LeetCode 热题 HOT 100 （4）

236. 二叉树的最近公共祖先

思路

(递归) $O(n)$

考虑`p`和`q`这两个节点共有三种情况：
1、`p` 和 `q`在`root`的子树中，且位于两侧。
2、`p = root` 且 `q` 在 `root` 的左或右子树中。
3、`q = root` 且 `p` 在 `root` 的左或右子树中。

考虑在左子树和右子树中查找这两个节点，如果两个节点分别位于左子树和右子树，则最近公共祖先为自己(`root`)，若左子树中两个节点都找不到，说明最低公共祖先一定在右子树中，反之亦然。考虑到二叉树的递归特性，因此可以通过递归来求得。

时间复杂度分析： 需要遍历整颗树，复杂度为 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
         if(!root) return NULL;  //没有找到，返回null
        if(root == p || root == q) return root; //找到其中之一，返回root
        TreeNode* left = lowestCommonAncestor(root->left, p, q);  //返回左子树查找节点
        TreeNode* right = lowestCommonAncestor(root->right,p ,q); //返回右子树查找节点
        if(left && right) return root;
        if(left) return left;
        else return right;
    }
};
```

238. 除自身以外数组的乘积

思路

(前缀积) $O(n)$

最为直接的思路：申请两个数组，一个用来记录每个位置左边的乘积，另一个用来记录它右边的乘积，最后再把两个数组乘起来即可，但是这样的空间复杂度为$O(n)$。

类比于前缀和，我们用一个`p`数组，来存贮`nums[0] * nums[1] *...* nums[i - 1]`。然后从数组末尾开始遍历，用`s`记录数组末尾若干数字的乘积，然后每次更新`p[i]`即可。

具体过程如下：

- 1、遍历整个数组，利用前缀积公式`p[i] = p[i - 1] * nums[i - 1]`，求出`p`数组。
- 2、初始化`s = 1`，倒序遍历数组，每次先执行`p[i] *= s`，然后`s *= nums[i]`。
- 3、最后返回`p`数组。

时间复杂度分析： $O(n)$。

c++代码

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

239. 滑动窗口最大值

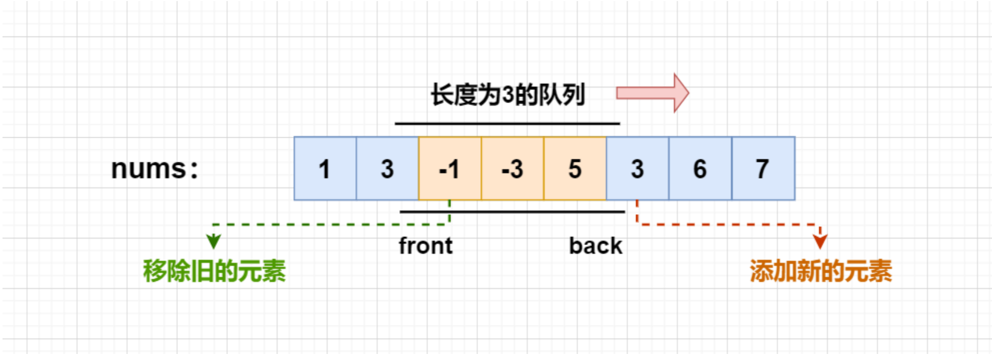
思路

给定一个数组 `nums` 和滑动窗口的大小 `k`，让我们找出所有滑动窗口里的最大值。

样例：

如样例所示，`nums =  [1,3,-1,-3,5,3,6,7]`，`k = 3`，我们输出`[3,3,5,5,6,7]`。

首先，我们可以想到最朴素的做法是模拟滑动窗口的过程，每向右滑动一次都遍历一遍滑动窗口，找到最大的元素输出，这样的时间复杂度是$O(nk)$。考虑优化，其实滑动窗口类似于数据结构双端队列，窗口向右滑动过程相当于向队尾添加新的元素，同时再把队首元素删除。

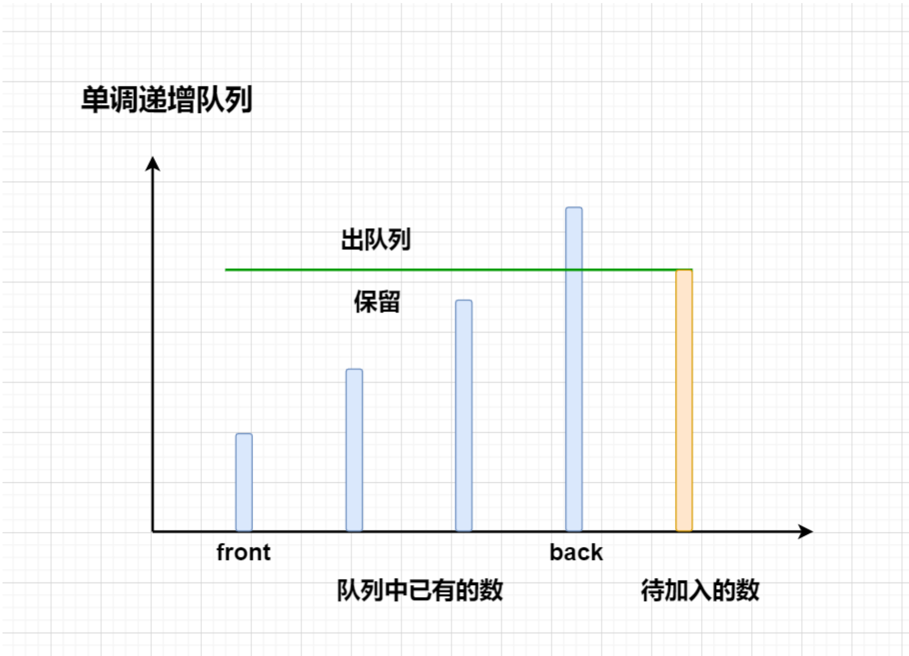


如何更快的找到队列中的最大值?

其实我们可以发现，队列中没必要维护窗口中的所有元素，我们可以在队列中只保留那些可能成为窗口中的最大元素，去掉那些不可能成为窗口中的最大元素。

考虑这样一种情况，如果新进来的数字大于滑动窗口的末尾元素，那么末尾元素就不可能再成为窗口中最大的元素了，因为这个大的数字是后进来的，一定会比之前先进入窗口的小的数字要晚离开窗口，因此我们就可以将滑动窗口中比其小的数字弹出队列，于是队列中的元素就会维持从队头到队尾单调递减，这就是**单调递减队列**。

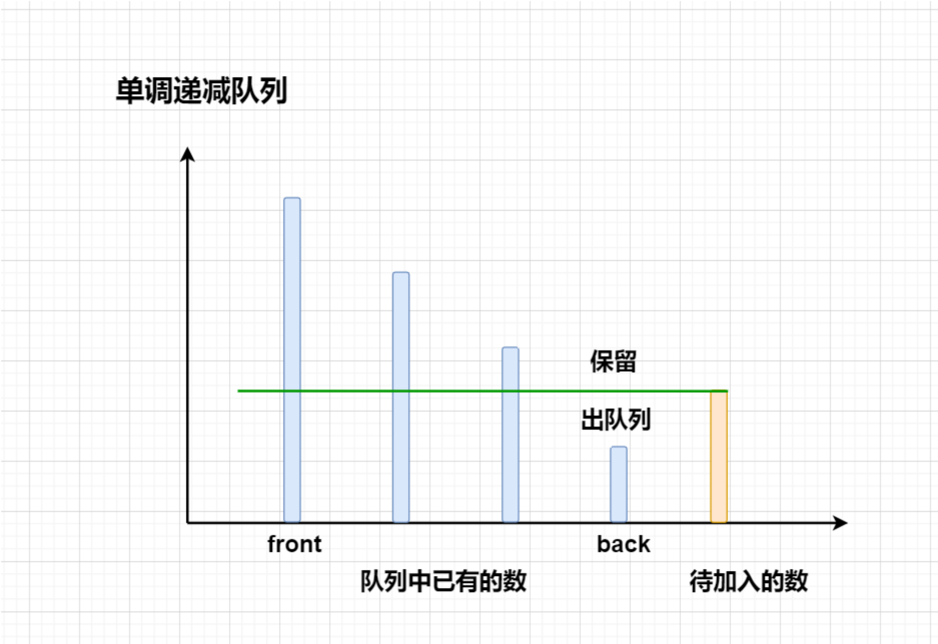
单调递增队列



对于队列内的元素来说：

1. 在队列内自己左边的数就是数组中左边第一个比自己小的元素。
2. 当被弹出时，遇到的就是数组中右边第一个比自己小的元素 。(只要元素还在队列中，就意味着暂时还没有数组中找到自己右侧比自己小的元素)
3. **队头到队尾单调递增，队首元素为队列最小值。**

单调递减队列



对于队列内的元素来说：

1. 在队列内自己左边的数就是数组中左边第一个比自己大的元素。
2. 当被弹出时，遇到的就是数组中右边第一个比自己大的元素 ，只要元素还在队列中，就意味着暂时还没有数组中找到自己右侧比自己大的元素。
3. **队头到队尾单调递减，队首元素为队列最大值。**

了解了单调队列的一些性质以后，对于这道题我们就可以维护一个单调递减队列，来保存队列中所有递减的元素 ，随着入队和出队操作实时更新队列，这样队首元素始终就是队列中的最大值。同时如果队首元素在滑动窗口中，我们就可以将其加入答案数组中。

实现细节：

为了方便判断队首元素与滑动窗口的位置关系，**队列中保存的是对应元素的下标。**

具体解题过程如下：

初始时单调队列为空，随着对数组的遍历过程中，每次插入元素前，需要考察两个事情：

- 1、合法性检查：队头下标如果距离 `i` 超过了 `k` ，则应该出队。
- 2、单调性维护：如果 `nums[i]` 大于或等于队尾元素下标所对应的值，则当前队尾再也不可能充当某个滑动窗口的最大值了，故需要队尾出队，始终保持队中元素从队头到队尾单调递减。
- 3、如次遍历一遍数组，队头就是每个滑动窗口的最大值所在下标。

时间复杂度分析： 每个元素最多入队出队一次，复杂度为$O(n)$。

c++代码

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;
        vector<int>res;
        for(int i = 0; i < nums.size(); i++){
            while(q.size() && i - k + 1 > q.front()) q.pop_front();
            while(q.size() && nums[i] >= nums[q.back()]) q.pop_back();
            q.push_back(i);
            if(i >= k - 1) res.push_back(nums[q.front()]);
        }
        return res;
    }
```

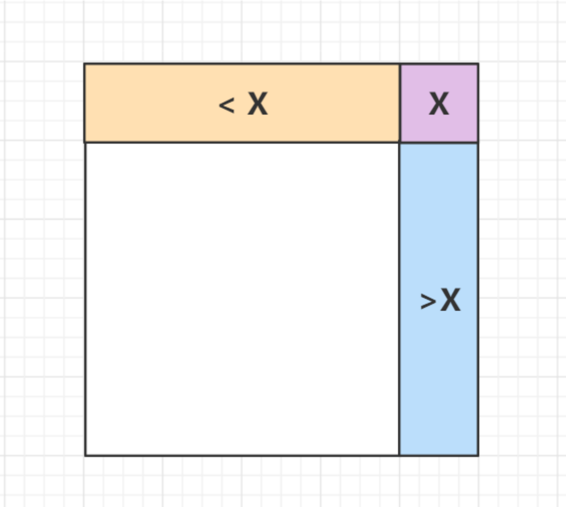
```
};
```

240. 搜索二维矩阵 II

思路

(单调性扫描) $O(n+m)$

在 `m x n` 矩阵 `matrix` 中我们可以发现一个性质：对于每个子矩阵右上角的数 `x`， `x` 左边的数都小于等于 `x`， `x` 下边的数都大于 `x`。



因此我们可以从整个矩阵的右上角开始枚举，假设当前枚举的数是 `x`：

- 如果 `x` 等于 `target`，则说明我们找到了目标值，返回 `true`；
- 如果 `x` 小于 `target`，则 `x` 左边的数一定都小于 `target`，我们可以直接排除当前一整行的数；
- 如果 `x` 大于 `target`，则 `x` 下边的数一定都大于 `target`，我们可以直接排序当前一整列的数；

排除一整行就是让枚举的点的横坐标加一，排除一整列就是让纵坐标减一。当我们排除完整个矩阵后仍没有找到目标值时，就说明目标值不存在，返回 `false`。

具体过程如下：

- 1、初始化 `i = 0`, `j = matrix[0].size() - 1`。
- 2、如果 `matrix[i][j] == target`，返回 `true`。
- 3、如果 `matrix[i][j] < target`， `i++`，排除一行。
- 4、如果 `matrix[i][j] > target`， `j--`，排除一列。
- 5、如果出界还未找到 `target`，则返回 `false`。

时间复杂度分析： 每一步会排除一行或者一列，矩阵一共有 n 行，m 列，所以最多会进行$n+m$步。所以时间复杂度是 $O(n+m)$。

c++代码

```cpp
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size(), m = matrix[0].size();
        if(!n || !m) return false;
        int i = 0, j = m - 1;
        while(i < n &&j >= 0 ){
            if(target == matrix[i][j]) return true;
            else if(target > matrix[i][j]) i++;
            else if(target < matrix[i][j]) j--;
```

```
        }
        return false;
    }
};
```

[279. 完全平方数]

思路

(动态规划 + 背包问题) $O(n\sqrt{n})$

状态表示： `f[i]` 表示通过平方数组成 `i` 所需要的最少完全平方数的个数。

状态计算：

每个物品的体积：`1`，`2`，`4`，`,,,` ，`j`，$\sqrt{i}$ （`j * j <= i`）

背包大小为`i`，我们去枚举每个物品`w`，物品的体积为`w * w`，考虑最后一个物品`j`，有两种选择：

- 1、不选物品`j`，则 `f[i] = f[i]`。
- 2、选物品`j`，则`f[i] = f[i - j * j] + 1`。

两种选择取最小值，**状态转移方程为：** `f[i] = min(f[i],f[i - j * j] + 1)`

初始化： `f[0] = 0`

c++代码

```
class Solution {
public:
    int numSquares(int n) {
        vector<int>f(n + 1, n);
        f[0] = 0; //初始化
        for(int i = 1; i <= n; i++)
            for(int j = 1; j * j <= n; j++)
                if(i >= j * j)
                    f[i] = min(f[i], f[i - j * j] + 1);
        return f[n];
    }
};
```

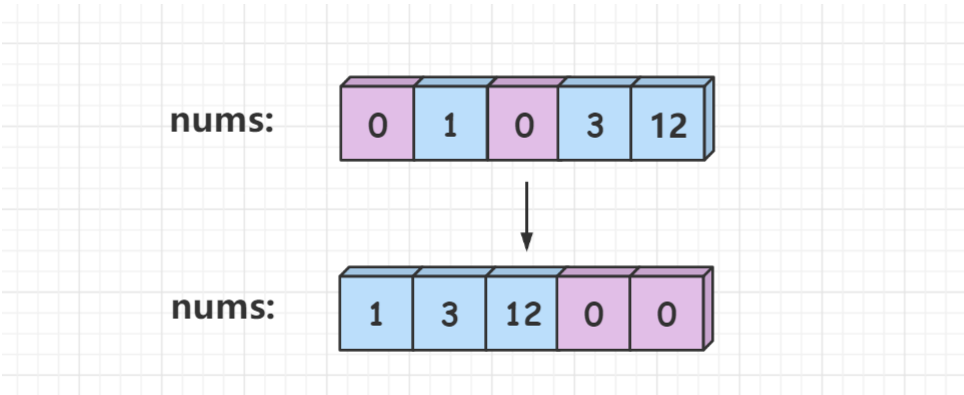
[283. 移动零]

思路

(双指针) $O(n)$

给定一个数组 `nums`，要求我们将所有的 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

样例：



如样例所示，数组`nums = [0,1,0,3,12]`，移动完成后变成`nums = [1,3,12,0,0]`，下面来讲解双指针的做法。

我们定义两个指针，`i`指针和`k`指针，`i`指针用来遍历整个`nums`数组，`k`指针用来放置`nums`数组元素。然后将非`0`元素按照原有的相对顺序都放置到`nums`数组前面，剩下的位置都置为`0`。这样我们就完成了`0`元素的移动，同时也保持了非`0`元素的相对顺序。

具体过程如下：

- 1、定义两个指针`i`和`k`，初始化`i = 0`，`k = 0`。
- 2、`i`指针向后移动，遍整个`nums`数组，如果 `nums[i] != 0`，也就是说遇到了非`0`元素，此时我们就将`nums[i]`元素放置到`nums[k]`位置，同时`k++`后一位。
- 3、最后将`k`位置之后的元素都赋值为`0`。

实现细节：

遍历数组可以使用`for(int x : nums)`，这样就少定义一个指针，代码也显得更加简洁。

时间复杂度分析： $O(n)$，n是数组的长度，每个位置只被遍历一次。

c++代码

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int k = 0;
        for(int x : nums){
            if(x != 0) nums[k++] = x;
        }
        while(k < nums.size()) nums[k++] = 0;
    }
};
```

287. 寻找重复数

思路

(二分，抽屉原理) $O(nlogn)$

抽屉原理： `n+1` 个苹果放在 `n` 个抽屉里，那么至少有一个抽屉中会放两个苹果。

在这个题目里，一共有 `n+1` 个数，每个数的取值范围是`1`到`n`，所以至少会有一个数出现两次。

然后我们采用分治的思想，将每个数的取值的区间`[1, n]`划分成`[1, n/2]`和`[n/2+1, n]`两个子区间，然后分别统计两个区间中数的个数。
注意这里的区间是指 **数的取值范围**，而不是 **数组下标**。

划分之后，左右两个区间里一定至少存在一个区间，区间中数的个数大于区间长度。
这个可以用反证法来说明：如果两个区间中数的个数都小于等于区间长度，那么整个区间中数的个数就小于等于n，和有n+1个数矛盾。

因此我们可以把问题划归到左右两个子区间中的一个，而且由于区间中数的个数大于区间长度，根据抽屉原理，在这个子区间中一定存在某个数出现了两次。

依次类推，每次我们可以把区间长度缩小一半，直到区间长度为1时，我们就找到了答案。

时间复杂度分析： 每次会将区间长度缩小一半，一共会缩小 $O(logn)$ 次。每次统计两个子区间中的数时需要遍历整个数组，时间复杂度是 $O(n)$。所以总时间复杂度是 $O(nlogn)$。

c++代码

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int l = 1, r = nums.size() - 1;  //取值范围
        while(l < r){
            int mid = (l + r) / 2;
            int s = 0;
            for(int x : nums) s += x >= l && x <= mid; //左区间数的个数
            if(s > mid - l + 1) r = mid;
            else l = mid + 1;
        }
        return r;
    }
};
```

297. 二叉树的序列化与反序列化

思路

(前序遍历序列化) $O(n)$

1. 序列化：对整个二叉树进行先序遍历的序列存起来，同时需要把每个结点的空节点使用`"#"`进行标记，例如样例的顺序是`1,2,#,#,3,4,#,#,5,#,#`
2. 反序列化：对整个字符串按照`","`进行分割，把所有的元素按序存到链表中（链表元素的顺序是先序序列），按先序遍历的方式拿链表的元素，每次拿第一个元素作为根结点，并删除链表中的第一个元素，然后递归到左儿子做同样的操作，递归到右儿子做同样的操作。注意：若第一个元素是`"#"`,表示该节点是`null`，直接返回`null

时间复杂度分析： 每个节点仅遍历两次，故时间复杂度为 $O(n)$。

c++代码

```
// 前序遍历 DFS
class Codec {
public:
    string path;

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        // if (root) dfs_s(root);  加if (root) 是错的
        dfs_s(root); // 不能加 if (root),因为空树 也要编码成"#,"
        return path;
    }

```

```
    // 序列化 dfs_s() 的返回类型 是 void
    void dfs_s(TreeNode* root) {
        if (!root) path += "#,"; // "#,"是字符串，要双引号
        else {
            path += to_string(root->val) + ','; // 单个字符，用单引号
            dfs_s(root->left);
            dfs_s(root->right);
        }
    }


    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        int idx = 0;
        return dfs_d(data, idx); // 这里不能直接 dfs_s(data, 0)，0是常量，应该是变量
    }

    // 反序列化 dfs_d() 的 返回类型是 TreeNode*, 参数 是 字符串data，和 当前 字符串下标
    TreeNode* dfs_d(string& data, int& idx){ // 第1个&防止拷贝，第2个&相当于把idx作为全局变量
        if (data[idx] == '#'){
        idx += 2;
        return NULL;
        } else {
            int k = idx;
            while (data[idx] != ',') idx ++; // 跳出while循环时，data[idx] == ','
            auto root = new TreeNode(stoi(data.substr(k, idx - k))); // stoi 和 data.substr(k, num) 用法
            idx ++ ; // 从',' 往下 跳一位
            root->left = dfs_d(data, idx);
            root->right = dfs_d(data, idx);
            return root; // return root; 不要写在下面
        }
        // return root;
    }

};
```

300. 最长递增子序列

思路

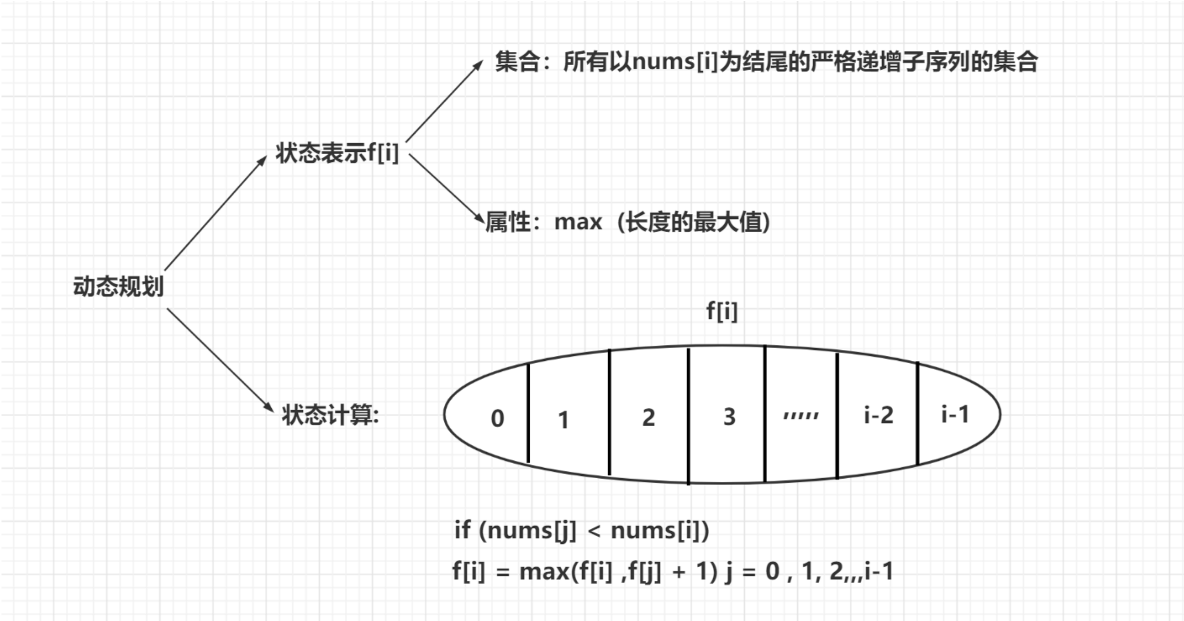
(动态规划) $O(n^2)$

状态表示： `f[i]` 表示以`nums[i]`为结尾的严格递增子序列的最大长度。

集合划分： 以`nums[i]`为结尾的严格递增子序列前一个数是`nums[0]`，`nums[1]`,,,`nums[i-1]`

状态计算： `f[i] = max(f[i],f[j] + 1)  (j<i && nums[j] < nums[i])`

图示说明：



时间复杂度分析： 状态数量为$O(n)$，状态计算为$O(n)$，故时间复杂度为$O(n^2)$。

c++代码

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        vector<int>f(n + 1);
        int res = 0;
        for(int i = 0; i < n; i++){
            f[i] = 1;
            for(int j = 0; j < i; j++)
                if(nums[i] > nums[j])
                    f[i] = max(f[i], f[j] + 1);
            res = max(res, f[i]);
        }
        return res;
    }
};
```

[309. 最佳买卖股票时机含冷冻期]

思路

(动态规划) $O(n)$

状态表示： `f[i]` 表示第 `i` 天结束后不持有股票的最大收益， `g[i]` 表示第 `i` 天结束后持有股票的最大收益。

状态计算：

- `f[i] = max(f[i - 1], g[i - 1] + prices[i])`，表示第 `i` 天什么都不做，或者卖掉持有的股票。
- `g[i] = max(g[i - 1], f[i - 2] - prices[i])`， 表示第 `i` 天什么都不做，或者买当天的股票，但需要从上两天的结果转移。

初始化： `f[0] = 0`，`g[0] = -prices[0]`。

第 `0` 天收益为 `0`，但为了持有股票，收益则为 `0 - prices[0]`。

时间复杂度分析： 状态数量为$O(n)$，状态计算为$O(1)$，故总的时间复杂度为$O(n)$。

c++代码

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<int>f(n + 1), g(n + 1);
        f[0] = 0, g[0] = -prices[0];
        for(int i = 1; i < n; i++){
            f[i] = max(f[i - 1], g[i - 1] + prices[i]);
            if(i >= 2) g[i] = max(g[i - 1], f[i - 2] - prices[i]);
            else g[i] = max(g[i - 1], -prices[i]);
        }
        return f[n - 1];
    }
};
```

312. 戳气球

思路

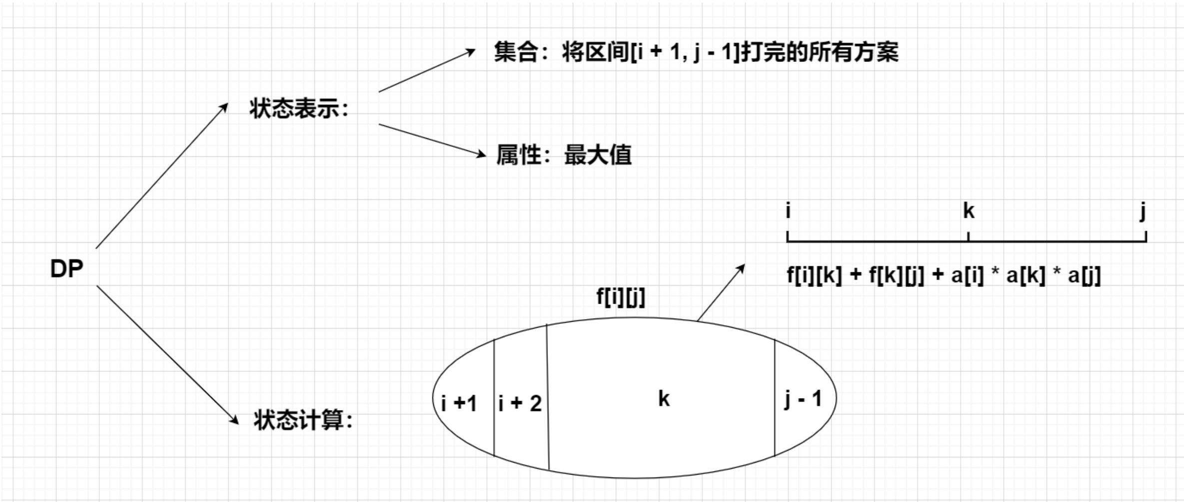
(动态规划) $O(n)$

状态表示： `f[i][j]`表示戳破区间`(i, j)`（开区间）所有气球所能获得硬币的最大数量。

状态计算：

假设最后一次戳破编号为`k`的气球：

`f[i][j] = max(f[i][j], f[i][k] + f[k][j] + a[i] * a[k] * a[j])`



时间复杂度分析： 三重循环$O(n^3)$。

c++代码

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        int n = nums.size();
        vector<int> a(n + 2, 1);  //全部初始化为1
        for(int i = 1; i <= n; i++)  a[i] =  nums[i - 1];  //下标从1开始
        vector<vector<int>> f(n + 2, vector<int>(n + 2));
```

```
        for(int len = 3; len <= n + 2; len++)  //枚举长度
            for(int i = 0; i + len - 1 <= n + 1; i++){ //[0, n + 1]  //左边界
                int j = i + len - 1;  // (i,j)  右边界
                for(int k = i + 1; k < j; k++)
                  f[i][j] = max(f[i][j], f[i][k] + f[k][j] + a[i] * a[j] * a[k]);
            }
        return f[0][n + 1];
    }
};
```

322. 零钱兑换

思路

(动态规划，完全背包问题) $O(nm)$

完全背包问题。

相当于有`n` 种物品，每种物品的体积是硬币面值，价值是`1`，每种物品可用无限次。问装满背包最少需要多少价值的物品?

先考虑二维状态

状态表示： `f[i][j]` 表示从前`i`种硬币中选，且总金额恰好为`j`的所需要的最少硬币数。

那么`f[n][amount]`就表示表示 从前`n`种硬币中选，且总金额恰好为`amount`的所需要的最少硬币数，即为答案。

集合划分：

按照第`i`种硬币可以选 `0`个,`1`个，`2`个，`3`个，，，，`k`个划分集合 `f[i][j]`。其中`k*w[i] <= j`，也就是说在背包能装下的情况下，枚举第`i`种硬币可以选择几个。

不使用第`i`种硬币，状态表示： `f[i-1][j]`

使用第`i`种硬币，假设我们使用`k`个(容量允许的情况下)，状态表示：`min(f[i-1][j - k*coin]) + k`

状态计算方程：

`f[i][j] = min(f[i-1][j],f[i-1][j-coins[i]] + 1,f[i-1][j-2*coins[i]] + 2,......,f[i-1][j-k*coins[i]] + k)`。

初始化条件:

`f[0][0]=0`，其余`f[0][j] = INF`，表示当没有任何硬币的时候，存在凑成总和为 `0` 的方案，方案所使用的硬币为 `0`；凑成其他总和的方案不存在。

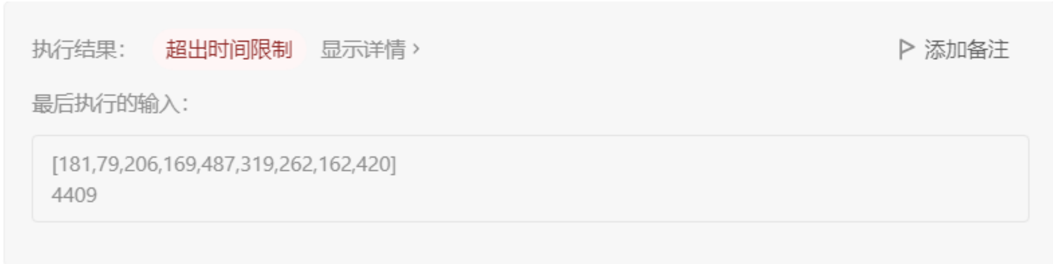
c++代码

```
class Solution {
public:
    int INF = 1000000000;
    int coinChange(vector<int>& coins, int amount) {
        int n = coins.size();
        vector<vector<int>> f (n + 1, vector<int>(amount + 1, INF));

        f[0][0] = 0;
        for(int i = 1; i <= n; i++)
        {
```

```
                int val = coins[i-1];
                for(int j = 0; j <= amount; j++)
                    for(int k = 0; k*val <= j; k++)
                    {
                        f[i][j] = min(f[i][j] , f[i-1][j-k*val] + k);
                    }
             }
            if (f[n][amount] == INF) f[n][amount] = -1;
            return f[n][amount];
        }
};
```

时间复杂度分析： 共有 $n * amount$ 个状态需要转移，每个状态转移最多遍历$amount$次。整体复杂度为$O(n * amount^2)$



超出时间限制，考虑一维优化。

一维优化

`v`代表第`i`件物品的体积(面值)

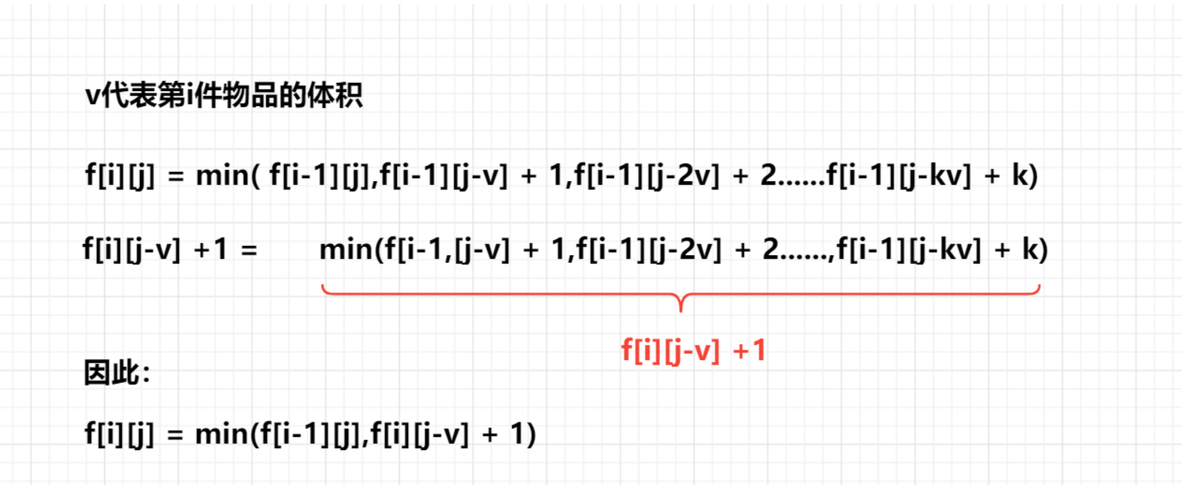
`f[i][j] = min( f[i-1][j],f[i-1][j-v] + 1,f[i-1][j-2v] + 2......f[i-1][j-kv] + k)`

`f[i][j-v] + 1 = min(f[i-1,[j-v] + 1,f[i-1][j-2v] + 2......,f[i-1][j-kv] + k)`

因此：

`f[i][j] = min(f[i-1][j],f[i][j-v] + 1)`

图示：



去掉一维：

状态计算方程为: `f[j] = min([j],[j-v] + 1)`

物品的体积即硬币面值: `f[j] = min([j],[j-coins[i]] + 1)`

时间复杂度分析：令 `n` 表示硬币种数，`m` 表示总价钱，则总共两层循环，所以时间复杂度是 $O(nm)$ 。

c++代码

```
class Solution {
public:

    int INF = 1000000000;

    int coinChange(vector<int>& coins, int amount) {
        vector<int> f(amount + 1, INF);
        f[0] = 0;
        for (int i = 0; i < coins.size(); i ++ )
            for (int j = coins[i]; j <= amount; j ++ )
                f[j] = min(f[j], f[j - coins[i]] + 1);
        if (f[amount] == INF) f[amount] = -1;
        return f[amount] ;
    }
};
```

338. 比特位计数

思路

(动态规划) $O(n)$

状态表示： `f[i]` 表示 `i` 的二进制表示中 `1` 的个数。

状态计算：

考虑 `i` 的奇偶性，有两种不同选择：

- `i` 是偶数，则 `f[i] = f[i/2]`，因为 `i/2 * 2` 本质上是 `i/2` 的二进制左移一位，低位补零，所以 `1` 的数量不变。
- `i` 是奇数，则 `f[i] = f[i - 1] + 1`，因为如果 `i` 为奇数，那么 `i - 1` 必定为偶数，而偶数的二进制最低位一定是 `0`，那么该偶数 `+1` 后最低位变为 `1` 且不会进位，所以奇数比它上一个偶数二进制表示上多一个 `1`。

初始化： `f[0] = 0`。

时间复杂度分析： $O(n)$。

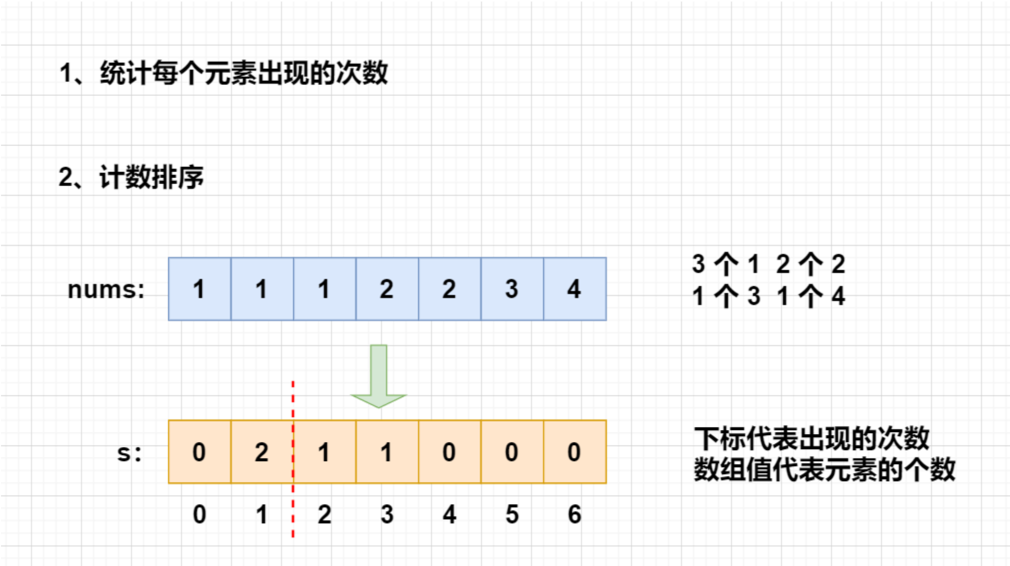
c++代码

```
class Solution {
public:
    vector<int> countBits(int n) {
        vector<int> f(n + 1);
        f[0] = 0; //初始化
        for(int i = 1; i <= n; i++){
            if(i & 1) f[i] = f[i - 1] + 1;
            else f[i] = f[i >> 1];
        }
        return f;
    }
};
```

347. 前 K 个高频元素

思路

(计数排序) $O(n)$



我们可以先统计每个数字出现了多少次，在统计一下出现次数为`t`次的元素各有多少个，然后利用计数排序的思想判断一下出现次数前`K`多的数字最少出现多少次，求出这个下界`i`，最后再遍历一次哈希表，将所有出现次数大于等于这个下界的元素加入答案。

具体过程:

- 1、先统计每个元素出现次数。
- 2、用`s`数组，`s[i]`表示出现了`i`次的元素有`s[i]`个。
- 3、根据`k`在`s`数组中找到一个分界线`i`，使得前`k`个高频元素的出现次数都`>i`次。

c++代码

```cpp
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        int n = nums.size();
        unordered_map<int, int> cnt; // 统计每个元素出现的次数
        for(int x : nums) cnt[x]++;
        vector<int> s(n + 1);  //统计每个次数出现的元素有多少个
        for(auto p : cnt) s[p.second]++;
        int i = n, t = 0;
        while(t < k) t += s[i--];
        vector<int> res;
        for(auto p : cnt){
            if(p.second > i)
              res.push_back(p.first);
        }
        return res;
    }
};


```

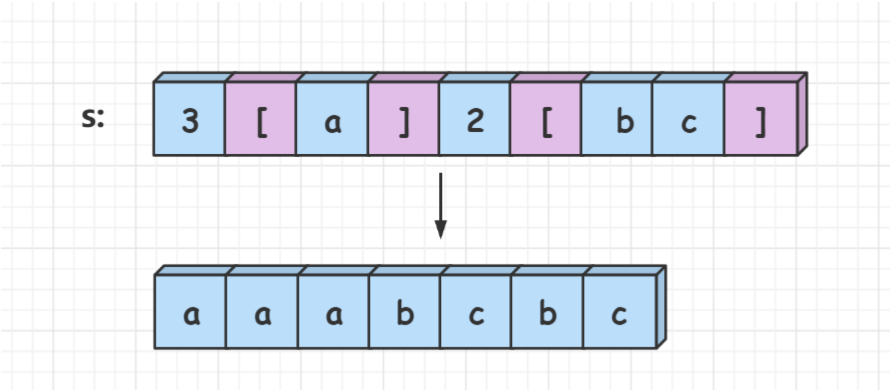
394. 字符串解码

思路

(递归) $O(n)$

给定一个经过编码的字符串，返回它解码后的字符串。

样例：



如样例所示，`s = "3[a]2[bc]"`，我们根据编码规则解码后输出`aaabcbc`，下面来讲解**递归**的做法。

我们首先来解析一下这个编码规则，方括号`[]`内包含要重复的字符串，方括号`[]`外的数字代表重复的次数，而且括号是可以嵌套的，比如样例2，`s = "3[a2[c]]"`。要想解码外层括号的字符串，就必须要先解码内层括号的字符串，这样就给了我们一种启发，我们可以先递归到内层，由内层到外层，层层解码。

递归函数设计：

```
string dfs(string &s, int &u)
```

`s`是当前要遍历的字符串，`u`是当前遍历的字符串的位置下标。

具体过程如下：

从左到右遍历整个字符串：

- 1、如果当前遇到的字符是字母，我们将其加入到答案字符串`res`中。
- 2、如果当前遇到了 `k[encoded_string]` 规则，则解析出数字`k`和字符串`encoded_string`，然后递归解码字符串`encoded_string`。
- 3、每次递归结束后，我们将解码得到的结果字符串`str`重复`k`次，然后将其添加到答案中。

我们以字符串`3[a2[c]]`为例，图示过程如下：

时间复杂度分析： 假设共有 `n` 个规则，则最坏情况下所有规则会嵌套 `n` 层：`k[k[...k[encoded_string]]]`。则最终解码后的字符串长度是 `encoded_string.length * k^n`。所以时间复杂度是 $O(k^n)$。

c++代码

```
class Solution {
public:
    string decodeString(string s) {
        int u = 0;     //当前遍历的字符串的位置下标
        return dfs(s, u);
    }
    string dfs(string &s, int &u){
```

```
        string res;
        while(u < s.size() && s[u] != ']'){
            if(s[u] >= 'a' && s[u] <= 'z' || s[u] >= 'A' && s[u] <= 'Z') res += s[u++];
            else if(s[u] >= '0' && s[u] <= '9')
            {
                int k = u, num = 0;
                while(s[k] >= '0' && s[k] <= '9') num = num * 10 + s[k++] - '0';   //将字符转换成数字
                u = k + 1;                 // 跳过左括号，递归到内层
                string str = dfs(s, u);  // 返回内层解码结果
                u++;                     // 跳过右括号
                while(num--)  res += str;
            }
        }
        return res;
    }
};
```

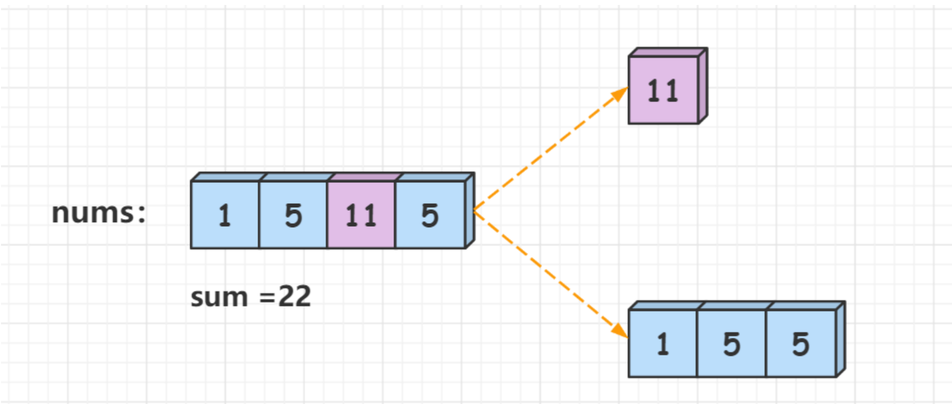
416. 分割等和子集

思路

(动态规划) $O(n*m)$

给定一个只包含正整数的非空数组 `nums[]`，判断是否可以将这个数组分割成两个子集，并且每个子集数字的和**恰好等于**整个数组的元素和的一半。

样例:



如样例所示，`nums = [1,5,11,5]`，数组可以分割成 `[1, 5, 5]` 和 `[11]`，因此返回 `ture`。从题意来看，这个问题可以转换成 `0-1` 背包问题，**如何看出来的？**我们不妨将换种表述方式:

将大小为 `n` 的数组看成 `n` 件物品，数组元素和 `sum` 的一半看成一个容量为 `sum / 2` 的背包，每件物品只能使用一次，每件物品的体积是 `nums[i]`，求解是否可以选出一些物品，使得这些物品的总体积**恰好**为背包的容量，因此可以使用动态规划求解，下面我们来讲解具体做法。

首先，如果 `sum` 是奇数，则不可能将数组分割成元素和相等的两个子集，因此直接返回 `false`。接下来我们去定义状态表示和推导状态转移方程。

状态表示: `f[i][j]` 表示从前 `i` 个数中选若干个数，是否使得这些数字的和**恰好等于** `j`。因此 `f[i][j]` 有两种状态，`true` 或者 `false`。

状态计算:

假定 `nums[]` 数组下标从 `1` 开始，如何确定 `f[i][j]` 的值？

一般去考虑最后一步，那么对于当前的数字 `nums[i]`，可以选取也可以不选取:

- 1、不选 `nums[i]`，那么我们就从前 `i - 1` 个数中选，看是否使得这些数字的和恰好等于 `j`，即 `f[i][j] = f[i - 1][j]`。
- 2、选择 `nums[i]`，在背包可以装下的情况下，那么相应的背包容量就要减去 `nums[i]`，`f[i][j]` 的状态就可以从 `f[i - 1][j - nums[i]]` 转移过来，即 `f[i][j] = f[i - 1][j - nums[i]]`。

综上，两种情况只要有一个为 `true`，那么 `f[i][j]` 就为 `true`。因此**状态转移方程为** `f[i][j] = f[i - 1][j] | f[i - 1][j - nums[i]]`。

初始化:

`f[0][0] = true`：在前 `0` 个数中，我们可以一个数都不去选，因此从前 `0` 个数中选，使得这些数字的和**恰好等于** `0` 的状态为 `true`，其余的状态都初始化为 `false`。

实现细节:

在推导状态转移方程时，我们假设的 `nums[]` 数组下标是从 `1` 开始的，而实际中的 `nums[]` 数组下标是从 `0` 开始的，因此在代码的编写过程中，我们需要将所有 `nums[i]` 的下标减去 `1`，与使用的语言保持一致。

时间复杂度分析: $O(n*m)$，`n` 是 `nums` 数组的大小，`m` 数组元素和的一半。

空间复杂度分析: $O(n*m)$

c++代码

```cpp
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int n = nums.size(), sum = 0;
        for(int x : nums) sum += x;
        if(sum % 2) return false;
        int m = sum / 2;
        vector<vector<bool>> f(n + 1, vector<bool>(m + 1, false));
        f[0][0] = true;
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                if(j >= nums[i - 1]) f[i][j] = f[i - 1][j - nums[i - 1]] | f[i - 1][j];
                else f[i][j] = f[i - 1][j];
            }
        }
        return f[n][m];
    }
};
```

java代码

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length, sum = 0;
        for(int x : nums) sum += x;
        if(sum % 2 != 0) return false;
        int m = sum / 2;
        boolean[][] f = new boolean[n + 1][m + 1];
        f[0][0] = true;
```

```
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                if(j >= nums[i - 1]) f[i][j] = f[i - 1][j - nums[i - 1]] || f[i - 1][j];
                else f[i][j] = f[i - 1][j];
            }
        }
        return f[n][m];
    }
}
```

一维优化

我们可以发现，在计算 `f[i][j]` 的过程中，每一行 `f[i][j]` 的值只与上一行的 `f[i - 1][j]` 有关，因此考虑去掉前一维，状态转移方程为：`f[j] = f[j] | f[j - nums[i]]`。

如果此时我们继续考虑第二层循环 `j` 从小往大计算，即：

```
for (int i = 1; i <= n; i++){              //为了下标对应，实际nums[i]应取nums[i - 1]
    for (int j = nums[i]; j <= m; j++){
         f[j] = f[i] | f[j - nums[i]];
    }
}
```

此时的状态便与二维的状态不等价了，因为在计算第 `i` 层的状态时，我们从小到大枚举， `j - nums[i]` 严格小于 `j`，那么 `f[j-nums[i]]` 一定会先于 `f[j]` 被计算出来，于是我们计算出来的 `f[j - nums[i]]` 仍为第 `i` 层状态，这样 `f[j - nums[i]]` 等价于 `f[i][j-nums[i]]` ，实际上 `f[j - nums[i]]` 应该等价于 `f[i - 1][j - nums[i]]`。

为了解决这个问题只需要将 `j` 从大到小枚举。

```
for (int i = 1; i <= n; i++){              //为了下标对应，实际nums[i]应取nums[i - 1]
    for (int j = m; j >= nums[i]; j -- ){
         f[j] = f[i] | f[j - nums[i]];
    }
}
```

因为我们从大到小枚举 `j`，而 `j - nums[i]` 严格小于 `j`，于是我们在计算 `f[j]` 的时候 `f[j - nums[i]]` 还未被第 `i` 层状态更新过，那么它存的就是上一层(`i - 1` 层)的状态，即 `f[i - 1][j - nums[i]]`。

空间复杂度分析：$O(n)$

c++代码

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int n = nums.size(), m = 0;
        for (int x: nums) m += x;
        if (m % 2) return false;
        m /= 2;
        vector<bool> f(m + 1);
        f[0] = true;
```

```
        for (int i = 1; i <= n; i++)
            for (int j = m; j >= nums[i - 1]; j -- )
                f[j] = f[j] | f[j - nums[i - 1]];
        return f[m];
    }
};
```

437. 路径总和 III

思路

(dfs) $O(n^2)$

我们遍历每一个节点 `node`，搜索以当前节点`node`为起始节点往下延伸的所有路径，并对路径总和为`targetSumtarget`的路径进行累加统计。

时间复杂度分析： 遍历整颗树需要$O(n)$的时间，搜索每条路径需要$O(n)$的时间，故总的时间复杂度为$O(n^2)$。

c++代码

```
/**
 * 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:
    int res = 0;
    int pathSum(TreeNode* root, int targetSum) {
        if(!root) return 0;
        dfs(root, targetSum);
        pathSum(root->left, targetSum);
        pathSum(root->right, targetSum);
        return res;
    }
    void dfs(TreeNode* root, int sum){
        if(!root) return ;
        sum -= root->val;
        if(!sum) res++;
        dfs(root->left, sum);
        dfs(root->right, sum);
    }
};
```

(前缀和 + 哈希) $O(n)$

求出二叉树的前缀和，统计以每个节点`node`节点为路径结尾的合法路径的数量，记录一个哈希表`cnt`，维护每个前缀和出现的次数。

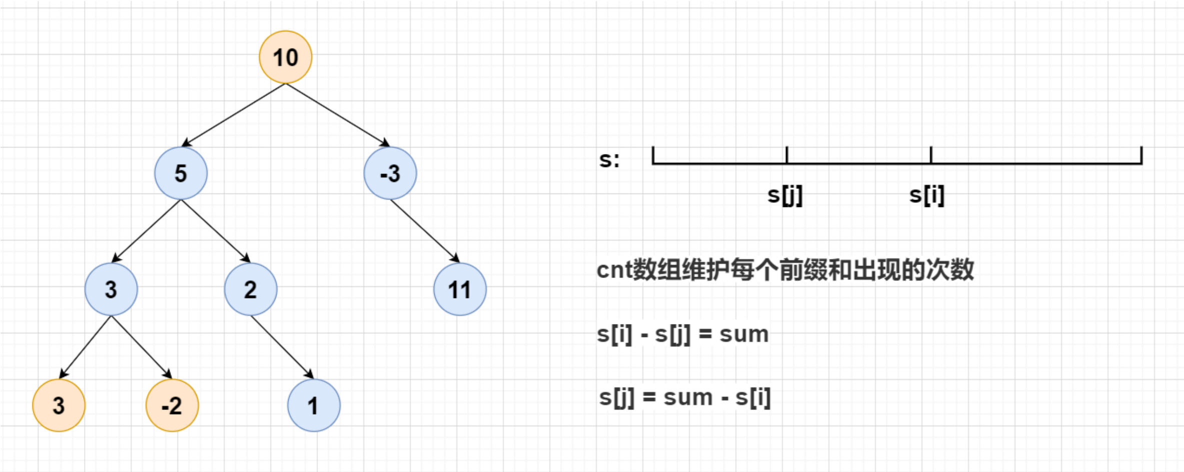
对于当前节点`root`，前缀和为`cur`，累加 `cnt [cur - sum]`的值，看看有几个路径起点满足。

递归函数设计：

```
void dfs(TreeNode* root, int sum, int cur)
```

`root`是当前遍历的节点，`sum`是目标数，`cur`是当前经过的路径之和。

如下图：



时间复杂度分析： 树中的每个节点被遍历一遍，故时间复杂度为$O(n)$。

c++代码

```
/**
 * 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:
    unordered_map<int, int>cnt;
    int res = 0;
    int pathSum(TreeNode* root, int targetSum) {
        cnt[0] = 1;  //前缀和0出现了一次
        dfs(root, targetSum, 0);
        return res;
    }

    void dfs(TreeNode* root, int sum, int cur){
        if(!root) return ;
        cur += root->val;
        res += cnt[cur - sum];
        cnt[cur]++;
        dfs(root->left, sum, cur), dfs(root->right, sum, cur);
        cnt[cur]--;
    }
};
```

438. 找到字符串中所有字母异位词

思路

(滑动窗口，哈希表) $O(n)$

1、定义两个哈希表`hs`，`hp`，`hs`哈希表维护的是`s`字符串中滑动窗口中各个字符出现多少次，`ht`哈希表维护的是`t`字符串各个字符出现多少次。

2、定义两个指针`j`和`i`，`j`指针用于收缩窗口，`i`指针用于延伸窗口，则区间`[j,i]`表示当前滑动窗口。首先让`i`和`j`指针都指向字符串`s`开头，然后枚举整个字符串`s`，枚举过程中，不断增加`i`使滑动窗口增大，相当于向右扩展滑动窗口。

3、每次向右扩展滑动窗口一步，将`s[i]`加入滑动窗口中，而新加入了`s[i]`，相当于滑动窗口维护的字符数加一，即`hs[s[i]]++`。

4、当`hs[s[i]] > hp[s[i]`时，说明`hs`哈希表中`s[i]`的数量多于`hp`哈希表中`s[i]`的数量，此时我们就需要向右收缩滑动窗口，`j++`并使`hs[s[j]]--`，即`hs[s[j ++ ]] --`。

5、当`i - j + 1 == p.size()`，我们将起始索引`j`加入答案数组中。

时间复杂度分析： $O(n)$

c++代码

```cpp
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        unordered_map<char, int> hs, hp;
        for(int c : p) hp[c]++;
        vector<int> res;
        for(int i = 0, j = 0; i < s.size(); i++){
            hs[s[i]]++;
            while(hs[s[i]] > hp[s[i]])  hs[s[j++]]--;
            if(i - j + 1 == p.size()){
                res.push_back(j);
            }
        }
        return res;
    }
};
```

448. 找到所有数组中消失的数字

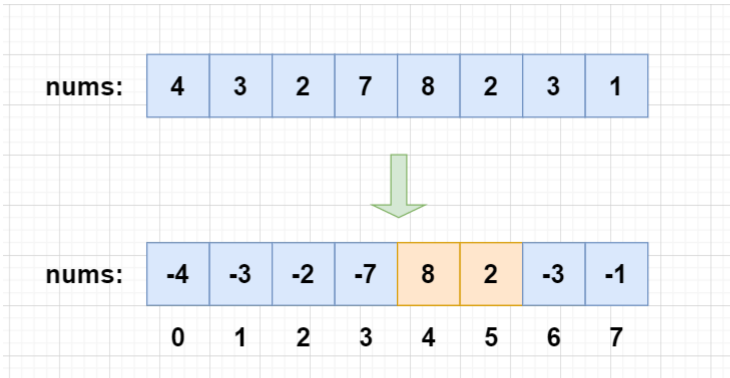
思路

(数组) $O(n)$

用负号识别当前数是否用过

1、遍历每个元素，对索引进行标记，将对应索引位置的值变为负数；

2、遍历下索引，看看哪些索引位置上的数不是负数的，位置上不是负数的索引，对应的元素就是不存在的。



时间复杂度分析： 遍历两次数组，故时间复杂度为$O(n)$。

c++代码

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

461. 汉明距离

思路

(位运算) $O(logx)$

1、先将`x`和`y`作异或运算，异或运算之后，相同位为`0`，不同位为`1`。

2、统计`x ^ y`中`1`的个数。

时间复杂度分析： 异或的时间复杂度为 $O(1)$，统计二进制位 1 的个数时间复杂度为 $O(logx)$，故总时间复杂度为 $O(logx)$。

c++代码

```
class Solution {
public:
    int hammingDistance(int x, int y) {
        int n = x ^ y;    //相同位为0,不同位
        int res = 0;
        while(n){
            n -= n & -n; //lowbit
            res++;
        }
        return res;
    }
};
```