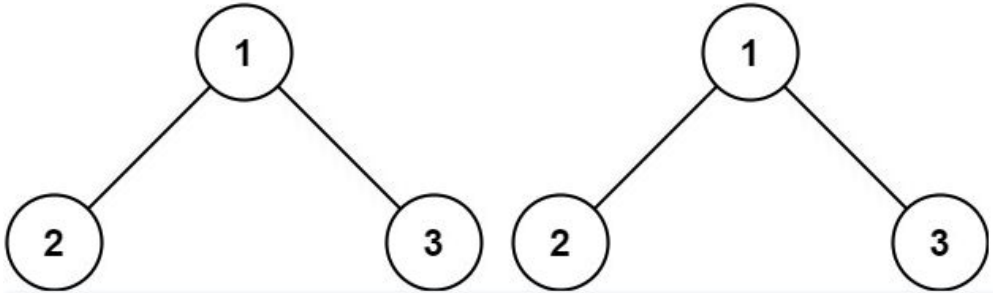
[LeetCode 100] Same Tree (Easy)

- Given the roots of two binary trees p and q, write a function to check if they are the same or not.
- Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

#1-3

[LeetCode 100] Same Tree (Easy)

- `true`



#1-3

Time: O(N)
Space: O(N)

```
bool isSameTree(TreeNode* p, TreeNode* q) {
    if(p==NULL&&q==NULL) return true;
    else if(p==NULL||q==NULL) return false;
    return p->val==q->val&&isSameTree(p->left,q->left)
                         &&isSameTree(p->right,q->right);
}
```

#1-4

[LeetCode 461] Hamming Distance (Easy)

- The Hamming distance between two integers is the number of positions at which the corresponding bits are different.
- Given two integers x and y, return *the **Hamming distance** between them*.

#1-4

[LeetCode 461] Hamming Distance (Easy)

- **Example 1:**
  - **Input**: `x = 1, y = 4`
  - **Output**: 2
- **Example 2:**
  - **Input**: `x = 3, y = 1`
  - **Output**: 1

#1-4

```
int hammingDistance(int x, int y) {
    return __builtin_popcount(x^y);
}
```

Time: O(1)
Space: O(1)

#1-5

[LeetCode 389] Find the Difference (Easy)

- You are given two strings `s` and `t`.
- String `t` is generated by random shuffling string `s` and then add one more letter at a random position.
- Return the letter that was added to `t`.

#1-5

[LeetCode 389] Find the Difference (Easy)

- **Example 1:**
  - **Input**: `s = "abcd", t = "abcde"`
  - **Output**: `"e"`
  - **Explanation**: `'e' is the letter that was added.`
- **Example 2:**
  - **Input**: `s = "", t = "y"`
  - **Output**: `"y"`

#1-5

Time: O(N+M)
Space: O(1)

```
char findTheDifference(string s, string t) {
    char ret=0;
    for (char ch : s) ret^=ch;
    for (char ch : t) ret^=ch;
    return ret;
}
```

Problem Set #2

#2-1

[LeetCode 15] 3Sum (Medium)

- Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.
- Notice that the solution set must not contain duplicate triplets.

#2-1

[LeetCode 15] 3Sum (Medium)

- **Example 1**:
  - **Input**: `nums = [-1,0,1,2,-1,-4]`
  - **Output**: `[[-1,-1,2],[-1,0,1]]`
- **Example 2**:
  - **Input**: `nums = [0,1,1]`
  - **Output**: `[]`

#2-1

Time: $O(n^2)$
Add'l Space: $O(n^2)$

```
vector<vector<int>> threeSum(vector<int> &v) {
    int n=v.size();
    if(n<3) return {};
    vector<vector<int>> ret;
    sort(v.begin(), v.end());
    for(int i=0; i<n-2; i++) {
        if(i>0 && v[i]==v[i-1]) continue;
        int j=i+1, k=n-1;
        while(j<k) { /* */ }
    }
    return ret;
}
```

Time: $O(n^2)$
Add'l Space: $O(n^2)$

#2-1

```
int val=v[i]+v[j]+v[k];
if(val>0) k--;
else if(val<0) j++;
else {
    ret.push_back({v[i],v[j],v[k]}), j++, k--;
    while(j<k && v[j]==v[j-1]) j++;
    while(j<k && v[k]==v[k+1]) k--;
}
```

#2-2

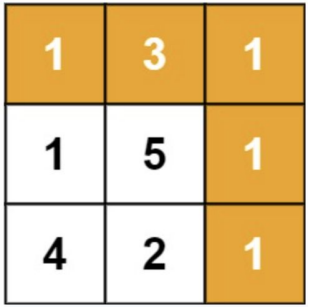
[LeetCode 64] Minimum Path Sum (Medium)

- Given a `m x n grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.
- **Note**: You can only move either down or right at any point in time.

#2-2

[LeetCode 64] Minimum Path Sum (Medium)

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



#2-2

Time: O(nm)
Space: O(nm)

```
int minPathSum(vector<vector<int>>& ret) {
    int n=ret.size(), m=ret[0].size();
    for (int i=1; i<n; i++)
        ret[i][0]+=ret[i-1][0];
    for (int j=1; j<m; j++)
        ret[0][j]+=ret[0][j-1];
    for (int i=1; i<n; i++)
        for (int j=1; j<m; j++)
            ret[i][j]+=min(ret[i-1][j],ret[i][j-1]);
    return ret[n-1][m-1];
}
```

#2-3

[LeetCode 78] Subsets (Medium)

- Given an integer array nums of **unique** elements, return *all possible subsets (the power set)*.
- The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

#2-3

[LeetCode 78] Subsets (Medium)

- **Example 1**:
  - **Input**: `nums = [1,2,3]`
  - **Output**: `[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`
- **Example 2**:
  - **Input**: `nums = [0]`
  - **Output**: `[[],[0]]`

#2-3

Time: $O(N \cdot 2^N)$
Space: $O(N \cdot 2^N)$

```
vector<vector<int>> subsets(vector<int>& v) {
    int sz=v.size();
    vector<vector<int>> ret;
    sort(v.begin(), v.end());
    for (int bt=0; bt<(1<<sz); bt++) {
        vector<int> cur;
        for (int i=0; i<sz; i++)
            if(bt&(1<<i)) cur.push_back(v[i]);
        ret.push_back(cur);
    }
    return ret;
}
```

#2-4

[LeetCode 90] Subsets II (Medium)

- Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set).*
- The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

#2-4

[LeetCode 90] Subsets II (Medium)

- **Example 1**:
  - **Input**: `nums = [1,2,2]`
  - **Output**: `[[],[1],[1,2],[1,2,2],[2],[2,2]]`
- **Example 2**:
  - **Input**: `nums = [0]`
  - **Output**: `[[],[0]]`

#2-4

Time: $O(2^N \cdot N \cdot \log N)$
Space: $O(2^N \cdot N \cdot \log N)$

```
vector<vector<int>> subsetsWithDup(vector<int>& v) {
    sort(v.begin(), v.end());
    set<vector<int>> st;
    for (int bt=0; bt<(1<<v.size()); bt++) {
        vector<int> cur;
        for (int i=0; i<v.size(); i++)
            if(bt&(1<<i)) cur.push_back(v[i]);
        st.insert(cur);
    }
    vector<vector<int>> ret(st.begin(), st.end());
    return ret;
}
```

#2-5

[LeetCode 151] Reverse Words in a String (Medium)

- Given an input string s, reverse the order of the **words**.
- A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.
- Return *a string of the words in reverse order concatenated by a single space*.
- **Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

#2-5

[LeetCode 151] Reverse Words in a String (Medium)

- **Example 1**:
  - **Input**: `s = "the sky is blue"`
  - **Output**: `"blue is sky the"`
- **Example 2**:
  - **Input**: `s = "  hello world  "`
  - **Output**: `"world hello"`
  - **Explanation**: `Your reversed string should not contain leading or trailing spaces.`

#2-5

Time: O(n)
Space: O(n)

```
string reverseWords(string s) {
    stringstream ss(s); vector<string> v;
    string cur, ret;
    while(ss>>cur) v.push_back(cur);
    reverse(v.begin(),v.end());
    bool first=true;
    for (auto &e : v) {
        if(!first) ret+=" ";
        ret+=e, first=false;
    }
    return ret;
}
```

Problem Set #3

#3-1

[LeetCode 50] **Pow(x, n)** (Medium)

- Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).

#3-1

[LeetCode 50] **Pow(x, n)** (Medium)

- **Example 1**:
  - **Input**: `x = 2.00000, n = 10`
  - **Output**: `1024.00000`
- **Example 2**:
  - **Input**: `x = 2.10000, n = 3`
  - **Output**: `9.26100`
- **Example 3**:
  - **Input**: `x = 2.00000, n = -2`
  - **Output**: `0.25000`

#3-1

Time: O(log |n|)
Space: O(1)

```
double myPow(double x, int n) {
    bool minus=false;
    long long m=n;
    if(m<0) m=-m, minus=true;
    double ret=1, val=x;
    while(m) {
        if(m%2) ret=(ret*val);
        val=val*val, m>>=1;
    }
    return minus ? 1/ret : ret;
}
```

#3-2

[LeetCode 204] **Count Primes** (Medium)

- Given an integer n, return *the number of prime numbers that are strictly less than* n.

#3-2

[LeetCode 204] **Count Primes** (Medium)

- **Example 1**:
  - **Input**: `n = 10`
  - **Output**: `4`
  - **Explanation**: `There are 4 prime numbers less than 10; they are 2, 3, 5, 7.`
- **Example 2**:
  - **Input**: `n = 0`
  - **Output**: `0`
- **Example 3**:
  - **Input**: `n = 1`
  - **Output**: `0`

Time: O(N log log N)
Space: O(N)

#3-3

```
int countPrimes(int n) {
    vector<bool> flag(int(1e7),true);
    flag[0]=flag[1]=false;
    for (int i=2; i*i<n; i++)
        if(flag[i])
            for (int j=i; j*i<n; j++)
                flag[j*i]=false;
    int ret=0;
    for (int i=0; i<n; i++)
        if(flag[i]) ret++;
    return ret;
}
```

#3-4

[LeetCode 678] **Valid Parenthesis String** (Medium)

- Given a string `s` containing only three types of characters: `'('`, `')'` and `'*'`, return `true` *if* `s` *is* ***valid***.
- The following rules define a **valid** string:
  - Any left parenthesis `'('` must have a corresponding right parenthesis `')'`.
  - Any right parenthesis `')'` must have a corresponding left parenthesis `'('`.
  - Left parenthesis `'('` must go before the corresponding right parenthesis `')'`.
  - `'*'` could be treated as a single right parenthesis `')'` or a single left parenthesis `'('` or an empty string `""`.

#3-4

[LeetCode 678] **Valid Parenthesis String** (Medium)

- **Example 1**:
  - **Input**: `s = "()"`
  - **Output**: `true`
- **Example 2**:
  - **Input**: `s = "(*)"`
  - **Output**: `true`
- **Example 3**:
  - **Input**: `s = "(*))"`
  - **Output**: `true`

#3-4

Time: O(N)
Space: O(1)

```
bool checkValidString(string s) {
    int left=0, right=0;
    for (char ch : s) {
        left += (ch=='(') ? 1 : -1,
        right += (ch!=')') ? 1 : -1;
        if(right<0) break;
        left=max(left,0);
    }
    return !left;
}
```

Problem Set #4

#4-1

[LeetCode 1079] Letter Tile Possibilities (Medium)

- You have `n  tiles`, where each tile has one letter `tiles[i]` printed on it.
- Return *the number of possible non-empty sequences of letters* you can make using the letters printed on those `tiles`.
- n can be at most 7.

#4-1

[LeetCode 1079] Letter Tile Possibilities (Medium)

- **Example 1**:
  - **Input**: `tiles = "AAB"`
  - **Output**: `8`
  - **Explanation**: `The possible sequences are "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA".`
- **Example 2**:
  - **Input**: `tiles = "AAABBC"`
  - **Output**: `188`
- **Example 3**:
  - **Input**: `tiles = "V"`
  - **Output**: `1`

Time: O(n·n!)
Space: O(n·n!)

#4-1

```
int numTilePossibilities(string s) {
    sort(s.begin(),s.end());
    set<string> st;
    int n=s.size();
    do {
        string cur;
        for (int i=0; i<n; i++)
            cur+=s[i], st.insert(cur);
    } while(next_permutation(s.begin(),s.end()));
    return st.size();
}
```

#4-2

[LeetCode 1395] Count Number of Teams (Medium)

- There are n soldiers standing in a line. Each soldier is assigned a **unique** `rating` value.
- You have to form a team of 3 soldiers amongst them under the following rules:
  - Choose 3 soldiers with index (`i`, `j`, `k`) with rating (`rating[i]`, `rating[j]`, `rating[k]`).
  - A team is valid if: (`rating[i] < rating[j] < rating[k]`) or (`rating[i] > rating[j] > rating[k]`) where (`0 <= i < j < k < n`).
- Return the number of teams you can form given the conditions. (soldiers can be part of multiple teams).
- n can be up to 1000.

#4-2

[LeetCode 1395] Count Number of Teams (Medium)

- **Example 1**:
  - **Input**: `rating = [2,5,3,4,1]`
  - **Output**: `3`
  - **Explanation**: `We can form three teams given the conditions. (2,3,4), (5,4,1), (5,3,1).`
- **Example 2**:
  - **Input**: `rating = [2,1,3]`
  - **Output**: `0`
  - **Explanation**: `We can't form any team given the conditions.`

#4-2

Time: $O(n^2)$
Space: O(1)

```
int numTeams(vector<int>& v) {
    int n=v.size(), ret=0;
    for (int i=1; i<n-1; i++) {
        vector<int> a(2,0), b(2,0);
        for (int j=0; j<i; j++)
            if(v[j]<v[i]) a[0]++; else a[1]++;
        for (int j=i+1; j<n; j++)
            if(v[j]<v[i]) b[0]++; else b[1]++;
        ret+=a[0]*b[1]+a[1]*b[0];
    }
    return ret;
}
```

#4-3

[LeetCode 973] K Closest Points to Origin (Medium)

- Given an array of `points` where `points[i] = [x_i, y_i]` represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin `(0, 0)`.
- The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., square root of sum of squares of differences).
- You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

#4-3

[LeetCode 973] K Closest Points to Origin (Medium)

- **Example**:
  - **Input**: `points = [[1,3],[-2,2]], k = 1`
  - **Output**: `[[-2,2]]`
  - **Explanation**:
    `The distance between (1, 3) and the origin is sqrt(10).`
    `The distance between (-2, 2) and the origin is sqrt(8).`
    `Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.`
    `We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].`

#4-3

Time: O(N log N)
Space: O(K)

```
int dist(const vector<int> &x) {
    return x[0]*x[0]+x[1]*x[1];
}
vector<vector<int>> kClosest(vector<vector<int>>&
points, int K) {
    sort(points.begin(),points.end(), [this](const
vector<int> &x, const vector<int> &y) -> bool { return
this->dist(x) < this->dist(y); });
    points.resize(K);
    return points;
}
```

#4-4

[LeetCode 1362] Closest Divisors (Medium)

- Given an integer `num`, find the closest two integers in absolute difference whose product equals `num + 1` or `num + 2`.
- Return the two integers in any order.

#4-4

[LeetCode 1362] Closest Divisors (Medium)

- **Example 1**:
  - **Input**: `num = 8`
  - **Output**: `[3,3]`
  - **Explanation**: `For num + 1 = 9, the closest divisors are 3 & 3, for num + 2 = 10, the closest divisors are 2 & 5, hence 3 & 3 is chosen.`
- **Example 2**:
  - **Input**: `num = 123`
  - **Output**: `[5,25]`

#4-4

Time: O(√N)
Space: O(1)

```
vector<int> closestDivisors(int num) {
    int x=-1, y=-1, diff=INT_MAX;
    for (int i=1; i<=2; i++) {
        int z=num+i, x0=-1, y0=-1, d0=INT_MAX;
        for (ll d=1; d*d<=z; d++) if(z%d==0) {
            ll e=z/d, cur=abs(d-e);
            if(d0>cur) d0=cur, x0=d, y0=e;
        }
        if(diff>d0) diff=d0, x=x0, y=y0;
    }
    return {x,y};
}
```

#4-5

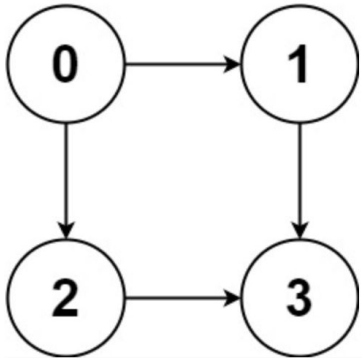
[LeetCode 797] All Paths From Source to Target (Medium)

- Given a directed acyclic graph (**DAG**) of `n` nodes labeled from `0` to `n - 1`, find all possible paths from node `0` to node `n - 1` and return them in **any order**.
- The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node `i` (i.e., there is a directed edge from node `i` to node `graph[i][j]`).

#4-5

[LeetCode 797] All Paths From Source to Target (Medium)

- **Example**:
  - **Input**: `graph = [[1,2],[3],[3],[]]`
  - **Output**: `[[0,1,3],[0,2,3]]`
  - **Explanation**: `There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.`



#4-5

Time: O(V+E)
Space: O(E)

```
vector<vector<int>> ret;

vector<vector<int>>
allPathsSourceTarget(vector<vector<int>>& graph) {
    vector<int> empty;
    traverse(graph, empty, 0);
    return ret;
}
```

Time: O(V+E)
Space: O(E)

#4-5

```
void traverse(vector<vector<int>>& graph,
              vector<int>& path,
              int cur) {
    path.push_back(cur);
    if (cur+1==graph.size()) {
        ret.push_back(path), path.pop_back();
        return;
    }
    for (int next : graph[cur])
        traverse(graph, path, next);
    path.pop_back();
}
```

#4-6

[LeetCode 136] Single Number (Easy)

- Given a **non-empty** array of integers `nums`, every element appears twice except for one. Find that single one.
- You must implement a solution with a linear runtime complexity and use only constant extra space.

#4-6

[LeetCode 136] Single Number (Easy)

- **Example 1**:
  - **Input**: `nums = [2, 2, 1]`
  - **Output**: `1`
- **Example 2**:
  - **Input**: `nums = [4,1,2,1,2]`
  - **Output**: `4`
- **Example 3**:
  - **Input**: `nums = [1]`
  - **Output**: `1`

#4-6

Time: O(N)
Space: O(1)

```
int singleNumber(vector<int>& v) {
    int x=0;
    for (auto y : v)
        x^=y;
    return x;
}
```

Problem Set #5

#5-1

[LeetCode 414] Third Maximum Number (Easy)

- Given an integer array `nums`, return the ***third distinct maximum*** *number in this array*. *If the third maximum does not exist, return the* ***maximum*** *number*.

#5-1

[LeetCode 414] Third Maximum Number (Easy)

- **Example**:
  - **Input**: `nums = [3,2,1]`
  - **Output**: `1`
  - **Explanation**:
    `The first distinct maximum is 3.`
    `The second distinct maximum is 2.`
    `The third distinct maximum is 1.`

Time: O(n log n)
Space: O(n)

#5-1

```
class Solution {
public:
    int thirdMax(vector<int>& v) {
        set<int> st(v.begin(),v.end());
        vector<int> w(st.begin(),st.end());
        return (w.size()<3) ? w.back()
                            : w[w.size()-3];
    }
};
```

#5-2

[LeetCode 605] Can Place Flowers (Easy)

- You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in **adjacent** plots.
- Given an integer array `flowerbed` containing `0`'s and 1's, where `0` means empty and 1 means not empty, and an integer n, return *if* n new flowers can be planted in the `flowerbed` without violating the no-adjacent-flowers rule.

#5-2

[LeetCode 605] Can Place Flowers (Easy)

- **Example 1**:
  - **Input**: `flowerbed = [1,0,0,0,1], n = 1`
  - **Output**: `true`
- **Example 2**:
  - **Input**: `flowerbed = [1,0,0,0,1], n = 2`
  - **Output**: `false`

#5-2

Time: O(n)
Space: O(1)

```
bool canPlaceFlowers(vector<int>& v, int k) {
    int ret=0, n=v.size();
    for (int i=0; i<n; i++) {
        if(v[i]) continue;
        bool ok=true;
        for (int j : {-1,1}) {
            int k=i+j;
            if(k<0||k>=n) continue;
            if(v[k]) ok=false;
        }
        if(ok) ret++, v[i]=true;
    }
    return k<=ret;
}
```

#5-3

[LeetCode 168] Excel Sheet Column Title (Easy)

- Given an integer `columnNumber`, return *its corresponding column title as it appears in an Excel sheet*.
- For example:

```
A -> 1
B -> 2
C -> 3
…
Z -> 26
AA -> 27
...
```

#5-3

[LeetCode 168] Excel Sheet Column Title (Easy)

- **Example 1**:
  - **Input**: `columnNumber = 1`
  - **Output**: `"A"`
- **Example 2**:
  - **Input**: `columnNumber = 28`
  - **Output**: `"AB"`
- **Example 3**:
  - **Input**: `columnNumber = 701`
  - **Output**: `"ZY"`

Time: O(log n)
Space: O(log n)

#5-3

```
class Solution {
public:
    string convertToTitle(int n) {
        string s;
        while(n)
            s=char('A'+(n-1)%26)+s, n=(n-1)/26;
        return s;
    }
};
```

#5-4

[LeetCode 1346] Check If N and Its Double Exist (Easy)

- Given an array `arr` of integers, check if there exists two indices `i` and `j` such that:
  - `i != j`
  - `0 <= i, j < arr.length`
  - `arr[i] == 2 * arr[j]`

#5-4

[LeetCode 1346] Check If N and Its Double Exist (Easy)

- **Example 1**:
  - **Input**: `arr = [10,2,5,3]`
  - **Output**: `true`
  - **Explanation**: `For i = 0 and j = 2, arr[i] == 10 == 2 * 5 == 2 * arr[j]`
- **Example 2**:
  - **Input**: `arr = [3,1,7,11]`
  - **Output**: `false`
  - **Explanation**: `There is no i and j that satisfy the conditions.`

Time: $O(n^2)$
Space: $O(1)$

#5-4

```
class Solution {
public:
    bool checkIfExist(vector<int>& v) {
        int n=v.size();
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                if(i!=j && v[i]==2*v[j])
                    return true;
        return false;
    }
};
```

#5-5

[LeetCode 58] Length of Last Word (Easy)

- Given a string s consisting of words and spaces, return *the length of the **last** word in the string*.
- A **word** is a maximal substring consisting of non-space characters only.

#5-5

[LeetCode 58] Length of Last Word (Easy)

- **Example 1**:
  - **Input**: `s = "Hello World"`
  - **Output**: `5`
  - **Explanation**: `The last word is "World" with length 5.`
- **Example 2**:
  - **Input**: `s = "   fly me   to   the moon  "`
  - **Output**: `4`
  - **Explanation**: `The last word is "moon" with length 4.`

Time: O(n)
Space: O(n)

#5-5

```
class Solution {
public:
    int lengthOfLastWord(string s) {
        stringstream ss(s);
        string cur;
        while(ss>>cur);
        return cur.size();
    }
};
```

#5-6

[LeetCode 1201] Ugly Number III (Medium)

- An **ugly number** is a positive integer that is divisible by a, b, or c.
- Given four integers n, a, b, and c, return the n^{th} **ugly number**.

#5-6

[LeetCode 1201] Ugly Number III (Medium)

- **Example 1**:
  - **Input**: `n = 3, a = 2, b = 3, c = 5`
  - **Output**: `4`
  - **Explanation**: `The ugly numbers are 2, 3, 4, 5, 6, 8, 9... The 3rd is 4.`
- **Example 2**:
  - **Input**: `n = 4, a = 2, b = 3, c = 4`
  - **Output**: `6`
  - **Explanation**: `The ugly numbers are 2, 3, 4, 6, 8, 9, 10... The 4th is 6.`

Time: O(log INT_MAX)
Space: O(1)

#5-6

```
typedef long long ll;
class Solution {
public:
    int nthUglyNumber(int n, int a, int b, int c) {
        int low=1, high=INT_MAX;
        while(low<high) {
            int mid=low+((high-low)>>1);
            if(count(mid,a,b,c)>=n) high=mid;
            else low=mid+1;
        }
        return low;
    }
```

#5-6

Time: O(log INT_MAX)
Space: O(1)

```
    ll lcm(ll a, ll b) {
        return a/__gcd(a,b)*b;
    }

    ll count(ll x, ll a, ll b, ll c) {
        return x/a + x/b + x/c
               - x/lcm(a,b) - x/lcm(a,c) - x/lcm(b,c)
               + x/lcm(a,lcm(b,c));
    }
};
```

Problem Set #6

#6-1

[LeetCode 1442] **Count Triplets That Can Form Two Arrays of Equal XOR** (Medium)

- Given an array of integers `arr`.
- We want to select three indices `i`, `j` and k where (`0 <= i < j <= k < arr.length`).
- Let's define a and b as follows:
  - `a = arr[i] ^ arr[i + 1] ^ ... ^ arr[j - 1]`
  - `b = arr[j] ^ arr[j + 1] ^ ... ^ arr[k]`
- Note that ^ denotes the **bitwise-xor** operation.
- Return *the number of triplets* (`i`, `j` and k) where `a == b`.

#6-1

[LeetCode 1442] **Count Triplets That Can Form Two Arrays of Equal XOR** (Medium)

- **Example 1**:
  - **Input**: `arr = [2,3,1,6,7]`
  - **Output**: `4`
  - **Explanation**: `The triplets are (0,1,2), (0,2,2), (2,3,4) and (2,4,4)`
- **Example 2**:
  - **Input**: `arr = [1,1,1,1,1]`
  - **Output**: `10`

Time: $O(n^3)$
Space: $O(n)$

#6-1

```
int countTriplets(vector<int>& v) {
    int n=v.size();
    vector<int> pre(n+1,0);
    for (int i=0; i<n; i++) pre[i+1]=pre[i]^v[i];
    int ret=0;
    for (int i=0; i<n; i++)
        for (int j=i+1; j<n; j++)
            for (int k=j; k<n; k++)
                if((pre[j]^pre[i])==(pre[k+1]^pre[j]))
                    ret++;
    return ret;
}
```

#6-2

[LeetCode 1414] **Find the Minimum Number of Fibonacci Numbers Whose Sum Is K** (Medium)

- Given an integer k, *return the minimum number of Fibonacci numbers whose sum is equal to* k. The same Fibonacci number can be used multiple times.
- The Fibonacci numbers are defined as:
  - $F_1 = 1$
  - $F_2 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 2$.
- It is guaranteed that for the given constraints we can always find such Fibonacci numbers that sum up to k.

#6-2

[LeetCode 1414] **Find the Minimum Number of Fibonacci Numbers Whose Sum Is K** (Medium)

- **Example 1**:
  - **Input**: `k = 7`
  - **Output**: `2`
  - **Explanation**: `The Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, ... For k = 7 we can use 2 + 5 = 7.`
- **Example 2**:
  - **Input**: `k = 10`
  - **Output**: `2`
  - **Explanation**: `For k = 10 we can use 2 + 8 = 10.`

Time: O(log k)
Space: O(log k)

#6-2

```
const int mx=80;
int findMinFibonacciNumbers(int k) {
    vector<long long> fib(mx);
    fib[0]=fib[1]=1;
    for (int i=2; i<mx; i++)
        fib[i]=fib[i-1]+fib[i-2];
    int ret=0;
    for (int i=mx-1; i>=0; i--)
        ret+=k/fib[i], k%=fib[i];
    return ret;
}
```

#6-3

[LeetCode 49] **Group Anagrams** (Medium)

- Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.
- 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.

#6-3

[LeetCode 49] **Group Anagrams** (Medium)

- **Example 1**:
  - **Input**: `strs = ["eat","tea","tan","ate","nat","bat"]`
  - **Output**: `[["bat"],["nat","tan"],["ate","eat","tea"]]`
- **Example 2**:
  - **Input**: `strs = [""]`
  - **Output**: `[[""]]`
- **Example 3**:
  - **Input**: `strs = ["a"]`
  - **Output**: `[["a"]]`

#6-3

Time: O(n|s|log |s|)
Space: O(n|s|)

```
vector<vector<string>> groupAnagrams(vector<string>& v) {
    map<string,vector<string>> mp;
    for (string &cur : v) {
        string key=cur;
        sort(key.begin(),key.end());
        mp[key].push_back(cur);
    }
    vector<vector<string>> ret;
    for (auto it : mp)
        ret.push_back(it.second);
    return ret;
}
```

#6-4

[LeetCode 77] **Combinations** (Medium)

- Given two integers n and k, return *all possible combinations of* k *numbers out of the range* [1, n].
- You may return the answer in **any order**.

#6-4

[LeetCode 77] **Combinations** (Medium)

- **Example 1**:
  - **Input:** `n = 4, k = 2`
  - **Output:** `[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]`
  - **Explanation:** `There are 4 choose 2 = 6 total combinations.`
    `Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.`
- **Example 2**:
  - **Input**: `n = 1, k = 1`
  - **Output**: `[[1]]`
  - **Explanation**: `There is 1 choose 1 = 1 total combination.`

#6-4

Time: O(nCk)
Space: O(nCk)

```
vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> ret;
    vector<int> cur;
    eval(n,k,1,cur,ret);
    return ret;
}
```

Time: O(nCk)
Space: O(nCk)

#6-4

```
void eval(int n, int k, int l,
          vector<int>&cur, vector<vector<int>>&ret) {
    if(cur.size()==k) {
        ret.push_back(cur);
        return;
    }
    for (int i=l; i<=n; i++)
        cur.push_back(i),
        eval(n,k,i+1,cur,ret),
        cur.pop_back();
}
```

#6-5

[LeetCode 1366] **Rank Teams by Votes** (Medium)

- In a special ranking system, each voter gives a rank from highest to lowest to all teams participated in the competition.
- The ordering of teams is decided by who received the most position-one votes. If two or more teams tie in the first position, we consider the second position to resolve the conflict, if they tie again, we continue this process until the ties are resolved. If two or more teams are still tied after considering all positions, we rank them alphabetically based on their team letter.

#6-5

[LeetCode 1366] **Rank Teams by Votes** (Medium)

- Given an array of strings `votes` which is the votes of all voters in the ranking systems. Sort all teams according to the ranking system described above.
- Return *a string of all teams* **sorted** by the ranking system.

#6-5

[LeetCode 1366] **Rank Teams by Votes** (Medium)

- **Example**:
  - **Input**: `votes = ["ABC","ACB","ABC","ACB","ACB"]`
  - **Output**: `"ACB"`
  - **Explanation**: Team A was ranked first place by 5 voters. No other team was voted as first place so team A is the first team.
    Team B was ranked second by 2 voters and was ranked third by 3 voters.
    Team C was ranked second by 3 voters and was ranked third by 2 voters.
    As most of the voters ranked C second, team C is the second team and team B is the third.

#6-5

Time: O(nm log m)
Space: $O(m^2)$

```
string rankTeams(vector<string>& v) {
    int n=v.size(), m=v[0].size();
    map<char,vector<int>> w;
    string ret=v[0];
    for (auto ch : v[0])
        w[ch]=vector<int>(m,0);
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            w[v[i][j]][j]++;
```

#6-5

Time: O(nm log m)
Space: $O(m^2)$

```
    sort(ret.begin(),ret.end(),
         [&](char x, char y) {
        for (int i=0; i<m; i++)
            if(w[x][i]>w[y][i]) return true;
            else if(w[x][i]<w[y][i]) return false;
        return x<y;
    });
    return ret;
}
```

Problem Set #7

#7-1

[LeetCode 71] Simplify Path (Medium)

- Given a string `path`, which is an **absolute path** (starting with a slash `'/'`) to a file or directory in a Unix-style file system, convert it to the simplified **canonical path**.
- In a Unix-style file system, a period `'.'` refers to the current directory, a double period `'..'` refers to the directory up a level, and any multiple consecutive slashes (e.g. `'//'`) are treated as a single slash `'/'`. For this problem, any other format of periods such as `'...'` are treated as file/directory names.

#7-1

[LeetCode 71] Simplify Path (Medium)

- The **canonical path** should have the following format:
  - The path starts with a single slash `'/'`.
  - Any two directories are separated by a single slash `'/'`.
  - The path does not end with a trailing `'/'`.
  - The path only contains the directories on the path from the root directory to the target file or directory (i.e., no period `'.'` or double period `'..'`)
- Return *the simplified* ***canonical path***.

#7-1

[LeetCode 71] Simplify Path (Medium)

- **Example 1**:
  - **Input**: `path = "/home/"`
  - **Output**: `"/home"`
  - **Explanation**: `Note that there is no trailing slash after the last directory name.`
- **Example 2**:
  - **Input**: `path = "/home//foo/"`
  - **Output**: `"/home/foo"`
  - **Explanation**: `In the canonical path, multiple consecutive slashes are replaced by a single one.`

#7-1

Time: O(n)
Space: O(n)

```
string simplifyPath(string s) {
    int sz=s.size();
    for (int i=0; i<sz; i++)
        if(s[i]=='/') s[i]=' ';
```

#7-1

Time: O(n)
Space: O(n)

```
stringstream ss(s);
stack<string> stk;
string cur;
while(ss>>cur) {
    if(cur="..") {
        if(!stk.empty()) stk.pop();
    } else if(cur=".") {
    } else stk.push(cur);
}
```

#7-1

Time: O(n)
Space: O(n)

```
    string ret;
    while(!stk.empty())
        ret="/"+stk.top()+ret, stk.pop();
    if(ret.empty()) return "/";
    return ret;
}
```

#7-2

[LeetCode 75] Sort Colors (Medium)

- Given an array `nums` with `n` objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.
- We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.
- You must solve this problem without using the library's sort function.

#7-2

[LeetCode 75] Sort Colors (Medium)

- **Example 1**:
  - **Input**: `nums = [2,0,2,1,1,0]`
  - **Output**: `[0,0,1,1,2,2]`
- **Example 2**:
  - **Input**: `nums = [2,0,1]`
  - **Output**: `[0,1,2]`

#7-2

Time: O(n)
Add'l Space: O(1)

```
void sortColors(vector<int>& v) {
    int sz=v.size(), red=0, blue=sz-1, i=0;
    for (int i=0; i<=blue; i++) {
        if(v[i]==0)
            swap(v[i],v[red]), red++;
        else if(v[i]==2)
            swap(v[i],v[blue]), blue--, i--;
    }
}
```

#7-3

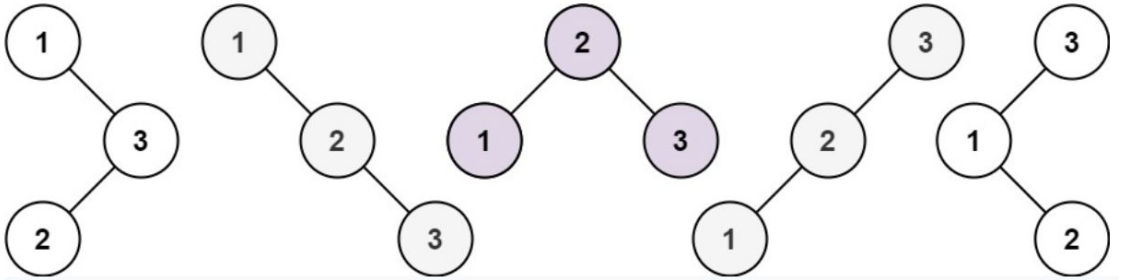
[LeetCode 96] Unique Binary Search Trees (Medium)

- Given an integer n, return *the number of structurally unique **BST**'s (binary search trees) which has exactly n nodes of unique values from* 1 *to* n.

#7-3

[LeetCode 96] Unique Binary Search Trees (Medium)

- **Example**:
  - **Input**: n = 3
  - **Output**: 5



Time: $O(n^2)$
Space: $O(n^2)$

#7-3

```
const int mx=50;
typedef long long ll;
int numTrees(int n) {
    ll binom[mx][mx];
    for (int i=0; i<mx; i++) {
        binom[i][0]=binom[i][i]=1;
        for (int j=1; j<i; j++)
            binom[i][j]=binom[i-1][j]+binom[i-1][j-1];
    }
    return int(binom[2*n][n]/(n+1));
}
```

#7-4

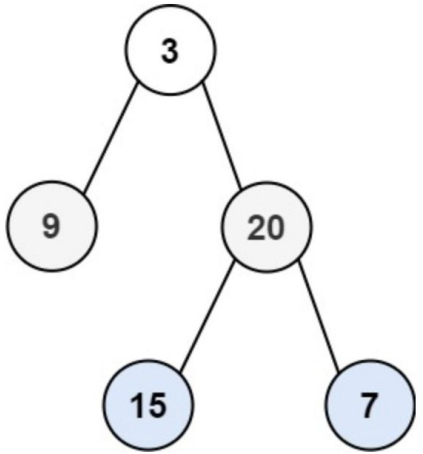
[LeetCode 107] Binary Tree Level Order Traversal II (Medium)

- Given the `root` of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

#7-4

[LeetCode 107] Binary Tree Level Order Traversal II (Medium)

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



#7-4

Time: O(n)
Space: O(n)

```
vector<vector<int>> levelOrderBottom(TreeNode *root) {
    vector<vector<int>> ans;
    levelOrderBottom(root, 0, ans);
    reverse(ans.begin(), ans.end());
    return ans;
}
```

#7-4

Time: O(n)
Space: O(n)

```
void levelOrderBottom(TreeNode *root,
                      int depth,
                      vector<vector<int>>& ans) {
    if (!root) return;
    while (ans.size() <= depth) ans.push_back({});
    ans[depth].push_back(root->val);
    levelOrderBottom(root->left, depth + 1, ans);
    levelOrderBottom(root->right, depth + 1, ans);
}
```

#7-5

[LeetCode 122] Best Time to Buy and Sell Stock II (Medium)

- You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i`th day.
- On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.
- Find and return *the **maximum** profit you can achieve*.

#7-5

[LeetCode 122] Best Time to Buy and Sell Stock II (Medium)

- **Example**:
  - **Input**: `prices = [7,1,5,3,6,4]`
  - **Output**: `7`
  - **Explanation**: `Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.`
    `Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.`
    `Total profit is 4 + 3 = 7.`

#7-5

Time: O(n)
Add'l Space: O(1)

```
int maxProfit(vector<int>& prices) {
    int ret=0;
    for (int i=1; i<prices.size(); i++)
        ret+=max(0,prices[i]-prices[i-1]);
    return ret;
}
```

Problem Set #8

#8-1

[LeetCode 791] Custom Sort String (Medium)

- You are given two strings `order` and `s`. All the words of `order` are unique and were sorted in some custom order previously.
- Permute the characters of `s` so that they match the order that `order` was sorted. More specifically, if a character `x` occurs before a character `y` in `order`, then `x` should occur before `y` in the permuted string.
- Return *any permutation of* `s` *that satisfies this property*.

#8-1

[LeetCode 791] Custom Sort String (Medium)

- **Example 1**:
  - **Input**: `order = "cba", s = "abcd"`
  - **Output**: `"cbad"`
  - **Explanation**: `"a", "b", "c" appear in order, so the order of "a", "b", "c" should be "c", "b", and "a".`
    `Since "d" does not appear in order, it can be at any position in the returned string. "dcba", "cdba", "cbda" are also valid outputs.`
- **Example 2**:
  - **Input**: `order = "cbafg", s = "abcd"`
  - **Output**: `"cbad"`

#8-1

Time: O(n log n)
Space: O(n)

```
string customSortString(string S, string T) {
    map<char,int> order;
    for (int i=0; i<S.size(); i++)
        order[S[i]]=i;
    sort(T.begin(),T.end(),[&](char a, char b) {
       return order[a]<order[b];
    });
    return T;
}
```

#8-2

[LeetCode 921] Minimum Add to Make Parentheses Valid (Medium)

- A parentheses string is valid if and only if:
  - It is the empty string,
  - It can be written as AB (A concatenated with B), where A and B are valid strings, or
  - It can be written as (A), where A is a valid string.
- You are given a parentheses string s. In one move, you can insert a parenthesis at any position of the string.
  - For example, if s = "())", you can insert an opening parenthesis to be "(())" or a closing parenthesis to be "())))".
- Return *the minimum number of moves required to make* s *valid*.

#8-2

[LeetCode 921] Minimum Add to Make Parentheses Valid (Medium)

- **Example 1**:
  - **Input**: `s = "())"`
  - **Output**: 1
- **Example 2**:
  - **Input**: `s = "((("`
  - **Output**: 3

#8-2

Time: O(n)
Add'l Space: O(1)

```
int minAddToMakeValid(string S) {
    int ret=0, stk=0;
    for (const char &ch : S) {
        if(ch=='(') stk++;
        else {
            if(!stk) ret++;
            else stk--;
        }
    }
    ret+=stk;
    return ret;
}
```

#8-3

[LeetCode 856] Score of Parentheses (Medium)

- Given a balanced parentheses string `s`, return *the **score** of the string*.
- The **score** of a balanced parentheses string is based on the following rule:
  - `"()"` has score 1.
  - `AB` has score `A + B`, where `A` and `B` are balanced parentheses strings.
  - `(A)` has score `2 * A`, where `A` is a balanced parentheses string.

#8-3

[LeetCode 856] Score of Parentheses (Medium)

- **Example 1**:
  - **Input**: `s = "()"`
  - **Output**: 1
- **Example 2**:
  - **Input**: `s = "(())"`
  - **Output**: 2
- **Example 3**:
  - **Input**: `s = "()()"`
  - **Output**: 2

Time: $O(n^2)$
Space: O(n)

#8-3

```
int scoreOfParentheses(string s) {
    if(s.empty()) return 0;
    int level=0, ret=0;
    string t;
    for (const char &ch : s) {
        t+=ch;
        if(ch=='(') level++;
        else level--;
        if(!level) ret+=getScore(t), t.clear();
    }
    return ret;
}
```

#8-3

Time: $O(n^2)$
Space: $O(n)$

```
int getScore(const string &s) {
    string t=s.substr(1,s.size()-2);
    if(t.empty()) return 1;
    return 2*scoreOfParentheses(t);
}
```

#8-4

[LeetCode 886] Possible Bipartition (Medium)

- We want to split a group of n people (labeled from 1 to n) into two groups of **any size**. Each person may dislike some other people, and they should not go into the same group.
- Given the integer n and the array `dislikes` where `dislikes[i] = [`a_i, b_i`]` indicates that the person labeled a_i does not like the person labeled b_i, return `true` *if it is possible to split everyone into two groups in this way*.

#8-4

[LeetCode 886] Possible Bipartition (Medium)

- **Example 1**:
  - **Input**: `n = 4, dislikes = [[1,2],[1,3],[2,4]]`
  - **Output**: `true`
  - **Explanation**: `group1 [1,4] and group2 [2,3].`
- **Example 2**:
  - **Input**: `n = 3, dislikes = [[1,2],[1,3],[2,3]]`
  - **Output**: `false`
- **Example 3**:
  - **Input**: `n = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]`
  - **Output**: `false`

#8-4

Time: $O(n^2)$
Space: $O(n^2)$

```
bool possibleBipartition(int n,
                         vector<vector<int>>& es) {
    vector<vector<int>> adj(n);
    for (auto e : es) {
        int x=e[0]-1, y=e[1]-1;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    vector<int> col(n,-1);
```

#8-4

Time: $O(n^2)$
Space: $O(n^2)$

```
    for (int i=0; i<n; i++) {
        if(col[i]!=-1) continue;
        queue<int> q; q.push(i); col[i]=1;
        while(!q.empty()) {
            int cur=q.front(); q.pop();
            for (int nxt : adj[cur])
                if(col[nxt]==-1)
                    col[nxt]=3-col[cur], q.push(nxt);
                else { if(col[cur]+col[nxt]!=3) return false; }
        }
    }
    return true;
}
```

#8-5

[LeetCode 789] Escape The Ghosts (Medium)

- You are playing a simplified PAC-MAN game on an infinite 2-D grid. You start at the point `[0, 0]`, and you are given a destination point `target` = $[x_{target}, y_{target}]$ that you are trying to get to. There are several ghosts on the map with their starting positions given as a 2D array ghosts, where `ghosts[i]` = $[x_i, y_i]$ represents the starting position of the i^{th} ghost. All inputs are **integral coordinates**.

#8-5

[LeetCode 789] Escape The Ghosts (Medium)

- Each turn, you and all the ghosts may independently choose to either **move 1 unit** in any of the four cardinal directions: north, east, south, or west, or **stay still**. All actions happen **simultaneously**.
- You escape if and only if you can reach the target **before** any ghost reaches you. If you reach any square (including the target) at the **same time** as a ghost, it **does not** count as an escape.
- Return `true` *if it is possible to escape regardless of how the ghosts move, otherwise return* `false`.

#8-5

[LeetCode 789] Escape The Ghosts (Medium)

- **Example 1**:
  - **Input**: `ghosts = [[1,0],[0,3]], target = [0,1]`
  - **Output**: `true`
  - **Explanation**: `You can reach the destination (0, 1) after 1 turn, while the ghosts located at (1, 0) and (0, 3) cannot catch up with you.`
- **Example 2**:
  - **Input**: `ghosts = [[1,0]], target = [2,0]`
  - **Output**: `false`
  - **Explanation**: `You need to reach the destination (2, 0), but the ghost at (1, 0) lies between you and the destination.`

#8-5

Time: O(n)
Add'l Space: O(1)

```
bool escapeGhosts(vector<vector<int>>& ghosts,
                  vector<int>& target) {
    int cur=abs(target[0])+abs(target[1]);
    for (auto ghost : ghosts)
        if(abs(target[0]-ghost[0])
            +abs(target[1]-ghost[1])<=cur)
            return false;
    return true;
}
```

Problem Set #Extra

#E-1

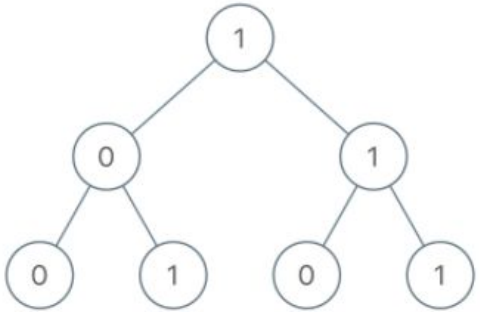
[LeetCode 1022] Sum of Root To Leaf Binary Numbers (Easy)

- You are given the `root` of a binary tree where each node has a value `0` or 1. Each root-to-leaf path represents a binary number starting with the most significant bit.
  - For example, if the path is `0 -> 1 -> 1 -> 0 -> 1`, then this could represent `01101` in binary, which is 13.
- For all leaves in the tree, consider the numbers represented by the path from the root to that leaf. Return *the sum of these numbers*.
- The test cases are generated so that the answer fits in a **32-bits** integer.

#E-1

[LeetCode 1022] Sum of Root To Leaf Binary Numbers (Easy)

- **Example**:
  - **Input**: `root = [1,0,1,0,1,0,1]`
  - **Output**: 22
  - **Explanation**: `(100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22`



Time: O(n)
Space: O(1)

#E-1

```
int sumRootToLeaf(TreeNode* root) {
    return eval(root,0);
}

int eval(TreeNode* cur, int val) {
    if(!cur) return 0;
    if(!cur->left && !cur->right)
        return val*2+cur->val;
    return eval(cur->left,val*2+cur->val)
          +eval(cur->right,val*2+cur->val);
}
```

#E-2

[LeetCode 338] Counting Bits (Easy)

- Given an integer `n`, return *an array* `ans` *of length* `n + 1` *such that for each* `i (0 <= i <= n)`, `ans[i]` *is the* ***number of*** `1`***'s*** *in the binary representation of* `i`.
- Follow up:
  - It is very easy to come up with a solution with a runtime of O(n log n). Can you do it in linear time O(n) and possibly in a single pass?
  - Can you do it without using any built-in function (i.e., like __builtin_popcount in C++)?

#E-2

[LeetCode 338] Counting Bits (Easy)

- **Example**:
  - **Input**: `n = 5`
  - **Output**: `[0,1,1,2,1,2]`
  - **Explanation**:

```
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101
```

Time: O(n)
Space: O(n)

#E-2

```
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> ret(num+1,0);
        for(int i=1; i<=num; i++)
            ret[i]=ret[i>>1]+i%2;
        return ret;
    }
};
```

#E-3

[LeetCode 1051] Height Checker (Easy)

- A school is trying to take an annual photo of all the students. The students are asked to stand in a single file line in **non-decreasing order** by height. Let this ordering be represented by the integer array `expected` where `expected[i]` is the expected height of the `i`th student in line.
- You are given an integer array `heights` representing the **current order** that the students are standing in. Each `heights[i]` is the height of the `i`th student in line (**0-indexed**).
- Return the ***number of indices*** where `heights[i] != expected[i]`.

#E-3

[LeetCode 1051] Height Checker (Easy)

- **Example**:
  - **Input**: `heights = [1,1,4,2,1,3]`
  - **Output**: `3`
    Explanation:
    `heights:  [1,1,4,2,1,3]`
    `expected: [1,1,1,2,3,4]`
    `Indices 2, 4, and 5 do not match.`

Time: O(n log n)
Space: O(n)

#E-3

```
class Solution {
public:
    int heightChecker(vector<int>& h1) {
        vector<int> h2(h1);
        sort(h2.begin(), h2.end());
        int ret=0;
        for (int i=0; i<h1.size(); i++)
            ret+=int(h1[i]!=h2[i]);
        return ret;
    }
};
```

#E-4

[LeetCode 1299] Replace Elements with Greatest Element on Right Side (Easy)

- Given an array `arr`, replace every element in that array with the greatest element among the elements to its right, and replace the last element with -1.
- After doing so, return the array.

#E-4

[LeetCode 1299] Replace Elements with Greatest Element on Right Side (Easy)

- **Example**:
  - **Input**: `arr = [17,18,5,6]`
  - **Output**: `[18,6,6,-1]`
  - **Explanation**:
    ```
    - index 0 --> the greatest element to the right of index 0 is index 1 (18).
    - index 1 --> the greatest element to the right of index 1 is index 3 (6).
    - index 2 --> the greatest element to the right of index 2 is index 3 (6).
    - index 3 --> there are no elements to the right of index 3, so we put -1.
    ```

Time: O(n)
Space: O(n)

#E-4

```
class Solution {
public:
    vector<int> replaceElements(vector<int>& v) {
        int n=v.size();
        vector<int> ret(n);
        int mx=-1;
        for (int i=n-1; i>=0; i--)
            ret[i]=mx, mx=max(mx,v[i]);
        return ret;
    }
};
```

#E-5

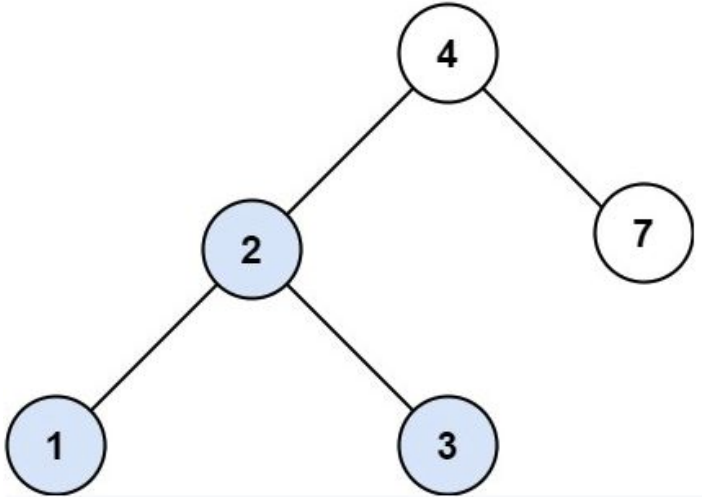
[LeetCode 700] Search in a Binary Search Tree (Easy)

- You are given the `root` of a binary search tree (BST) and an integer `val`.
- Find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.

#E-5

[LeetCode 700] Search in a Binary Search Tree (Easy)

- **Example**:
  - **Input**: `root = [4,2,7,1,3], val = 2`
  - **Output**: `[2,1,3]`



Time: O(n)
Space: O(1)

#E-5

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* cur, int val) {
        if(!cur) return NULL;
        if(cur->val==val) return cur;
        else if(cur->val>val)
            return searchBST(cur->left,val);
        else return searchBST(cur->right,val);
    }
};
```

#E-6

[LeetCode 961] N-Repeated Element in Size 2N Array (Easy)

- You are given an integer array `nums` with the following properties:
  - `nums.length == 2 * n`.
  - `nums` contains `n + 1` **unique** elements.
  - Exactly one element of `nums` is repeated `n` times.
- Return *the element that is repeated* `n` *times*.

#E-6

[LeetCode 961] N-Repeated Element in Size 2N Array (Easy)

- **Example 1**:
  - **Input**: `nums = [1,2,3,3]`
  - **Output**: 3
- **Example 2**:
  - **Input**: `nums = [2,1,2,5,3,2]`
  - **Output**: 2
- **Example 3**:
  - **Input**: `nums = [5,1,5,2,5,3,5,4]`
  - **Output**: 5

Time: O(n)
Space: O(1)

#E-6

```
class Solution {
public:
    int repeatedNTimes(vector<int>& A) {
        for (int i=0; i<A.size()-3; i++)
            for (int j=0; j<4; j++)
                for (int k=j+1; k<4; k++)
                    if(A[i+j]==A[i+k])
                        return A[i+j];
        return -1;
    }
};
```

#E-7

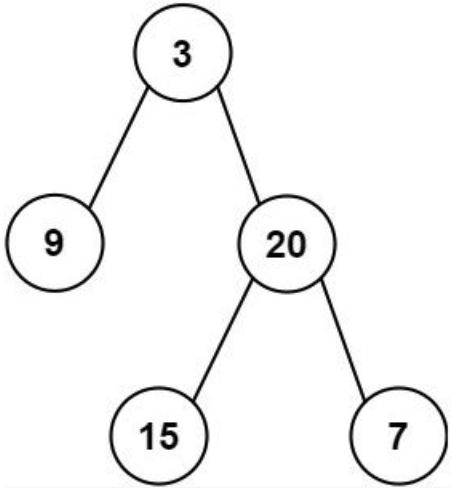
[LeetCode 637] Average of Levels in Binary Tree (Easy)

- Given the `root` of a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within 10^{-5} of the actual answer will be accepted.

#E-7

[LeetCode 637] Average of Levels in Binary Tree (Easy)

- **Example**:
  - **Input**: `root = [3,9,20,null,null,15,7]`
  - **Output**: `[3.00000,14.50000,11.00000]`
  - **Explanation**:
    `The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11. Hence return [3, 14.5, 11].`



#E-7

Time: O(n)
Space: O(n)

```
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> ret;
    queue<TreeNode*> q; q.push(root);
```

#E-7

Time: O(n)
Space: O(n)

```
    while(!q.empty()) {
        int sz=q.size(); double sum=0;
        for (int i=0; i<sz; i++) {
            auto cur=q.front(); q.pop();
            sum+=cur->val;
            if(cur->left) q.push(cur->left);
            if(cur->right) q.push(cur->right);
        }
        ret.push_back(sum/sz);
    }
    return ret;
}
```

#E-8

[LeetCode 1441] Build an Array With Stack Operations (Easy)

- You are given an array `target` and an integer `n`.
- You have an empty stack with the two following operations:
  - `"Push"`: pushes an integer to the top of the stack.
  - `"Pop"`: removes the integer on the top of the stack.
- You also have a stream of the integers in the range `[1, n]`.

#E-8

[LeetCode 1441] Build an Array With Stack Operations (Easy)

- Use the two stack operations to make the numbers in the stack (from the bottom to the top) equal to `target`. You should follow the following rules:
  - If the stream of the integers is not empty, pick the next integer from the stream and push it to the top of the stack.
  - If the stack is not empty, pop the integer at the top of the stack.
  - If, at any moment, the elements in the stack (from the bottom to the top) are equal to `target`, do not read new integers from the stream and do not do more operations on the stack.
- Return *the stack operations needed to build* `target` following the mentioned rules. If there are multiple valid answers, return **any of them**.

#E-8

[LeetCode 1441] Build an Array With Stack Operations (Easy)

- **Example**:
  - **Input**: `target = [1,3], n = 3`
  - **Output**: `["Push","Push","Pop","Push"]`
  - **Explanation**:
    `Initially the stack s is empty. The last element is the top of the stack.`
    `Read 1 from the stream and push it to the stack. s = [1].`
    `Read 2 from the stream and push it to the stack. s = [1,2].`
    `Pop the integer on the top of the stack. s = [1].`
    `Read 3 from the stream and push it to the stack. s = [1,3].`

#E-8

Time: O(n)
Space: O(n)

```
vector<string> buildArray(vector<int>& v, int n) {
    vector<string> ret;
    int cur=1;
    for (int i=0; i<v.size(); i++) {
        int nxt=v[i];
        for (int j=cur; j<nxt; j++)
            ret.push_back("Push"), ret.push_back("Pop");
        ret.push_back("Push");
        cur=nxt+1;
    }
    return ret;
}
```

#E-9

[LeetCode 852] Peak Index in a Mountain Array (Medium)

- An array `arr` a **mountain** if the following properties hold:
  - `arr.length` >= 3
  - There exists some `i` with `0 < i < arr.length - 1` such that:
    - `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`
    - `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`
- Given a mountain array `arr`, return the index `i` such that `arr[0] < arr[1] < ... < arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`.

#E-9

[LeetCode 852] Peak Index in a Mountain Array (Medium)

- **Example 1**:
  - **Input**: `arr = [0,1,0]`
  - **Output**: `1`
- **Example 2**:
  - **Input**: `arr = [0,2,1,0]`
  - **Output**: `1`
- **Example 3**:
  - **Input**: `arr = [0,10,5,2]`
  - **Output**: `1`

Time: O(n)
Space: O(1)

#E-9

```
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& A) {
        for (int i=1; i<A.size()-1; i++)
            if(A[i-1]<A[i]&&A[i]>A[i+1])
                return i;
        return -1;
    }
};
```

#E-10

[LeetCode 1464] Maximum Product of Two Elements in an Array (Easy)

- Given the array of integers nums, you will choose two different indices `i` and `j` of that array. *Return the maximum value of* `(nums[i]-1)*(nums[j]-1)`.

#E-10

[LeetCode 1464] Maximum Product of Two Elements in an Array (Easy)

- **Example 1**:
  - **Input**: `nums = [3,4,5,2]`
  - **Output**: `12`
  - **Explanation**: `If you choose the indices i=1 and j=2 (indexed from 0), you will get the maximum value, that is, (nums[1]-1)*(nums[2]-1) = (4-1)*(5-1) = 3*4 = 12.`
- **Example 2**:
  - **Input**: `nums = [1,5,4,5]`
  - **Output**: `16`
  - **Explanation**: `Choosing the indices i=1 and j=3 (indexed from 0), you will get the maximum value of (5-1)*(5-1) = 16.`

Time: O(n log n)
Space: O(1)

#E-10

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        sort(nums.rbegin(),nums.rend());
        return (nums[0]-1)*(nums[1]-1);
    }
};
```

#E-11

[LeetCode 349] Intersection of Two Arrays (Easy)

- Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

#E-11

[LeetCode 349] Intersection of Two Arrays (Easy)

- **Example 1**:
  - **Input**: `nums1 = [1,2,2,1], nums2 = [2,2]`
  - **Output**: `[2]`
- **Example 2**:
  - **Input**: `nums1 = [4,9,5], nums2 = [9,4,9,8,4]`
  - **Output**: `[9,4]`
  - **Explanation**: `[4,9] is also accepted.`

Time: O((n+m) log n)
Space: O(n)

#E-11

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1,
                             vector<int>& nums2) {
        set<int> st(nums1.begin(),nums1.end());
        vector<int> ret;
        for (const auto &x : nums2)
            if(st.count(x))
                st.erase(st.find(x)), ret.push_back(x);
        return ret;
    }
};
```

#E-12

[LeetCode 1037] Valid Boomerang (Easy)

- Given an array `points` where `points[i] = [x_i, y_i]` represents a point on the **X–Y** plane, return `true` *if these points are a* ***boomerang***.
- A **boomerang** is a set of three points that are **all distinct** and **not in a straight line**.

#E-12

[LeetCode 1037] Valid Boomerang (Easy)

- **Example 1**:
  - **Input**: `points = [[1,1],[2,3],[3,2]]`
  - **Output**: `true`
- **Example 2**:
  - **Input**: `points = [[1,1],[2,2],[3,3]]`
  - **Output**: `false`

Time: O(1)
Space: O(1)

#E-12

```
bool isBoomerang(vector<vector<int>>& v) {
    set<pair<int,int>> st;
    for (auto x : v)
        st.insert({x[0],x[1]});
    if(st.size()!=3)
        return false;
    pair<int,int> p={v[1][0]-v[0][0],v[1][1]-v[0][1]};
    pair<int,int> q={v[2][0]-v[0][0],v[2][1]-v[0][1]};
    return p.first*q.second!=p.second*q.first;
}
```

#E-13

[LeetCode 1021] Remove Outermost Parentheses (Easy)

- A valid parentheses string is either empty `""`, `"(" + A + ")"`, or `A + B`, where `A` and `B` are valid parentheses strings, and `+` represents string concatenation.
  - For example, `""`, `"()"`, `"(())()"`, and `"(()(()))"` are all valid parentheses strings.
- A valid parentheses string `s` is primitive if it is nonempty, and there does not exist a way to split it into `s = A + B`, with `A` and `B` nonempty valid parentheses strings.

#E-13

[LeetCode 1021] Remove Outermost Parentheses (Easy)

- Given a valid parentheses string s, consider its primitive decomposition: s = P_1 + P_2 + ... + P_k, where P_i are primitive valid parentheses strings.
- Return s *after removing the outermost parentheses of every primitive string in the primitive decomposition of* s.

#E-13

[LeetCode 1021] Remove Outermost Parentheses (Easy)

- **Example**:
  - **Input**: `s = "(()())(())"`
  - **Output**: `"()()()"`
  - **Explanation**:
    `The input string is "(()())(())", with primitive decomposition "(()())" + "(())".`
    `After removing outer parentheses of each part, this is "()()" + "()" = "()()()".`

#E-13

Time: O(n)
Space: O(1)

```
string removeOuterParentheses(string s) {
    string ret;
    int level=0;
    for (const auto &ch : s) {
        if(ch=='(') {
            level++;
            if(level!=1) ret+=ch;
        }
```

#E-13

Time: O(n)
Space: O(1)

```
        else {
            level--;
            if(level) ret+=ch;
        }
    }
    return ret;
}
```

#E-14

[LeetCode 1030] Matrix Cells in Distance Order (Easy)

- You are given four integers `rows`, `cols`, `rCenter`, and `cCenter`. There is a `rows x cols` matrix and you are on the cell with the coordinates `(rCenter, cCenter)`.
- Return *the coordinates of all cells in the matrix, sorted by their **distance** from* `(rCenter, cCenter)` *from the smallest distance to the largest distance*. You may return the answer in **any order** that satisfies this condition.
- The **distance** between two cells (r_1, c_1) and (r_2, c_2) is $|r_1 - r_2| + |c_1 - c_2|$.

#E-14

[LeetCode 1030] Matrix Cells in Distance Order (Easy)

- **Example 1**:
  - **Input**: `rows = 1, cols = 2, rCenter = 0, cCenter = 0`
  - **Output**: `[[0,0],[0,1]]`
  - **Explanation**: `The distances from (0, 0) to other cells are: [0,1]`
- **Example 2**:
  - **Input**: `rows = 2, cols = 2, rCenter = 0, cCenter = 1`
  - **Output**: `[[0,1],[0,0],[1,1],[1,0]]`
  - **Explanation**: `The distances from (0, 1) to other cells are: [0,1,1,2]` `The answer [[0,1],[1,1],[0,0],[1,0]] would also be accepted as correct.`

#E-14

Time: O(RC log (RC))
Space: O(RC)

```
vector<vector<int>> allCellsDistOrder(int R, int C,
                                      int r0, int c0) {
    vector<vector<int>> ret;
    for (int i=0; i<R; i++)
        for (int j=0; j<C; j++)
            ret.push_back({i,j});
    sort(ret.begin(),ret.end(),[&](const vector<int>
&a, const vector<int> &b) { return
abs(a[0]-r0)+abs(a[1]-c0)<abs(b[0]-r0)+abs(b[1]-c0);});
    return ret;
}
```

#E-15

[LeetCode 1046] Last Stone Weight (Easy)

- You are given an array of integers `stones` where `stones[i]` is the weight of the i^{th} stone.
- We are playing a game with the stones. On each turn, we choose the **heaviest two stones** and smash them together. Suppose the heaviest two stones have weights `x` and `y` with `x <= y`. The result of this smash is:
  - If `x == y`, both stones are destroyed, and
  - If `x != y`, the stone of weight `x` is destroyed, and the stone of weight `y` has new weight `y - x`.
- At the end of the game, there is **at most one** stone left.
- Return *the weight of the last remaining stone*. If there are no stones left, return `0`.

#E-15

[LeetCode 1046] Last Stone Weight (Easy)

- **Example**:
  - **Input**: `stones = [2,7,4,1,8,1]`
  - **Output**: `1`
  - **Explanation**:
    ```
    We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1]
    then,
    we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,
    we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,
    we combine 1 and 1 to get 0 so the array converts to [1] then that's
    the value of the last stone.
    ```

Time: O(n log n)
Space: O(n)

#E-15

```
int lastStoneWeight(vector<int>& v) {
    priority_queue<int> pq;
    for (const auto &x : v)
        pq.push(x);
    while(pq.size()>1) {
        int x=pq.top(); pq.pop();
        int y=pq.top(); pq.pop();
        if(x!=y) pq.push(x-y);
    }
    return pq.empty() ? 0 : pq.top();
}
```

#E-16

[LeetCode 1047] Remove All Adjacent Duplicates In String (Easy)

- You are given a string s consisting of lowercase English letters. A **duplicate removal** consists of choosing two **adjacent** and **equal** letters and removing them.
- We repeatedly make **duplicate removals** on s until we no longer can.
- Return *the final string after all such duplicate removals have been made*. It can be proven that the answer is **unique**.

#E-16

[LeetCode 1047] Remove All Adjacent Duplicates In String (Easy)

- **Example**:
  - **Input**: `s = "abbaca"`
  - **Output**: `"ca"`
  - **Explanation**:
    `For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and this is the only possible move. The result of this move is that the string is "aaca", of which only "aa" is possible, so the final string is "ca".`

#E-16

Time: O(n)
Space: O(n)

```
string removeDuplicates(const string &s) {
    stack<char> stk;
    for (const char &ch : s)
        if(stk.empty()) stk.push(ch);
        else stk.top()==ch ? stk.pop() :
                             stk.push(ch);
    string ret;
    while(!stk.empty()) ret+=stk.top(), stk.pop();
    reverse(ret.begin(),ret.end());
    return ret;
}
```

Contact Information

- **Email**: yongwhan@yongwhan.io
- **Personal Website**: https://www.yongwhan.io
- **LinkedIn Profile**: https://www.linkedin.com/in/yongwhan
  - Feel free to send me a connection request.
  - Always happy to make connections with promising students!
- **1:1 Meeting Opportunity**: https://calendly.com/yongwhan/one-on-one


THANK YOU