Past Problems

- LeetCode-Solutions
  - 896. Monotonic Array
  - 892. Surface Area of 3D Shapes
  - 890. Find and Replace Pattern
  - 888. Fair Candy Swap
  - 887. Projection Area of 3D Shapes
  - 885. Boats to Save People
  - 883. Generate Random Point in a Circle
  - 882. Random Point in Non-overlapping Rectangles
  - 873. Length of Longest Fibonacci Subsequence
  - 872. Leaf-Similar Trees
  - 870. Advantage Shuffle
  - 859. Buddy Strings
  - 846. Hand of Straights
  - 807. Max Increase to Keep City Skyline
  - 784. Letter Case Permutation
  - 746. Min Cost Climbing Stairs
  - 590. N-ary Tree Postorder Traversal
  - 589. N-ary Tree Preorder Traversal
  - 559. Maximum Depth of N-ary Tree
  - 495. Teemo Attacking
  - 442. Find All Duplicates in an Array
  - 409. Longest Palindrome
  - 429. N-ary Tree Level Order Traversal
  - 350. Intersection of Two Arrays II
  - 345. Reverse Vowels of a String
  - 274. H-Index
  - 256. Paint House
  - 238. Product of Array Except Self
  - 219. Contains Duplicate II
  - 200. Number of Islands
  - 198. House Robber
  - 169. Majority Element
  - 102. Binary Tree Level Order Traversal
  - 73. Set Matrix Zeroes
  - 69. Sqrt(x)

- 54. Spiral Matrix
- 33. Search in Rotated Sorted Array
- 19. Remove Nth Node From End of List
- 13. Roman to Integer
- 11. Container With Most Water

896. Monotonic Array

Description

Solution

The idea is that the total increment/decrement of a montonic array is same as the difference between the first and last element in the array. Hence, we just need to sum up the absolute value of the difference of two adjacent elements, then compare with the absolute value of the difference between the first element and last element.

For example, given an array `a = [1, 2, 2, 3, 4]`. Since

```
sum = |a[1] - a[0]| + |a[2] - a[1]| + |a[3] - a[2]| + |a[4] - a[3]|
    = |2 - 1| + |2 - 2| + |3 - 2| + |4 - 3|
    = 1 + 0 + 1 + 1 = 3 == |4 - 1|,
```

array `a` is a monotonic array. However, if `b = [1, 2, 1, 3, 4]`. Since

```
sum = |b[1] - b[0]| + |b[2] - b[1]| + |b[3] - b[2]| + |b[4] - b[3]|
    = |2 - 1| + |1 - 2| + |3 - 1| + |4 - 3|
    = 1 + 1 + 2 + 1
    = 5 > |4 - 1| = 3,
```

then `b` is not a monotonic array.

Here is the solution in C++. Time complexity: O(N) vs. Space complexity: O(1).

```
class Solution {
public:
    bool isMonotonic(vector<int>& A) {
        int n = A.size(), sum = 0;
        for (int i = 0; i < A.size() - 1; ++i)
            sum += abs(A[i] - A[i + 1]);
        return abs(A[n-1] - A[0]) == sum;
    }
};
```

892. Surface Area of 3D Shapes

Description

Solution

```
const int d[4][2] = { {1, 0}, {0, 1}, {-1, 0}, {0, -1} };
class Solution {
public:
    int surfaceArea(vector<vector<int>>& grid) {
        int ret = 0;
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[0].size(); ++j) {
                if (grid[i][j])
                    ret += 2;
                for (int k = 0; k < 4; ++k) {
                    int ni = i + d[k][0];
                    int nj = j + d[k][1];
                    if (ni < 0 || ni == grid.size() || nj < 0 || nj ==
grid[0].size())
                        ret += grid[i][j];
                    else if (grid[ni][nj] < grid[i][j])
                        ret += grid[i][j] - grid[ni][nj];
                }
            }
        }
        return ret;
    }
};
```

890. Find and Replace Pattern

Description

Solution

The idea is to check whether the number of letter occurrences in the given word and that of pattern are the same, besides, the number of pairs `(words[k][i], pattern[i])` should also be the same.

```
class Solution {
public:
    vector<string> findAndReplacePattern(vector<string>& words, string pattern)
{
        vector<string> ret;
        for (auto w : words)
```

```
            if (matchPattern(w, pattern))
                ret.push_back(w);
        return ret;
    }

    bool matchPattern(string word, string pattern) {
        set<char> w, p;
        set<pair<char, char>> wp;
        for (int i = 0; i < word.size(); ++i) {
            w.insert(word[i]);
            p.insert(pattern[i]);
            wp.insert(pair<char, char>(word[i], pattern[i]));
        }
        return p.size() == w.size() && p.size() == wp.size();
    }
};
```

888. Fair Candy Swap

Description

Solution

The idea is to find the difference `diff` between the sum of two arrays `A` and `B`. Then find two elements, one in `A` and the other in `B`, such that `a[i] - b[j] = diff / 2`.

```
class Solution {
public:
    vector<int> fairCandySwap(vector<int>& A, vector<int>& B) {
        sort(A.begin(), A.end());
        sort(B.begin(), B.end());
        int diff{0};
        for (auto a : A)
            diff += a;
        for (auto b : B)
            diff -= b;
        diff /= 2;
        vector<int> ret;
        for (int i = 0, j = 0; i < A.size() && j < B.size();) {
            if (A[i] - B[j] == diff) {
                ret.push_back(A[i]), ret.push_back(B[j]);
                return ret;
            }
            else if (A[i] - B[j] > diff)
                ++j;
            else
```

```
                ++i;
            }
        }
    };
```

887. Projection Area of 3D Shapes

Description

Solution

The idea is

1. For the projection on xy-plane, just count the number of nonzero entries `grid[i][j] != 0`, where `i = 0, ..., gridRowSize - 1, j = 0, ..., gridColSizes[0] - 1`.
2. For the projection on xz-plane, sum up the maximums of each row.
3. For the projection on yz-plane, sum up the maximums of each column.
4. Sum up all the numbers above.

Time complexity: O(M * N) vs. Space complexity: O(1).

```
int projectionArea(int** grid, int gridRowSize, int *gridColSizes) {
    int xy = 0, xz = 0, yz = 0;
    for (int i = 0; i < gridRowSize; ++i) {
        int rowMax = grid[i][0];
        for (int j = 0; j < gridColSizes[i]; ++j) {
            rowMax = rowMax > grid[i][j] ? rowMax : grid[i][j];
            if (grid[i][j] != 0)
                ++xy;
        }
        xz += rowMax;
    }
    for (int j = 0; j < gridColSizes[0]; ++j) {
        int colMax = grid[0][j];
        for (int i = 0; i < gridRowSize; ++i)
            colMax = colMax > grid[i][j] ? colMax : grid[i][j];
        yz += colMax;
    }
    return xy + xz + yz;
}
```

885. Boats to Save People

Description

Solution

The idea is to sort people's weight in descending order, and then try to pair the one of highest weight with the one of the lowest weight. If the total weight of the two is no more than `limit`, then pair them up and carry on. If not, the one of highest weight takes one boat, and the one of lowest weight remains waiting to be paired with the one of second highest weight. Repeat the above.

Time complexity: O(N log(N)) vs. Space comlexity: O(1).

```
int comparator(const void *a, const void *b) {
    return *(int *) a < *(int *) b;
}

int numRescueBoats(int* people, int peopleSize, int limit) {
    qsort(people, peopleSize, sizeof(int), comparator);
    int ret = 0;
    for (int l = 0, r = peopleSize - 1; l <= r; ++l, ++ret)
        if (l != r && people[l] + people[r] <= limit)
            --r;
    return ret;
}
```

883. Generate Random Point in a Circle

Description

Solution

The idea is to use generate uniformly distributed angles and radius, then transform them into Cartesian coordinates.

However, I got `Internal Error` for the Python code and `Wrong Answer` for the C code. Since these points are random, how could the expected answer be the same as my output???!

Update: Thanks to caraxin, fixed the issue. Use `x = obj->x_center + obj->raidus * sqrt((double) rand() / RAND_MAX) * cos(theta)` rather than `x = obj->x_center + obj->raidus * ((double) rand() / RAND_MAX) * cos(theta)`, same for `y`.

C:

```
typedef struct {
    double radius;
    double x_center;
    double y_center;
} Solution;
```

```c
Solution* solutionCreate(double radius, double x_center, double y_center) {
    Solution *solution = malloc(sizeof(Solution));
    solution->radius = radius;
    solution->x_center = x_center;
    solution->y_center = y_center;
    return solution;
}

double* solutionRandPoint(Solution* obj, int *returnSize) {
    *returnSize = 2;
    double *ret = malloc(*returnSize * sizeof(double));
    double theta = (double) rand() / RAND_MAX * atan(1) * 8;
    double scale = (double) rand() / RAND_MAX;
    ret[0] = obj->x_center + obj->radius * sqrt(scale) * cos(theta);
    ret[1] = obj->y_center + obj->radius * sqrt(scale) * sin(theta);
    return ret;
}

void solutionFree(Solution* obj) {
    free(obj);
}

/**
 * Your Solution struct will be instantiated and called as such:
 * struct Solution* obj = solutionCreate(radius, x_center, y_center);
 * double* param_1 = solutionRandPoint(obj);
 * solutionFree(obj);
 */
```

Python:

```python
class Solution(object):

    def __init__(self, radius, x_center, y_center):
        """
        :type radius: float
        :type x_center: float
        :type y_center: float
        """
        self.radius = radius
        self.x_center = x_center
        self.y_center = y_center

    def randPoint(self):
        """
        :rtype: List[float]
        """
```

```
        scale = random.uniform(0, 1)
        theta = random.uniform(0, 8 * math.atan(1))
        x = self.x_center + self.radius * math.sqrt(scale) * math.cos(theta)
        y = self.y_center + self.radius * math.sqrt(scale) * math.sin(theta)
        return [x, y]

# Your Solution object will be instantiated and called as such:
# obj = Solution(radius, x_center, y_center)
# param_1 = obj.randPoint()
```

882. Random Point in Non-overlapping Rectangles

Description

Solution

1. First calculate the areas of rectangles, and then calculate the ratio of area of each rectangle against the total areas of rectangels, using `cumsum` to obtain the probability to pick the rectangle.
2. Once pick certain rectangle, we determine the point on the rectangle by calculating `x = x1 - 0.5 + (double) rand() / RAND_MAX * (x2 - x1 + 1)`, then round to the nearest integer. Do the same for `y`.

C:

```
typedef struct {
    int rectsSize;
    int **rects;
    double *distribution;
} Solution;

Solution* solutionCreate(int** rects, int rectsSize) {
    double *distribution = malloc(rectsSize * sizeof(double));
    double *areas = malloc(rectsSize * sizeof(double));
    double sum = 0;
    for (int i = 0; i < rectsSize; i++) {
        areas[i] = (rects[i][2] - rects[i][0] + 1) * (rects[i][3] - rects[i][1]
+ 1);
        sum += areas[i];
    }
    for (int i = 0; i < rectsSize; i++) {
        distribution[i] = 0;
        for (int j = 0; j <= i; j++)
            distribution[i] += areas[j] / sum;
    }

    Solution *obj = malloc(sizeof(Solution));
```

```
    obj->rectsSize = rectsSize;
    obj->rects = rects;
    obj->distribution = distribution;
    free(areas);
    return obj;
}

int* solutionPick(Solution* obj, int *returnSize) {
    double pickRect = (double) rand() / RAND_MAX;
    int i;
    for (i = 0; pickRect > obj->distribution[i]; i++) {}
    double x = (obj->rects)[i][0] + (double) rand() / RAND_MAX * ((obj->rects)
[i][2] - (obj->rects)[i][0] + 1) - 0.5;
    double y = (obj->rects)[i][1] + (double) rand() / RAND_MAX * ((obj->rects)
[i][3] - (obj->rects)[i][1] + 1) - 0.5;

    int *ret = malloc(*returnSize * sizeof(int));
    ret[0] = round(x);
    ret[1] = round(y);
    *returnSize = 2;
    return ret;
}

void solutionFree(Solution* obj) {
    free(obj);
}

/**
 * Your Solution struct will be instantiated and called as such:
 * struct Solution* obj = solutionCreate(rects, rectsSize);
 * int* param_1 = solutionPick(obj);
 * solutionFree(obj);
 */
```

Python:

```
import numpy as np

class Solution:

    def __init__(self, rects):
        """
        :type rects: List[List[int]]
        """
        self.rects = rects
        areas = [(rect[2] - rect[0] + 1) * (rect[3] - rect[1] + 1) for rect in
rects]
        self.distribution = np.cumsum(areas) / sum(areas)
```

```
    def pick(self):
        """
        :rtype: List[int]
        """
        pickRect = random.uniform(0, 1)
        i = 0
        while pickRect > self.distribution[i]:
            i += 1
        x = self.rects[i][0] + random.uniform(0, 1) * (self.rects[i][2] -
self.rects[i][0] + 1) - 0.5
        y = self.rects[i][1] + random.uniform(0, 1) * (self.rects[i][3] -
self.rects[i][1] + 1) - 0.5
        return [round(x), round(y)]

# Your Solution object will be instantiated and called as such:
# obj = Solution(rects)
# param_1 = obj.pick()
```

873. Length of Longest Fibonacci Subsequence

Description

Solution

```
typedef struct linkedlist {
    int key;
    int val;
    struct linkedlist *next;
} node;

void init(node *head, int key, int val, node *next) {
    head->key = key;
    head->val = val;
    head->next = next;
}

void insert(node *head, int key, int val) {
    node* cur = head;

    if (key == cur->key)
        cur->val += val;
    for (cur = head; key > cur->key; cur = cur->next) {
        if (cur->next && key > cur->next->key)
            continue;
        else if (cur->next && key == cur->next->key)
```

```
            cur->next->val += val;
        else {
            node *new = (node *) malloc(sizeof(node));
            init(new, key, val, cur->next);
            cur->next = new;
        }
    }
}

int search(node *head, int key) {
    node *cur = head;
    while (cur && key != cur->key)
        cur = cur->next;
    return cur && key == cur->key && cur->val > 0 ? 1 : 0;
}

#define N 10000

int lenLongestFibSubseq(int* A, int ASize) {
    node *hash = malloc(N * sizeof(node));
    for (int i = 0; i < N; init(&hash[i], i, 0, NULL), i++) {}
    for (int i = 0; i < ASize; insert(&hash[A[i] % N], A[i], 1), i++) {}

    int cur_max = 0;
    for (int i = 0; i < ASize - 2; i++) {
        for (int j = i + 1; j < ASize - 1; j++) {
            int cnt = 2;
            int first = A[i], second = A[j], third = first + second;
            while (search(&hash[third % N], third)) {
                first = second, second = third, third = first + second;
                cnt++;
            }
            if (cnt > 2)
                cur_max = cur_max > cnt ? cur_max : cnt;
        }
    }
    free(hash);
    return cur_max;
}
```

872. Leaf-Similar Trees

Description

Solution

Basic idea: traverse two trees using recursion to obtain the leaves and then compare leaves one by one.

C (0 ms):

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
void traverse(struct TreeNode *root, int *leaves, int *i) {
    if (!root)
        return;
    if (!root->left && !root->right)
        leaves[(*i)++] = root->val;
    traverse(root->left, leaves, i);
    traverse(root->right, leaves, i);
}

bool leafSimilar(struct TreeNode* root1, struct TreeNode* root2) {
    int leaves1[100] = {0}, leaves2[100] = {0}, leaves1Size = 0, leaves2Size =
0;
    traverse(root1, leaves1, &leaves1Size);
    traverse(root2, leaves2, &leaves2Size);
    if (leaves1Size != leaves2Size)
        return false;
    for (int i = 0; i < leaves1Size; i++)
        if (leaves1[i] != leaves2[i])
            return false;
    return true;
}
```

Python3 (40 ms):

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def leafSimilar(self, root1, root2):
        """
        :type root1: TreeNode
        :type root2: TreeNode
```

```
        :rtype: bool
        """
        return self.traverse(root1) == self.traverse(root2)

    def traverse(self, root):
        leaves = []
        if not root:
            return leaves
        if not root.left and not root.right:
            leaves.append(root.val)
        leaves.extend(self.traverse(root.left))
        leaves.extend(self.traverse(root.right))
        return leaves
```

870. Advantage Shuffle

Description

Solution

Basic idea:

1. Use an array of `struct pair (j, b)` to store the indices and elements of array `B` in order to find keep the index of B after sorting.
2. Sorting both arrays in ascending order gives `[a1, a2, ..., an]` and `[(j1, b1), (j2, b2), ..., (jn, bn)]`.
3. Compare elements in the sorted array one by one:
   - If `a1 > b1`, then `a1` has advantage against `b1`, thus put `a1` into `ret[j1]`.
   - If `a1 <= b1`, `a1` would never have advantage against any other elements in `B`, thus put `a1` into `ret[jn]`.

Time complexity: O(N log(N)) vs. Space complexity: O(N).

```
/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */

struct pair {
    int key;
    int val;
};

int comparator(const void *a, const void *b) {
    return *(int *) a > *(int *) b;
```

```
}

int comparator_pair(const void *a, const void *b) {
    struct pair *c = (struct pair *) a;
    struct pair *d = (struct pair *) b;
    return c->val > d->val;
}

int* advantageCount(int* A, int ASize, int* B, int BSize, int* returnSize) {
    struct pair *bp = malloc(BSize * sizeof(struct pair));
    for (int i = 0; i < BSize; i++)
        bp[i].key = i, bp[i].val = B[i];
    qsort(A, ASize, sizeof(int), comparator);
    qsort(bp, BSize, sizeof(struct pair), comparator_pair);
    for (int i = 0, j = 0, k = BSize - 1; i < ASize; i++)
        if (A[i] > bp[j].val)
            B[bp[j++].key] = A[i];
        else
            B[bp[k--].key] = A[i];
    *returnSize = BSize;
    return B;
}
```

859. Buddy Strings

Description

Solution

In order to be buddy strings:

- The number of occurrences of each letter in both strings (hence also the length of two strings) shall be the same;
- If the condition above is satisfied, then the number of different occurrences must be exactly 2, or 0 unless there is at least one repeating letter.

```
bool buddyStrings(char* A, char* B) {
    int a[26] = {0}, b[26] = {0}, repeated_letter = 0, cnt = 0;
    for (int i = 0; A[i]; a[A[i++] - 'a']++) {}
    for (int i = 0; B[i]; b[B[i++] - 'a']++) {}
    for (int i = 0; i < 26; i++)
        if (a[i] != b[i])
            return false;
        else if (a[i] >= 2)
            repeated_letter = 1;
    for (int i = 0; A[i]; i++)
```

```
        if (A[i] != B[i])
            cnt++;
    return cnt == 2 || (cnt == 0 && repeated_letter);
}
```

846. Hand of Straights

Description

Solution

The idea is to check whether the remainder of `hand[i] / W` are evenly distributed, that is, the number of `0, 1, ..., W - 1` should be the same.

Time complexity: O(N) vs. Space complexity: O(N).

```
bool isNStraightHand(int* hand, int handSize, int W) {
    if (handSize % W != 0)
        return false;
    int target = handSize / W;
    int *remainder = calloc(W, sizeof(int));
    for (int i = 0; i < handSize; i++) {
        remainder[hand[i] % W] += 1;
        if (remainder[hand[i] % W] > target)
            return false;
    }
    return true;
}
```

Hand of Straights

Submission Detail

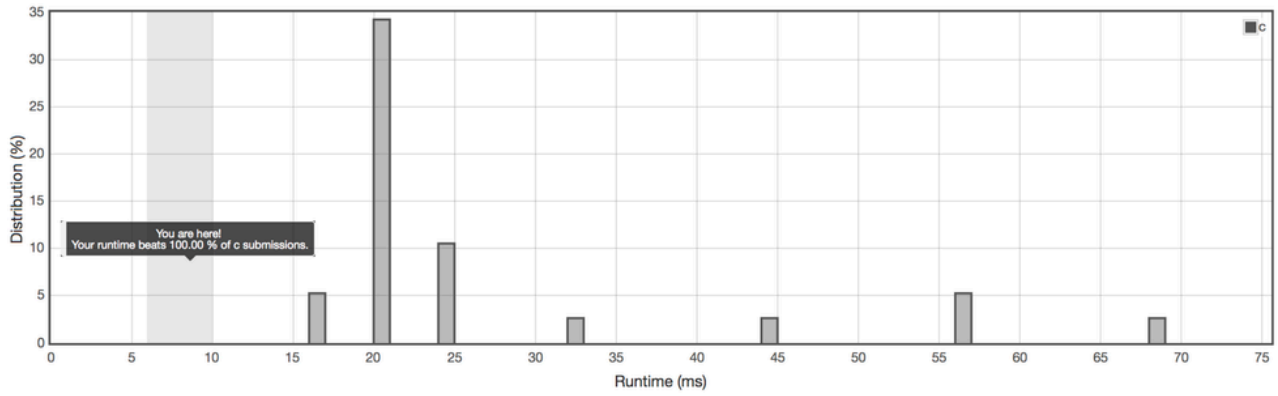
65 / 65 test cases passed.

Status: Accepted

Runtime: **8 ms**

Submitted: **0 minutes ago**

Accepted Solutions Runtime Distribution

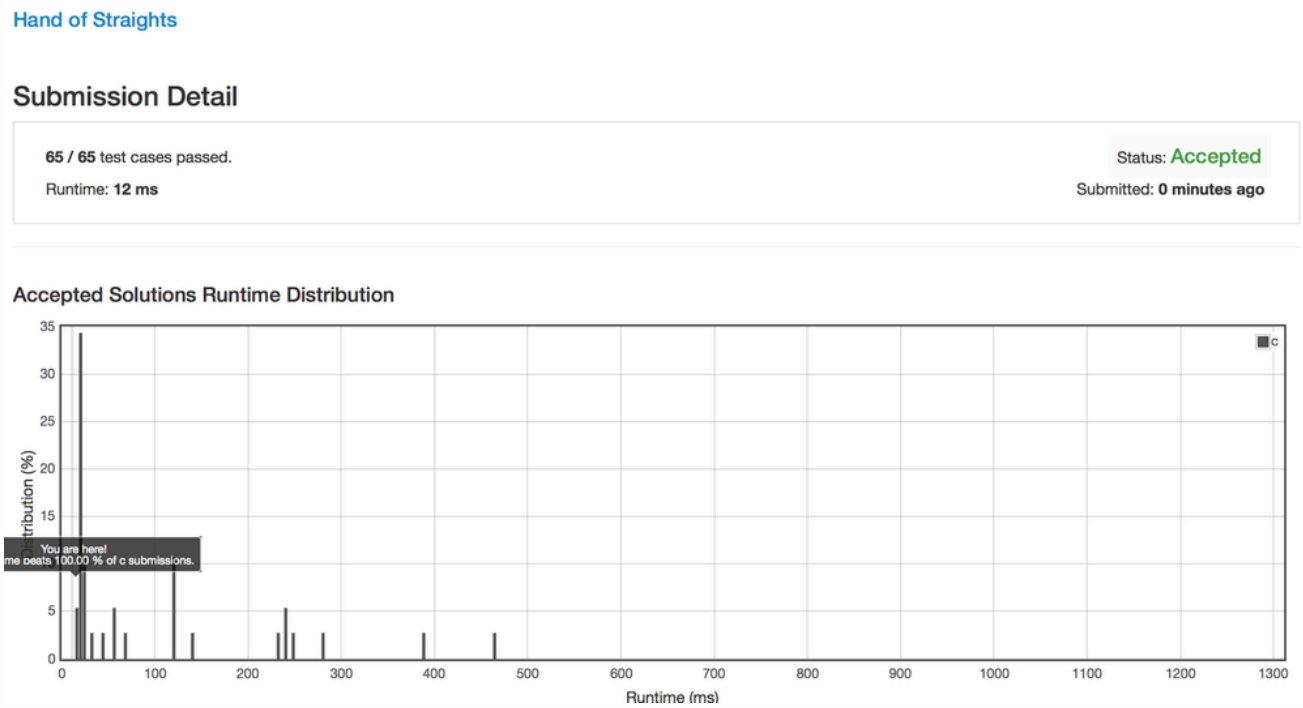


However, this doesn't mean it is flawless. For example, it cannot pass the following case: `hand = [1, 2, 3, 6], W = 2`. In order to pass the above test case, borrowed the idea from this post. Here is the corrected C code:

```c
int comparator(const void *a, const void *b) {
    return *(int *) a > *(int *) b;
}

bool isNStraightHand(int* hand, int handSize, int W) {
    if (handSize % W != 0)
        return false;

    // Check whether the number of cards of each group are the same by checking
    // whether the remainders of handSize / W are evenly distributed
    int rowSize = W, colSize = handSize / rowSize;
    int *colIndex = calloc(rowSize, sizeof(int));
    for (int i = 0; i < handSize; i++) {
        colIndex[hand[i] % W]++;
        if (colIndex[hand[i] % W] > colSize)
            return false;
    }

    // Rearrange cards such that cards in each row have the same remainder
    int **handMatrix = malloc(rowSize * sizeof(int *));
    free(colIndex), colIndex = calloc(rowSize, sizeof(int));
    for (int i = 0; i < handSize; i++) {
        int row = hand[i] % rowSize;
        if (colIndex[row] == 0)
            handMatrix[row] = malloc(colSize * sizeof(int));
```

```
            handMatrix[row][colIndex[row]++] = hand[i];
            if (colIndex[row] == colSize)
                qsort(handMatrix[row], colSize, sizeof(int), comparator);
        }

        // Check whether each column forms a straight (consecutive numbers)
        for (int j = 0; j < colSize; j++) {
            int min = INT_MAX, max = INT_MIN;
            for (int i = 0; i < rowSize; i++) {
                min = min < handMatrix[i][j] ? min : handMatrix[i][j];
                max = max > handMatrix[i][j] ? max : handMatrix[i][j];
            }
            if (max - min != rowSize - 1)
                return false;
        }

        for (int i = 0; i < rowSize; i++)
            free(handMatrix[i]);
        free(handMatrix);
        free(colIndex);
        return true;
    }
```

Still remains "beats 100%":



807. Max Increase to Keep City Skyline

Description

Solution

The idea is to sum up the difference between `grid[i][j]` and minimum of the maximum of row `i` and column `j`.

Time complexity: O(M * N) vs. Space complexity: O(N).

```
class Solution {
public:
    int maxIncreaseKeepingSkyline(vector<vector<int>>& grid) {
        int ret = 0;
        if (!grid.size())
            return ret;
        vector<int> colMax, rowMax;
        for (int j = 0; j < grid[0].size(); ++j) {
            int curColMax = grid[0][j];
            for (int i = 1; i < grid.size(); ++i)
                curColMax = max(curColMax, grid[i][j]);
            colMax.push_back(curColMax);
        }
        for (int i = 0; i < grid.size(); ++i)
            rowMax.push_back(*max_element(grid[i].begin(), grid[i].end()));
        for (int i = 0; i < grid.size(); ++i)
            for (int j = 0; j < grid[i].size(); ++j)
                ret += min(rowMax[i], colMax[j]) - grid[i][j];
        return ret;
    }
};
```

784. Letter Case Permutation

Description

Solution

Update:

More concise C++ code:

```
class Solution {
public:
    vector<string> letterCasePermutation(string S) {
        vector<string> ret;
        if (!S.size()) {
            ret.push_back("");
```

```
            return ret;
        }
        char lastCh = S.back();
        S.pop_back();
        vector<string> cur = letterCasePermutation(S);
        if (isalpha(lastCh))
            for (auto c : cur)
                ret.push_back(c + string(1, tolower(lastCh))),
                ret.push_back(c + string(1, toupper(lastCh)));
        else
            for (auto c : cur)
                ret.push_back(c + string(1, lastCh));
        return ret;
    }
};
```

C: Using `realloc()` to directly manipulate the original array of strings.

```
/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
char** letterCasePermutation(char* S, int* returnSize) {
    char **ret;
    if (!S[0]) {
        *returnSize = 1;
        ret = malloc(sizeof(char *));
        ret[0] = malloc(sizeof(char));
        ret[0][0] = '\0';
        return ret;
    }

    char lastCh = S[strlen(S) - 1];
    S[strlen(S) - 1] = '\0';
    ret = letterCasePermutation(S, returnSize);
    for (int i = 0; i < *returnSize; ++i) {
        int len = strlen(ret[i]);
        ret[i] = realloc(ret[i], (len + 2) * sizeof(char));
        ret[i][len] = lastCh;
        ret[i][len + 1] = '\0';
    }
    if (isalpha(lastCh)) {
        *returnSize *= 2;
        ret = realloc(ret, *returnSize * sizeof(char *));
        for (int i = 0; i < *returnSize / 2; ++i) {
            int len = strlen(ret[i]);
            ret[i + *returnSize / 2] = malloc((len + 1) * sizeof(char));
            ret[i + *returnSize / 2][0] = '\0';
```

```
            ret[i][len - 1] = tolower(lastCh);
            ret[i][len] = '\0';
            strcpy(ret[i + *returnSize / 2], ret[i]);
            ret[i + *returnSize / 2][len - 1] = toupper(lastCh);
        }
    }
    return ret;
}
```

Original post:

C++:

```
class Solution {
public:
    vector<string> letterCasePermutation(string S) {
        vector<string> ret;
        if (!S.size()) {
            ret.push_back("");
            return ret;
        }
        char lastCh = S.back();
        S.pop_back();
        ret = letterCasePermutation(S);
        if (isalpha(lastCh)) {
            vector<string> ret_copy = ret;
            for (int i = 0; i < ret.size(); i++) {
                ret_copy[i].insert(ret_copy[i].end(), toupper(lastCh));
                ret[i].insert(ret[i].end(), tolower(lastCh));
            }
            ret.insert(ret.end(), ret_copy.begin(), ret_copy.end());
        }
        else {
            for (auto &r : ret)
                r.insert(r.end(), lastCh);
        }
        return ret;
    }
};
```

C:

```
/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
```

```c
char** letterCasePermutation(char* S, int* returnSize) {
    char **ret;
    if (!S[0]) {
        *returnSize = 1;
        ret = malloc(sizeof(char *));
        ret[0] = malloc(sizeof(char));
        ret[0][0] = '\0';
        return ret;
    }

    int size;
    char lastCh = S[strlen(S) - 1];
    S[strlen(S) - 1] = '\0';
    char **cur = letterCasePermutation(S, &size);
    if (isalpha(lastCh)) {
        *returnSize = 2 * size;
        ret = malloc(*returnSize * sizeof(char *));
        for (int i = 0; i < size; ++i) {
            int len = strlen(cur[i]);
            ret[2 * i] = malloc((len + 2) * sizeof(char));
            ret[2 * i + 1] = malloc((len + 2) * sizeof(char));
            // sprintf(ret[2 * i], "%s%c", cur[i], tolower(lastCh));
            // sprintf(ret[2 * i + 1], "%s%c", cur[i], toupper(lastCh));
            strcpy(ret[2 * i], cur[i]);
            ret[2 * i][len] = tolower(lastCh);
            strcpy(ret[2 * i + 1], cur[i]);
            ret[2 * i + 1][len] = toupper(lastCh);
            ret[2 * i][len + 1] = '\0';
            ret[2 * i + 1][len + 1] = '\0';
            free(cur[i]);
        }
    }
    else {
        *returnSize = size;
        ret = malloc(*returnSize * sizeof(char *));
        for (int i = 0; i < size; ++i) {
            int len = strlen(cur[i]);
            ret[i] = malloc((len + 2) * sizeof(char));
            // sprintf(ret[i], "%s%c", cur[i], lastCh);
            strcpy(ret[i], cur[i]);
            ret[i][len] = lastCh;
            ret[i][len + 1] = '\0';
            free(cur[i]);
        }
    }
    free(cur);
    return ret;
}
```

746. Min Cost Climbing Stairs

Description

```
On a staircase, the i-th step has some non-negative cost cost[i] assigned (0
indexed).

Once you pay the cost, you can either climb one or two steps. You need to find
minimum cost to reach the top of the floor, and you can either start from the
step with index 0, or the step with index 1.

Example 1:

Input: cost = [10, 15, 20]
Output: 15
Explanation: Cheapest is start on cost[1], pay that cost and go to the top.
Example 2:

Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
Output: 6
Explanation: Cheapest is start on cost[0], and only step on 1s, skipping
cost[3].
Note:

cost will have a length in the range [2, 1000].
Every cost[i] will be an integer in the range [0, 999].
```

Solution

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
  int minCostClimbingStairs(vector<int>& cost) {
    cost.push_back(0);
    for (int i = 2; i < cost.size(); ++i)
      cost[i] += min(cost[i - 2], cost[i - 1]);
    return cost.back();
  }
};
```

590. N-ary Tree Postorder Traversal

Description

Solution

Update:

Iterative solution:

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int> ret;
        if (!root)
            return ret;
        vector<Node *> v{root};
        while (!v.empty()) {
            int n = v.size();
            for (int i = 0; i < n; ++i) {
                Node *cur = v.back(); v.pop_back();
                ret.insert(ret.begin(), cur->val);
                v.insert(v.end(), cur->children.begin(), cur->children.end());
            }
        }
        return ret;
    }
};
```

The idea is to use DFS based on recursion, which is the trivial case in the problem statement...

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int> ret;
        if (!root)
            return ret;
        for (auto child : root->children) {
            vector<int> tmp = postorder(child);
            ret.insert(ret.end(), tmp.begin(), tmp.end());
        }
        ret.push_back(root->val);
        return ret;
    }
};
```

589. N-ary Tree Preorder Traversal

Description

Solution

Iterative solution:

```
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int> ret;
        if (!root)
            return ret;
        vector<Node *> n{root};
        while (!n.empty()) {
            Node *cur = n.back();
            n.pop_back();
            ret.push_back(cur->val);
            n.insert(n.end(), cur->children.rbegin(), cur->children.rend());
        }
        return ret;
    }
};
```

Recursive solution:

```
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int> ret;
        if (!root)
            return ret;
        ret.push_back(root->val);
        for (auto &c : root->children) {
            vector<int> tmp = preorder(c);
            ret.insert(ret.end(), tmp.begin(), tmp.end());
        }
        return ret;
    }
};
```

559. Maximum Depth of N-ary Tree

Description

Solution

Basically, I did the same thing, but got "beats 99.58%". Here is the code:

```
class Solution {
public:
    int maxDepth(Node* root) {
        if (!root)
            return 0;
        int depth = 0;
        for (auto &c : root->children)
            depth = max(depth, maxDepth(c));
        return depth + 1;
    }
};

static int disable_io_sync__=[](){
    std::ios::sync_with_stdio(false); // disable synchronization between
scanf/printf and cin/cout
    std::cin.tie(nullptr); // disable synchronization between std::cin and
std::cout
    return 0;
}();
```

Used black magic! Yep, add last several lines, your code runs faster, magically and surprisingly.

495. Teemo Attacking

Description

Solution

The idea is very simple: add the `duration` if the time difference is between two consecutive attacking time, otherwise add the time difference.

Time complexity: O(N) vs. Space complexity: O(1).

```
class Solution {
public:
    int findPoisonedDuration(vector<int>& timeSeries, int duration) {
        int ret = 0;
        for (int i = 0; timeSeries.size() && i < timeSeries.size() - 1; ++i)
            ret += timeSeries[i + 1] - timeSeries[i] < duration ? timeSeries[i +
1] - timeSeries[i] : duration;
        if (timeSeries.size())
            ret += duration;
        return ret;
    }
};
```

442. Find All Duplicates in an Array

Description

Solution

The idea is to make the best of the range of `a[i]`. Hence, we can use negation of `nums[abs(nums[i]) - 1]`th element to find the duplicate elements

Time complexity: O(N) vs. Space complexity: O(1).

```
class Solution {
public:
    vector<int> findDuplicates(vector<int>& nums) {
        vector<int> ret;
        for (int i = 0; i < nums.size(); ++i) {
            nums[abs(nums[i]) - 1] *= -1;
            if (nums[abs(nums[i]) - 1] > 0)
                ret.push_back(abs(nums[i]));
        }
        return ret;
    }
};
```

Solution using `map`:

```
class Solution {
public:
    vector<int> findDuplicates(vector<int>& nums) {
        map<int, int> m;
        vector<int> ret;
        for (int i = 0; i < nums.size(); ++i)
            ++m[nums[i]];
        for (auto i = m.begin(); i != m.end(); ++i)
            if (i->second == 2)
                ret.push_back(i->first);
        return ret;
    }
};
```

Solution using `sort`:

```
class Solution {
public:
    vector<int> findDuplicates(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<int> ret;
        for (int i = 0, j = 0; nums.size() && i < nums.size() - 1;) {
            for (j = 0; j < nums.size() && nums[i] == nums[i + j]; ++j);
            if (j == 2)
                ret.push_back(nums[i]);
            i += j;
        }
        return ret;
    }
};
```

409. Longest Palindrome

Description

Solution

The basic idea is to use hash table to get the frequency of each letter in the string. If the frequency is even, then the corresponding letter is considered as a candidate to form the longest palindrome string. If it is odd, however, we can take away one letter to make it even. At the end, pick one of the letters taken away into the middle of the string.

For example, given a string `"aaaaabbbccdde"`. We can get the following: `hash['a'] = 5`, `hash['b'] = 3`, `hash['c'] = 2`, `hash['d'] = 2`, and `hash['e'] = 1`. Pick all the even occurrence letters, `"cc"` and `"dd"`, and taking away one letter from odd occurrence letters gives `"aaaa"`, `"bb"`. As a result, the string candidates are `["aaaa", "bb", "cc", "dd"]`, and the leftover becomes `['a', 'b', 'e']`. Thus, we can form a palindrome string from candidate letters, say `"aabcddcbaa"`. At last, pick any one letter in the leftover, say `'e'`, and putting it into the middle of the string yields `"aabcdedcbaa"`.

Below is the code:

```
#define N 128
int longestPalindrome(char* s) {
    int hash[N] = {0}, res = 0, odd = 0;
    for (int i = 0; s[i]; hash[s[i++]]++) {}
    for (int i = 0; i < N; i++)
        if (hash[i] % 2)
            odd = 1, res += hash[i] - 1;
        else
            res += hash[i];
    return res + odd;
}
```

429. N-ary Tree Level Order Traversal

Description

Solution

The idea is to use recursion. The key is to merge vectors `ret` and `r`:

1. It is apparent that the `child` node is always **one** level lower than its parent node. Suppose `r = levelOrder(root->child)`, and `ret` a vector to store the final output for current level. Hence, `r[0]`, `r[1]`, `r[2]`, ..., should be merged to `ret[1]`, `ret[2]`, `ret[3]`, ..., respectively.
2. If `ret[1]` does NOT exist, i.e., `ret.size() == 1`, then `ret.push_back(r[0])`. Otherwise, combine `ret[1]` with `r[0]` by `ret[1].insert(ret[1].end(), r[0].begin(), r[0].end());`. To illustrate this, see below:

```
level   N   N+1  N+2  N+3  N+4  N+5
ret = [[1], [2], [3]]                              N    N+1    N+2   N+3  N+4
 N+5
                                          -->  ret = [[1], [2,4], [3,5], [6],
[7], [8]]
r   =      [[4], [5], [6], [7], [8]]
```

3. Repeat the above until all nodes are traversed.

```
/*
// Definition for a Node.
class Node {
public:
    int val = NULL;
    vector<Node*> children;

    Node() {}

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> ret;
        if (!root)
            return ret;
        ret.push_back(vector<int> {root->val});
        for (auto child : root->children) {
            vector<vector<int>> r = levelOrder(child);
            for (int i = 0; i < r.size(); ++i)
                if (i + 1 > ret.size() - 1)
                    ret.push_back(r[i]);
                else
                    ret[i + 1].insert(ret[i + 1].end(), r[i].begin(),
r[i].end());
        }
        return ret;
    }
};
```

350. Intersection of Two Arrays II

Description

Solution

There are two ways to work on this problem.

1. Sort two arrays first and then compare each elements of the sorted arrays.
2. Using hash table to record the elements of one of the arrays, then traverse the elements in the other array and check whether the element is in the hash table: if the element is indeed in the hash table

(there exists a key and the value/count is greater than 0), then update the value/count in the hash table and append the element into resulting array.

Solution based on the built-in function `qsort()` with space complexity: O(1), time complexity: O(N log N):

```
/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
int comparator(const void *a, const void *b) {
    return  *(int *) a > *(int *) b;
}

int* intersect(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
returnSize) {
    qsort(nums1, nums1Size, sizeof(int), comparator);
    qsort(nums2, nums2Size, sizeof(int), comparator);
    *returnSize = 0;
    int *res = malloc(nums1Size * sizeof(int));
    for (int i = 0, j = 0; i < nums1Size && j < nums2Size; ) {
        if (nums1[i] == nums2[j]) {
            res[(*returnSize)++] = nums1[i];
            i++, j++;
        }
        else if (nums1[i] < nums2[j])
            i++;
        else
            j++;
    }
    return realloc(res, *returnSize * sizeof(int));
}
```

Solution using hash table with space complexity: O(N), time complexity: O(N):

```
typedef struct linkedlist {
    int key;
    int val;
    struct linkedlist *next;
} node;

void init(node *head, int key, int val, node *next) {
    head->key = key;
    head->val = val;
    head->next = next;
}

void insert(node *head, int key, int val) {
```

```
    node* cur = head;

    if (key == cur->key)
        cur->val += val;
    for (cur = head; key > cur->key; cur = cur->next) {
        if (cur->next && key > cur->next->key)
            continue;
        else if (cur->next && key == cur->next->key)
            cur->next->val += val;
        else {
            node *new = (node *) malloc(sizeof(node));
            init(new, key, val, cur->next);
            cur->next = new;
        }
    }
}

int search(node *head, int key) {
    node *cur = head;

    while (cur && key != cur->key)
        cur = cur->next;

    if (cur && key == cur->key && cur->val > 0) {
        cur->val--;
        return 1;
    }
    else
        return 0;
}


/**
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
#define N 10000
int* intersect(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
returnSize) {
    node *hash = malloc(N * sizeof(node));
    for (int i = 0; i < N; init(&hash[i], i, 0, NULL), i++) {}
    for (int i = 0; i < nums2Size; insert(&hash[nums2[i] % N], nums2[i], 1),
i++) {}

    int *res = malloc(nums1Size * sizeof(int));
    *returnSize = 0;
    for (int i = 0; i < nums1Size; i++)
        if (search(&hash[nums1[i] % N], nums1[i]))
            res[(*returnSize)++] = nums1[i];
```

```
    free(hash);
    return realloc(res, *returnSize * sizeof(int));
}
```

345. Reverse Vowels of a String

Description

Solution

Solution without using two pointer:

1. Use two arrays `vows`, `vowsIndex` to record the vowels and the corresponding index in the string (one `for` loop).
2. Loop through the `vows`, and replace the original vowels.

Time complexity: O(N) vs. Space complexity: O(M), where N is the size of the string, and M is the number of vowels in the string.

```
char* reverseVowels(char* s) {
    int sSize = strlen(s), cnt = 0;
    int *vowsIndex = malloc(sSize * sizeof(int));
    char *vows = malloc((sSize + 1) * sizeof(char));
    vows[sSize] = '\0';
    for (int i = 0; s[i]; i++) {
        switch (s[i]) {
            case 'a': case 'e': case 'i': case 'o': case 'u':
            case 'A': case 'E': case 'I': case 'O': case 'U':
                vows[cnt] = s[i];
                vowsIndex[cnt++] = i;
                break;
        }
    }
    for (int i = 0; i < cnt; i++)
        s[vowsIndex[i]] = vows[cnt - i - 1];
    free(vowsIndex), free(vows);
    return s;
}
```

Solution using two pointers:

1. One pointer `l` starts from the beginning of the string and the other `r` starts from the end of string.
2. Two pointers move inwards until next closest vowels, and swap them.
3. Keep step 2 going until two pointers meet up.

Time complexity: O(N) vs. Space complexity: O(1).

```
char* reverseVowels(char* s) {
    for (int l = 0, r = strlen(s) - 1; l < r; l++, r--) {
        while (s[l] != 'a' && s[l] != 'e' && s[l] != 'i' && s[l] != 'o' && s[l]
!= 'u' &&
               s[l] != 'A' && s[l] != 'E' && s[l] != 'I' && s[l] != 'O' && s[l]
!= 'U')
            l++;
        while (s[r] != 'a' && s[r] != 'e' && s[r] != 'i' && s[r] != 'o' && s[r]
!= 'u' &&
               s[r] != 'A' && s[r] != 'E' && s[r] != 'I' && s[r] != 'O' && s[r]
!= 'U')
            r--;
        if (l < r) {
            char tmp = s[l];
            s[l] = s[r];
            s[r] = tmp;
        }
    }
    return s;
}
```

274. H-Index

Description

Solution

The idea is to sort the array `citations` in descending order, then apply the definition to find the h-index.

Time complexity: O(N log(N)) vs. Space comlexity: O(1).

```
int comparator(const void *a, const void *b) {
    return *(int *) a < *(int *) b;
}

int hIndex(int* citations, int citationsSize) {
    qsort(citations, citationsSize, sizeof(int), comparator);
    int h;
    for (h = 0; h < citationsSize && h < citations[h]; h++);
    return h;
}
```

Update: For the above `for` loop, whose worst case runtime is O(N). Instead, we can reduce the worst case runtime to O(log(N)) by applying binary search. However, the runtime of `qsort()` is still dorminant, hence the time complexity remains the same level as above, though it should be slightly faster if the citation is long enough.

Time complexity: O(N log(N)) vs. Space comlexity: O(1)

```c
int comparator(const void *a, const void *b) {
    return *(int *) a < *(int *) b;
}

int hIndex(int* citations, int citationsSize) {
    qsort(citations, citationsSize, sizeof(int), comparator);
    int low = 0, high = citationsSize;
    while (low < high) {
        int mid = (low + high) >> 1;
        if (mid < citations[mid])
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
```

256. Paint House

Description

```
There are a row of n houses, each house can be painted with one of the three
colors: red, blue or green. The cost of painting each house with a certain color
is different. You have to paint all the houses such that no two adjacent houses
have the same color.

The cost of painting each house with a certain color is represented by a n x 3
cost matrix. For example, costs[0][0] is the cost of painting house 0 with color
red; costs[1][2] is the cost of painting house 1 with color green, and so on...
Find the minimum cost to paint all houses.

Note:
All costs are positive integers.

Example:

Input: [[17,2,17],[16,16,5],[14,3,19]]
Output: 10
Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2
into blue.
             Minimum cost: 2 + 5 + 3 = 10.
```

Solution

01/14/2020 (Dynamic Programming):

```
class Solution {
public:
  int minCost(vector<vector<int>>& costs) {
    for (int i = 1; i < costs.size(); ++i) {
      for (int j = 0; j < costs[0].size(); ++j) {
        int cur_min = INT_MAX;
        for (int k = 0; k < costs[0].size(); ++k)
          if (k != j)
            cur_min = min(cur_min, costs[i - 1][k]);
        costs[i][j] += cur_min;
      }
    }
    return costs.size() > 0 ? *min_element(costs.back().begin(),
costs.back().end()) : 0;
  }
};
```

238. Product of Array Except Self

Description

Solution

Trivial solution using division:

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> ret;
        set<int> zeros;
        long prod = 1;
        for (int i = 0; i < nums.size(); ++i)
            if (nums[i] == 0)
                zeros.insert(i);
            else
                prod *= nums[i];
        for (int i = 0; i < nums.size(); ++i)
            if (zeros.size() == 0)
                ret.push_back(prod / nums[i]);
            else if (zeros.size() == 1 && zeros.count(i))
                ret.push_back(prod);
            else
                ret.push_back(0);
        return ret;
    }
};
```

Solution without division:

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> ret(nums.size(), 1);
        for (int i = 1; i < nums.size(); ++i)
            ret[i] = ret[i - 1] * nums[i - 1];
        int r_prod = 1;
        for (int i = nums.size() - 1; i >= 0; --i) {
            ret[i] *= r_prod;
            r_prod *= nums[i];
        }
        return ret;
    }
};
```

219. Contains Duplicate II

Description

Solution

```
typedef struct linkedlist {
    int key;
    int val;
    struct linkedlist *next;
} node;

void init(node *head, int key, int val, node *next) {
    head->key = key;
    head->val = val;
    head->next = next;
}

void insert(node *head, int key, int val) {
    node* cur = head;
    if (key == cur->key)
        cur->val += val;
    for (cur = head; key != cur->key; cur = cur->next) {
        if (cur->next && key > cur->next->key)
            continue;
        else if (cur->next && key == cur->next->key)
            cur->next->val += val;
        else {
            node *new = (node *) malloc(sizeof(node));
            init(new, key, val, cur->next);
            cur->next = new;
        }
    }
}

int search(node *head, int key) {
    node *cur = head;
    while (cur && key != cur->key)
        cur = cur->next;
    if (cur && key == cur->key && cur->val > 1) {
        cur->val--;
        return 1;
    }
    else
        return 0;
}

#define N 2503
```

```
bool containsNearbyDuplicate(int* nums, int numsSize, int k) {
    node *hash = malloc(N * sizeof(node));
    for (int i = 0; i < N; init(&hash[i], i, 0, NULL), ++i);
    for (int i = 0; i < numsSize; insert(&hash[(nums[i] % N + N) % N], nums[i],
1), ++i);
    for (int i = 0; i < numsSize; ++i)
        if (search(&hash[(nums[i] % N + N) % N], nums[i]))
            for (int j = 1; i + j < numsSize && j <= k; ++j)
                if (nums[i + j] == nums[i])
                    return true;
    return false;
}
```

200. Number of Islands

Description

Solution

The idea is to use DFS to identify the island. Once a part of the island has been found, say `grid[i][j] == '1'`, then use `dfs(grid, i, j)` to set the rest of island to be '0'.

C++ (8 ms, 98.51%):

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int ret = 0;
        for (int i = 0; i < grid.size(); ++i)
            for (int j = 0; j < grid[0].size(); ++j)
                if (grid[i][j] == '1')
                    ++ret, dfs(grid, i, j);
        return ret;
    }
    void dfs(vector<vector<char>>& grid, int i, int j) {
        if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i]
[j] == '0')
            return;
        grid[i][j] = '0';
        dfs(grid, i - 1, j), dfs(grid, i + 1, j), dfs(grid, i, j - 1), dfs(grid,
i, j + 1);
    }
};
```

C (4 ms, beats 100%):

```c
void dfs(char** grid, int m, int n, int i, int j) {
    if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == '0')
        return;
    grid[i][j] = '0';
    dfs(grid, m, n, i - 1, j);
    dfs(grid, m, n, i + 1, j);
    dfs(grid, m, n, i, j - 1);
    dfs(grid, m, n, i, j + 1);
}
int numIslands(char** grid, int gridRowSize, int gridColSize) {
    int ret = 0;
    for (int i = 0; i < gridRowSize; ++i)
        for (int j = 0; j < gridColSize; ++j)
            if (grid[i][j] == '1')
                ++ret, dfs(grid, gridRowSize, gridColSize, i, j);
    return ret;
}
```

198. House Robber

Description

Solution

The idea is to use Dynamic Programming. We only need to consider skipping one or two houses. If it is three, we can absolutely rob the one in the middle, hence narrows down to the case of skipping one house. Therefore, the recurrence is `D[n] = max(D[n - 2] + nums[n - 2], D[n - 3] + nums[n - 3])`, and the base case is `D[0] = 0, D[1] = 0`.

Time complexity: O(N) vs. Space complexity: O(N).

```c
#define max(x, y) (x) > (y) ? (x) : (y)
int rob(int* nums, int numsSize) {
    int *D = calloc((numsSize + 2), sizeof(int));
    for (int i = 2; i < numsSize + 2; i++)
        D[i] = max(D[i - 2] + nums[i - 2], D[i - 3] + nums[i - 3]);
    return max(D[numsSize + 1], D[numsSize]);
}
```

Update: Time complexity: O(N) vs. Space complexity: O(1).

```
#define max(x, y) (x) > (y) ? (x) : (y)
int rob(int* nums, int numsSize) {
    for (int i = 2; i < numsSize; i++)
        nums[i] += max(nums[i - 2], nums[i - 3]);
    return numsSize > 0 ? max(nums[numsSize - 1], nums[numsSize - 2]) : 0;
}
```

169. Majority Element

Description

Solution

The idea is

- Pair up the elements of `nums` arbitrarily, to get `n/2` pairs.
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them.
- Put these "survived" element in a new array which must contain the majority element.
- Do this over and over again, we can get the majority element.
- However, when `n` is odd, we need to pay close attention to the last element in the array, check the frequency of this element, if its frequency is greater than `n/2`, then it is the majority element, done. Otherwise, discard this element.

Time complexity: O(N) vs. Space complexity: O(N).

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        if (n == 1)
            return nums[0];
        if (n % 2 == 1) {
            int count = 1;
            for (int i = 0; i < n - 1; ++i)
                if (nums[n-1] == nums[i])
                    ++count;
            if (count > n / 2)
                return nums[n-1];
        }
        vector<int> numsTmp;
        for (int i = 0; i < n / 2; ++i)
            if (nums[2 * i] == nums[2 * i + 1])
                numsTmp.push_back(nums[2 * i]);
        return majorityElement(numsTmp);
    }
```

```
};
```

102. Binary Tree Level Order Traversal

Description

Solution

DFS solution (16 ms): The idea is same as this post. But seems not very effiecient... any suggestion to make it faster?

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ret;
        if (!root)
            return ret;
        ret.push_back({root->val});
        if (root->left) {
            vector<vector<int>> l = levelOrder(root->left);
            for (int i = 0; i < l.size(); ++i)
                if (i + 1 > ret.size() - 1)
                    ret.push_back(l[i]);
                else
                    ret[i + 1].insert(ret[i + 1].end(), l[i].begin(),
l[i].end());
        }
        if (root->right) {
            vector<vector<int>> r = levelOrder(root->right);
            for (int i = 0; i < r.size(); ++i)
                if (i + 1 > ret.size() - 1)
                    ret.push_back(r[i]);
                else
                    ret[i + 1].insert(ret[i + 1].end(), r[i].begin(),
r[i].end());
        }
        return ret;
    }
};
```

More concise but slower code (40 ms):

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
```

```
        vector<vector<int>> ret;
        if (!root)
            return ret;
        ret.push_back({root->val});
        vector<vector<int>> l = levelOrder(root->left);
        vector<vector<int>> r = levelOrder(root->right);
        merge_vector(&ret, l);
        merge_vector(&ret, r);
        return ret;
    }

    void merge_vector(vector<vector<int>> *ret, vector<vector<int>> tmp) {
         for (int i = 0; i < tmp.size(); ++i)
            if (i + 1 > (*ret).size() - 1)
                (*ret).push_back(tmp[i]);
            else
                (*ret)[i + 1].insert((*ret)[i + 1].end(), tmp[i].begin(),
tmp[i].end());
    }
};
```

BFS solution (4 ms):

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ret;
        if (!root)
            return ret;
        queue<TreeNode *> q;
        q.push(root);
        while (!q.empty()) {
            vector<int> r;
            int n = q.size();
            for (int i = 0; i < n; ++i) {
                auto cur = q.front(); q.pop();
                r.push_back(cur->val);
                if (cur->left)
                    q.push(cur->left);
                if (cur->right)
                    q.push(cur->right);
            }
            ret.push_back(r);
        }
        return ret;
    }
```

```
};
```

73. Set Matrix Zeroes

Description

Solution

The idea is to use two arrays, `row` and `col`, to record rows and columns that has at zeros. And then set the corresponding rows and columns to zero according to `row` and `col`.

Time complexity: O(N * M) vs. Space complexity: O(N + M), where N and M represent the number of rows and columns of the given matrix, respectively.

```
void setZeroes(int** matrix, int matrixRowSize, int matrixColSize) {
    int *row = calloc(matrixRowSize, sizeof(int));
    int *col = calloc(matrixColSize, sizeof(int));
    for (int i = 0; i < matrixRowSize; i++)
        for (int j = 0; j < matrixColSize; j++)
            if (matrix[i][j] == 0)
                row[i] = 1, col[j] = 1;
    for (int i = 0; i < matrixRowSize; i++)
        for (int j = 0; j < matrixColSize; j++)
            if (row[i] == 1 || col[j] == 1)
                matrix[i][j] = 0;
    free(row), free(col);
}
```

69. Sqrt(x)

Description

Solution

```c
int mySqrt(int x) {
    int low = 0, high = x;
    while (low < high) {
        long mid = (low + high) >> 1;
        if (mid * mid <= x)
            low = mid + 1;
        else
            high = mid;
    }
    return x > 1 ? low - 1 : x;
}
```

Update: Optimized the code a little bit, as a result, it outperformed the above, and performed just as good as applying the built-in function `sqrt()`.

```c
int mySqrt(int x) {
    int low = 0, high = x;
    while (low < high) {
        long mid = (low + high) / 2 + 1;
        if (mid * mid > x)
            high = mid - 1;
        else
            low = mid;
    }
    return low;
}
```

54. Spiral Matrix

Description

Solution

Here are steps:

1. Mark the visited entry `matrix[i][j]` by `INT_MIN`;
2. Update the indices `i` and `j` according to the current direction. If either the updated index `ni = i + d[k][0]` or `nj = i + d[k][1]` is out of bound or the new entry `matrix[ni][nj]` has been visited, then turn right 90 degrees (`k = (k + 1) % 4`);
3. Follow the above steps until the number of steps reaches the total number of entries `matrixRowSize * matrixColSize`.

Time complexity: O(M * N) vs. Space complexity: O(1).

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* spiralOrder(int** matrix, int matrixRowSize, int matrixColSize) {
    int d[][2] = { {0, 1}, {1, 0}, {0, -1}, {-1, 0}}, n = matrixRowSize *
matrixColSize;
    int *ret = malloc(n * sizeof(int));
    for (int i = 0, j = 0, k = 0, cnt = 0; cnt < n; cnt++) {
        ret[cnt] = matrix[i][j], matrix[i][j] = INT_MIN;
        int ni = i + d[k][0], nj = j + d[k][1];
        if (ni < 0 || ni > matrixRowSize - 1 ||     // if the updated i is out
of bound
            nj < 0 || nj > matrixColSize - 1 ||     // if the updated j is out
of bound
            matrix[ni][nj] == INT_MIN)              // if the new matrix entry
has been visited
            k = (k + 1) % 4;
        i = i + d[k][0], j = j + d[k][1];
    }
    return ret;
}
```

33. Search in Rotated Sorted Array

Description

Solution

The idea is to do the binary search when `target` and the `nums[mid]` in the same sorted section. If they are not, then adjust `mid` until `nums[mid]` is. Time complexity: O(log(N)) vs. Space complexity: O(1).

```
int search(int* nums, int numsSize, int target) {
    int low = 0, high = numsSize - 1;
    while (low <= high) {
        int mid = (low + high) >> 1;
        if (target >= nums[0] && nums[mid] < nums[0])
            high = mid - 1;
        else if (target < nums[0] && nums[mid] >= nums[0])
            low = mid + 1;
        else  if (nums[mid] == target)
            return mid;
        else if (nums[mid] > target)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

19. Remove Nth Node From End of List

Description

Solution

The idea is to move the `fast` pointer to the n-th node from the head node first, then move both `slow` and `fast` pointers at the same pace until `fast` reaches to the end of the linked list, at which `slow` is just right before the node to be removed. Remove the node by `slow->next = slow->next->next`. Done.

Time complexity: O(N) vs. Space complexity: O(1).

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeNthFromEnd(struct ListNode* head, int n) {
    struct ListNode *fast, *slow;
    for (fast = head; fast && fast->next && n > 0; fast = fast->next, n--);
    for (slow = head; fast && fast->next; fast = fast->next, slow = slow->next);
    if (slow && n == 0)
        slow->next = slow->next->next;
```

```
    return n == 1 ? head->next : head;
}
```

13. Roman to Integer

Description

Solution

The idea is very simple: just sum up the value of each symbol and change the sign if it appears before the symbol that is greater than itself, e.g., `I` before `V` or `X`.

```
int romanToInt(char* s) {
    int ret = 0;
    for (int i = 0; s[i]; i++) {
        switch (s[i]) {
            case 'I': ret += s[i + 1] && (s[i + 1] == 'V' || s[i + 1] == 'X') ?
-1 : 1; break;
            case 'X': ret += s[i + 1] && (s[i + 1] == 'L' || s[i + 1] == 'C') ?
-10 : 10; break;
            case 'C': ret += s[i + 1] && (s[i + 1] == 'D' || s[i + 1] == 'M') ?
-100 : 100; break;
            case 'V': ret += 5; break;
            case 'L': ret += 50; break;
            case 'D': ret += 500; break;
            case 'M': ret += 1000; break;
        }
    }
    return ret;
}
```

11. Container With Most Water

Description

Solution

The idea is straightforward: use two loops to find out the maximum area.

Time complexity: O(N^2) vs. Space complexity: O(1).

```cpp
class Solution {
public:
    int maxArea(vector<int>& height) {
        int ret = 0;
        for (int i = 0; i < height.size() - 1; ++i)
            for (int j = height.size() - 1; j > i; --j)
                ret = max(ret, min(height[i], height[j]) * (j - i));
        return ret;
    }
};
```