【第三十九课】金融系统中的 RSA 算法（三）

2119. 反转两次的数字

反转 一个整数意味着倒置它的所有位。

- 例如，反转 `2021` 得到 `1202` 。反转 `12300` 得到 `321` ，**不保留前导零** 。

给你一个整数 `num` ，**反转** `num` 得到 `reversed1` ，**接着反转** `reversed1` 得到 `reversed2` 。如果 `reversed2` 等于 `num` ，返回 `true` ；否则，返回 `false` 。

示例：

```
输入：num = 526
输出：true
解释：反转 num 得到 625 ，接着反转 625 得到 526 ，等于 num 。
```

```cpp
class Solution {
public:
    long long getNum(long long x) {
        long long y = 0;
        while (x) {
            y = y * 10 + x % 10;
            x /= 10;
        }
        return y;
    }
    bool isSameAfterReversals(int num) {
        return getNum(getNum(num)) == num;
    }
};
```

2110. 股票平滑下跌阶段的数目

给你一个整数数组 `prices` ，表示一支股票的历史每日股价，其中 `prices[i]` 是这支股票第 `i` 天的价格。

一个 **平滑下降的阶段** 定义为：对于 **连续一天或者多天** ，每日股价都比 **前一日股价恰好少** `1` ，这个阶段第一天的股价没有限制。

请你返回 **平滑下降阶段** 的数目。

示例 1：

```
输入：prices = [3,2,1,4]
输出：7
解释：总共有 7 个平滑下降阶段：
[3], [2], [1], [4], [3,2], [2,1] 和 [3,2,1]
注意，仅一天按照定义也是平滑下降阶段。
```

```cpp
class Solution {
public:
    long long getDescentPeriods(vector<int>& prices) {
        long long fi = 0, ans = 0, n = prices.size();
        for (int i = 0, pre = 0; i < n; i++) {
            if (prices[i] + 1 == pre) fi += 1;
            else fi = 1;
            ans += fi;
            pre = prices[i];
        }
        return ans;
    }
};
```

2140. 解决智力问题

给你一个下标从 **0** 开始的二维整数数组 `questions` ，其中 `questions[i] = [pointsi, brainpoweri]` 。

这个数组表示一场考试里的一系列题目，你需要 **按顺序** （也就是从问题 `0` 开始依次解决），针对每个问题选择 **解决** 或者 **跳过** 操作。解决问题 `i` 将让你 **获得** `pointsi` 的分数，但是你将 **无法** 解决接下来的 `brainpoweri` 个问题（即只能跳过接下来的 `brainpoweri` 个问题）。如果你跳过问题 `i` ，你可以对下一个问题决定使用哪种操作。

- 比方说，给你 `questions = [[3, 2], [4, 3], [4, 4], [2, 5]]` ：
  - 如果问题 `0` 被解决了， 那么你可以获得 `3` 分，但你不能解决问题 `1` 和 `2` 。
  - 如果你跳过问题 `0` ，且解决问题 `1` ，你将获得 `4` 分但是不能解决问题 `2` 和 `3` 。

请你返回这场考试里你能获得的 **最高** 分数。

示例 1：

```
输入：questions = [[3,2],[4,3],[4,4],[2,5]]
输出：5
解释：解决问题 0 和 3 得到最高分。
- 解决问题 0 ：获得 3 分，但接下来 2 个问题都不能解决。
- 不能解决问题 1 和 2
- 解决问题 3 ：获得 2 分
总得分为：3 + 2 = 5 。没有别的办法获得 5 分或者多于 5 分。
```

```cpp
class Solution {
public:
    long long mostPoints(vector<vector<int>>& questions) {
        long long n = questions.size(), dp[n + 1];
        dp[n] = 0;
        for (long long i = n - 1; i >= 0; --i) {
            dp[i] = max(dp[i + 1], dp[min(i + questions[i][1] + 1, n)] + questions[i][0]);
        }
        return dp[0];
    }
};
```

2134. 最少交换次数来组合所有的 1 II

交换 定义为选中一个数组中的两个 **互不相同** 的位置并交换二者的值。

环形 数组是一个数组，可以认为 **第一个** 元素和 **最后一个** 元素 **相邻** 。

给你一个 **二进制环形** 数组 `nums` ，返回在 **任意位置** 将数组中的所有 `1` 聚集在一起需要的最少交换次数。

示例 1：

```
输入：nums = [0,1,0,1,1,0,0]
输出：1
解释：这里列出一些能够将所有 1 聚集在一起的方案：
[0,0,1,1,1,0,0] 交换 1 次。
[0,1,1,1,0,0,0] 交换 1 次。
[1,1,0,0,0,0,1] 交换 2 次（利用数组的环形特性）。
无法在交换 0 次的情况下将数组中的所有 1 聚集在一起。
因此，需要的最少交换次数为 1 。
```

```cpp
class Solution {
public:
    int minSwaps(vector<int>& nums) {
        int ans = INT_MAX, l, r, z_cnt = 0, cnt = 0, n = nums.size();
        for (auto x : nums) if (x == 1) cnt += 1;
        for (int i = 0; i < cnt; i++) if (nums[i] == 0) z_cnt += 1;
        l = 0, r = cnt - 1;
        for (int i = 0; i < n; i++) {
            ans = min(ans, z_cnt);
            if (nums[l] == 0) z_cnt -= 1;
            if (nums[(r + 1) % n] == 0) z_cnt += 1;
            l += 1, r += 1;
        }
        return ans;
    }
};
```

2130. 链表最大孪生和

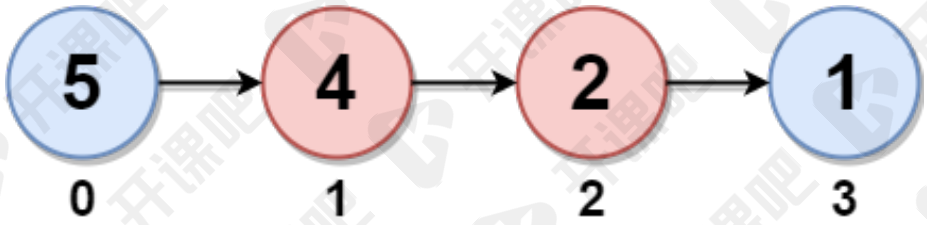
在一个大小为 `n` 且 `n` 为 **偶数** 的链表中，对于 `0 <= i <= (n / 2) - 1` 的 `i` ，第 `i` 个节点（下标从 **0** 开始）的孪生节点为第 `(n-1-i)` 个节点 。

- 比方说，`n = 4` 那么节点 `0` 是节点 `3` 的孪生节点，节点 `1` 是节点 `2` 的孪生节点。这是长度为 `n = 4` 的链表中所有的孪生节点。

孪生和 定义为一个节点和它孪生节点两者值之和。

给你一个长度为偶数的链表的头节点 `head` ，请你返回链表的 **最大孪生和** 。

示例 1：




```
输入：head = [5,4,2,1]
输出：6
解释：
节点 0 和节点 1 分别是节点 3 和 2 的孪生节点。孪生和都为 6 。
链表中没有其他孪生节点。
所以，链表的最大孪生和是 6 。
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode *node;
    int getMaxValue(ListNode *head, ListNode *q) {
        if (q == nullptr) {
            node = head;
            return INT_MIN;
        }
        int ans = getMaxValue(head, q->next);
        ans = max(ans, node->val + q->val);
        node = node->next;
        return ans;
```

```
    }
    int pairSum(ListNode* head) {
        ListNode *p = head, *q = head;
        while (p) {
            p = p->next->next;
            q = q->next;
        }
        return getMaxValue(head, q);
    }
};
```

2096. 从二叉树一个节点到另一个节点每一步的方向

给你一棵 **二叉树** 的根节点 `root` ，这棵二叉树总共有 `n` 个节点。每个节点的值为 `1` 到 `n` 中的一个整数，且互不相同。给你一个整数 `startValue` ，表示起点节点 `s` 的值，和另一个不同的整数 `destValue` ，表示终点节点 `t` 的值。

请找到从节点 `s` 到节点 `t` 的 **最短路径** ，并以字符串的形式返回每一步的方向。每一步用 **大写** 字母 `'L'` ， `'R'` 和 `'U'` 分别表示一种方向：

- `'L'` 表示从一个节点前往它的 **左孩子** 节点。
- `'R'` 表示从一个节点前往它的 **右孩子** 节点。
- `'U'` 表示从一个节点前往它的 **父** 节点。

请你返回从 `s` 到 `t` **最短路径** 每一步的方向。

示例 1：

```
输入：root = [5,1,2,3,null,6,4], startValue = 3, destValue = 6
输出："UURL"
解释：最短路径为：3 → 1 → 5 → 2 → 6 。
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    char buff[500000];
```

```
    void getPathString(TreeNode *root, int k, int a, int b, string &a_str, string &b_str, char *buff) {
        buff[k] = 0;
        if (root == nullptr) {
            return ;
        }
        if (root->val == a) a_str = buff;
        if (root->val == b) b_str = buff;
        buff[k] = 'L';
        getPathString(root->left,  k + 1, a, b, a_str, b_str, buff);
        buff[k] = 'R';
        getPathString(root->right, k + 1, a, b, a_str, b_str, buff);
        return ;
    }
    string getDirections(TreeNode* root, int startValue, int destValue) {
        string s_str, d_str;
        getPathString(root, 0, startValue, destValue, s_str, d_str, buff);
        int i = 0;
        while (s_str[i] && s_str[i] == d_str[i]) i += 1;
        s_str = s_str.substr(i, s_str.size());
        d_str = d_str.substr(i, d_str.size());
        for (i = 0; s_str[i]; i++) s_str[i] = 'U';
        return s_str + d_str;
    }
};
```

2136. 全部开花的最早一天

你有 `n` 枚花的种子。每枚种子必须先种下，才能开始生长、开花。播种需要时间，种子的生长也是如此。给你两个下标从 **0** 开始的整数数组 `plantTime` 和 `growTime` ，每个数组的长度都是 `n` ：

- `plantTime[i]` 是 **播种** 第 `i` 枚种子所需的 **完整天数** 。每天，你只能为播种某一枚种子而劳作。**无须** 连续几天都在种同一枚种子，但是种子播种必须在你工作的天数达到 `plantTime[i]` 之后才算完成。
- `growTime[i]` 是第 `i` 枚种子完全种下后生长所需的 **完整天数** 。在它生长的最后一天 **之后** ，将会开花并且永远 **绽放** 。

从第 `0` 开始，你可以按 **任意** 顺序播种种子。

返回所有种子都开花的 **最早** 一天是第几天。

示例 1：

```
输入：plantTime = [1,4,3], growTime = [2,3,1]
输出：9
解释：灰色的花盆表示播种的日子，彩色的花盆表示生长的日子，花朵表示开花的日子。
一种最优方案是：
第 0 天，播种第 0 枚种子，种子生长 2 整天。并在第 3 天开花。
第 1、2、3、4 天，播种第 1 枚种子。种子生长 3 整天，并在第 8 天开花。
第 5、6、7 天，播种第 2 枚种子。种子生长 1 整天，并在第 9 天开花。
因此，在第 9 天，所有种子都开花。
```

```cpp
class Solution {
public:
    int earliestFullBloom(vector<int>& plantTime, vector<int>& growTime) {
        int ans = INT_MIN, n = plantTime.size();
        vector<int> ind(n);
        for (int i = 0; i < n; i++) ind[i] = i;
        sort(ind.begin(), ind.end(), [&](int i, int j) -> bool { return growTime[i] > growTime[j]; });
        for (int i = 0, sum = 0; i < n; i++) {
            sum += plantTime[ind[i]];
            ans = max(ans, sum + growTime[ind[i]]);
        }
        return ans;
    }
};
```

2104. 子数组范围和

给你一个整数数组 `nums` 。`nums` 中，子数组的 **范围** 是子数组中最大元素和最小元素的差值。

返回 `nums` 中 **所有** 子数组范围的 **和** 。

子数组是数组中一个连续 **非空** 的元素序列。

示例 1：

```
输入：nums = [1,2,3]
输出：4
解释：nums 的 6 个子数组如下所示：
[1]，范围 = 最大 - 最小 = 1 - 1 = 0
[2]，范围 = 2 - 2 = 0
[3]，范围 = 3 - 3 = 0
[1,2]，范围 = 2 - 1 = 1
[2,3]，范围 = 3 - 2 = 1
[1,2,3]，范围 = 3 - 1 = 2
所有范围的和是 0 + 0 + 0 + 1 + 1 + 2 = 4
```

```cpp
class Solution {
public:
```

```
    long long getValue(deque<int> &q_min, deque<int> &q_max, vector<int> &nums) {
        auto p = q_min.begin(), q = q_max.begin();
        int pre_pos = -1;
        long long ans = 0;
        while (p != q_min.end()) {
            int pos = min(*p, *q);
            ans += (long long)(pos - pre_pos) * (long long)(nums[*q] - nums[*p]);
            if (*p == pos) p++;
            if (*q == pos) q++;
            pre_pos = pos;
        }
        return ans;
    }
    long long subArrayRanges(vector<int>& nums) {
        int n = nums.size();
        long long ans = 0;
        deque<int> q_min, q_max;
        for (int i = 0; i < n; i++) {
            while (!q_min.empty() && nums[i] <= nums[q_min.back()])
q_min.pop_back();
            while (!q_max.empty() && nums[i] >= nums[q_max.back()])
q_max.pop_back();
            q_min.push_back(i), q_max.push_back(i);
            ans += getValue(q_min, q_max, nums);
        }
        return ans;
    }
};
```

1028. 从先序遍历还原二叉树

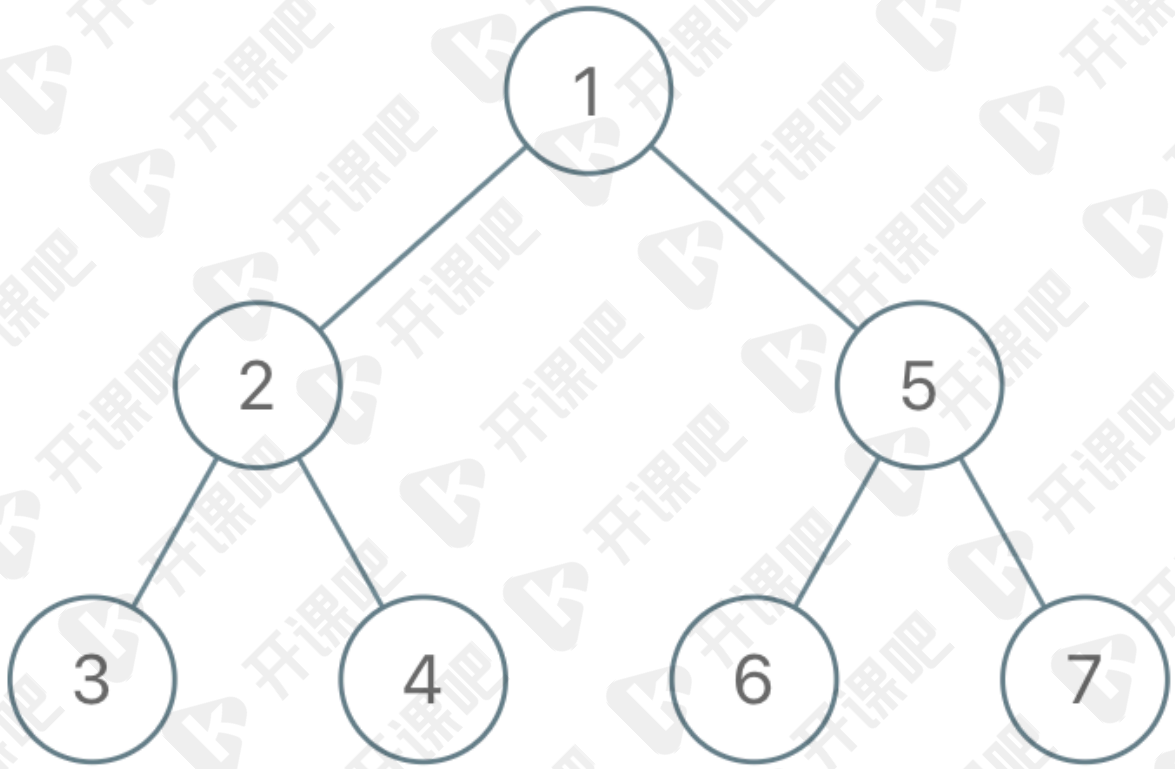
我们从二叉树的根节点 `root` 开始进行深度优先搜索。

在遍历中的每个节点处，我们输出 `D` 条短划线（其中 `D` 是该节点的深度），然后输出该节点的值。（*如果节点的深度为 `D`，则其直接子节点的深度为 `D + 1`。根节点的深度为 `0`*）。

如果节点只有一个子节点，那么保证该子节点为左子节点。

给出遍历输出 `S`，还原树并返回其根节点 `root`。

示例 1：




```
输入："1-2--3--4-5--6--7"
输出：[1,2,5,3,4,6,7]
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* recoverFromPreorder(string traversal) {
        int i = 0, k = 0, num = 0;
        stack<TreeNode *> s;
        TreeNode *p;
        while (traversal[i]) {
            k = num = 0;
            while (traversal[i] == '-') i++, k++;
            while (traversal[i] != '-' && traversal[i]) num = num * 10 +
traversal[i] - '0', i++;
```

```
            while (s.size() > k) s.pop();
            p = new TreeNode(num);
            if (s.size()) {
                if (s.top()->left == nullptr) s.top()->left = p;
                else s.top()->right = p;
            }
            s.push(p);
        }
        while (s.size()) p = s.top(), s.pop();
        return p;
    }
};
```

1125. 最小的必要团队

作为项目经理，你规划了一份需求的技能清单 `req_skills`，并打算从备选人员名单 `people` 中选出些人组成一个「必要团队」（编号为 `i` 的备选人员 `people[i]` 含有一份该备选人员掌握的技能列表）。

所谓「必要团队」，就是在这个团队中，对于所需求的技能列表 `req_skills` 中列出的每项技能，团队中至少有一名成员已经掌握。可以用每个人的编号来表示团队中的成员：

- 例如，团队 `team = [0, 1, 3]` 表示掌握技能分别为 `people[0]`，`people[1]`，和 `people[3]` 的备选人员。

请你返回 **任一** 规模最小的必要团队，团队成员用人员编号表示。你可以按 **任意顺序** 返回答案，题目数据保证答案存在。

示例 1：

```
输入：req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","reactjs"]]
输出：[0,2]
```

```
class Solution {
public:
    void update(string &s, unordered_map<string, int> &indeg, unordered_map<string, unordered_set<string>> &g) {
        for (auto x : g[s]) {
            indeg[x] -= 1;
            if (indeg[x] == 0) update(x, indeg, g);
        }
        return ;
    }
    vector<string> findAllRecipes(vector<string>& recipes, vector<vector<string>>& ingredients, vector<string>& supplies) {
        unordered_map<string, int> indeg;
        unordered_map<string, unordered_set<string>> g;
        int n = recipes.size();
        for (int i = 0; i < n; i++) {
```

```
            indeg[recipes[i]] = ingredients[i].size();
            for (auto x : ingredients[i]) {
                g[x].insert(recipes[i]);
            }
        }
        for (auto x : supplies) {
            indeg[x] = 0;
            update(x, indeg, g);
        }
        vector<string> ret;
        for (auto x : recipes) if (indeg[x] == 0) ret.push_back(x);
        return ret;
    }
};
```

2139. 得到目标值的最少行动次数

你正在玩一个整数游戏。从整数 `1` 开始，期望得到整数 `target` 。

在一次行动中，你可以做下述两种操作之一：

- **递增**，将当前整数的值加 1（即， `x = x + 1`）。
- **加倍**，使当前整数的值翻倍（即，`x = 2 * x`）。

在整个游戏过程中，你可以使用 **递增** 操作 **任意** 次数。但是只能使用 **加倍** 操作 **至多** `maxDoubles` 次。

给你两个整数 `target` 和 `maxDoubles` ，返回从 1 开始得到 `target` 需要的最少行动次数。

示例 1：

```
输入：target = 5, maxDoubles = 0
输出：4
解释：一直递增 1 直到得到 target 。
```

```
class Solution {
public:
    int minMoves(int target, int maxDoubles) {
        int cnt = floor(log2(target)), one_cnt = 0;
        cnt = min(maxDoubles, cnt);
        for (int i = 0; i < cnt; i++) {
            if (target & 1) one_cnt += 1;
            target >>= 1;
        }
        return target - 1 + cnt + one_cnt;
    }
};
```

2121. 相同元素的间隔之和

给你一个下标从 **0** 开始、由 `n` 个整数组成的数组 `arr` 。

`arr` 中两个元素的 **间隔** 定义为它们下标之间的 **绝对差** 。更正式地，`arr[i]` 和 `arr[j]` 之间的间隔是 `|i - j|` 。

返回一个长度为 `n` 的数组 `intervals` ，其中 `intervals[i]` 是 **`arr[i]`** 和 **`arr`** 中每个相同元素（与 `arr[i]` 的值相同）的 **间隔之和** 。

注意： `|x|` 是 `x` 的绝对值。

示例 1：

```
输入：arr = [2,1,3,1,2,3,3]
输出：[4,2,7,2,4,4,5]
解释：
- 下标 0 ：另一个 2 在下标 4 ，|0 - 4| = 4
- 下标 1 ：另一个 1 在下标 3 ，|1 - 3| = 2
- 下标 2 ：另两个 3 在下标 5 和 6 ，|2 - 5| + |2 - 6| = 7
- 下标 3 ：另一个 1 在下标 1 ，|3 - 1| = 2
- 下标 4 ：另一个 2 在下标 0 ，|4 - 0| = 4
- 下标 5 ：另两个 3 在下标 2 和 6 ，|5 - 2| + |5 - 6| = 4
- 下标 6 ：另两个 3 在下标 2 和 5 ，|6 - 2| + |6 - 5| = 5
```

```cpp
class Solution {
public:
    vector<long long> getDistances(vector<int>& arr) {
        int n = arr.size();
        vector<int> ind(n);
        for (int i = 0; i < n; i++) ind[i] = i;
        sort(ind.begin(), ind.end(), [&](int i, int j) -> bool {
            if (arr[i] - arr[j]) return arr[i] < arr[j];
            return i < j;
        });
        vector<long long> ret(n);
        for (int i = 0, j; i < n; i = j + 1) {
            j = i;
            while (j + 1 < n && arr[ind[j + 1]] == arr[ind[i]]) ++j;
            cout << arr[ind[i]] << " " << i << " " << j << endl;
            long long sum_pre = 0, sum_next = 0;
            for (int k = i; k <= j; k++) sum_next += ind[k];
            for (int k = i; k <= j; k++) {
                sum_next -= ind[k];
                ret[ind[k]] = (long long)(2 * k - i - j) * ind[k] - sum_pre + sum_next;
                sum_pre += ind[k];
            }
        }
```

```
        return ret;
    }
};
```