链表

二叉树

894.所有可能的满二叉树

https://leetcode-cn.com/problems/all-possible-full-binary-trees/

3.1					
看答案					

3.1 本题最开始拿到时，考虑的方法是利用DFS来进行回溯，但是结果发现回溯时，每次撤销导致有情况丢失。看答案如下，本质还是递归，不过用了排列组合的思想：

```
class Solution {
public:
    vector<TreeNode*> allPossibleFBT(int N) {
        vector<TreeNode*> temp;
        if (N == 0) return temp;
        if (N % 2 == 0)  return temp;
        if (N == 1) {
            temp.push_back(new TreeNode(0));
            return temp;
        }
        // 分别分奇数构建左右子树
        for (int i = 1; i < N-1; i+=2) {
            vector<TreeNode*> left = allPossibleFBT(i);
            vector<TreeNode*> right = allPossibleFBT(N-1-i);
            // 排列组合
            for (int i = 0; i < left.size(); ++i)
                for (int j = 0; j < right.size(); ++j) {
                    TreeNode* node = new TreeNode(0);
                    node->left = left[i];
                    node->right = right[j];
                    temp.push_back(node);

                }
        }
        return temp;
    }
};
```

1123.最深叶结点的最近公共祖先

https://leetcode-cn.com/problems/lowest-common-ancestor-of-deepest-leaves/

3.1					
自己做出来，和答案不同，答案更为巧妙					

3.1 先分两步，首先找到最深叶结点，利用BFS；其找到这些节点的最近公共祖先，参考之前写的找到最近公共祖先的题

```
class Solution {
public:
// 先分两步，首先找到最深叶结点，利用BFS；其找到这些节点的最近公共祖先
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        if (root == nullptr)    return nullptr;
        queue<TreeNode*> q, copy;
        q.push(root);
        while (!q.empty()) {
            copy = queue(q);
            int size = q.size();
            while (size--) {
                TreeNode* node = q.front();
                q.pop();
                if (node->left) q.push(node->left);
                if (node->right)    q.push(node->right);
            }
        }
        unordered_map<TreeNode*, int> dict;
        while(!copy.empty()) {
            dict[copy.front()]++;
            copy.pop();
        }
        return lowestCommonAncestor(root, dict);
    }

    TreeNode* lowestCommonAncestor(TreeNode* root, unordered_map<TreeNode*,
int>& dict) {
        if (root == nullptr)    return nullptr;
        if (dict.count(root))   return root;

        TreeNode* left = lowestCommonAncestor(root->left, dict);
        TreeNode* right = lowestCommonAncestor(root->right, dict);
        if (left == nullptr)    return right;
        if (right == nullptr)   return left;
        return root;
    }
};
```

更为清晰的方法，最深叶结点的祖先具有的特点，左右子树高度相等，如果不等，说明结果在高的子树里面：

```
class Solution {
public:
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        if (root == nullptr)    return nullptr;
```

```
        int left = len_tree(root->left);
        int right = len_tree(root->right);

        if (left == right)
            return root;
        else if (left > right)
            return lcaDeepestLeaves(root->left);
        else
            return lcaDeepestLeaves(root->right);
    }

    int len_tree(TreeNode* root) {
        if (root == nullptr)    return 0;
        return max(len_tree(root->left), len_tree(root->right)) + 1;
    }
};
```

此时存在多次重复计算子树深度的问题，将递归求解与递归求高度融合一起.

1315.最深叶结点的最近公共祖先

https://leetcode-cn.com/problems/sum-of-nodes-with-even-valued-grandparent/

3.1					
自己做出来					

3.1 祖父节点，其实也就是看 值为偶数的节点的孙子节点 是否存在 孙子节点有四个

```
class Solution {
public:
// 祖父节点，其实也就是看 值为偶数的节点的孙子节点 是否存在 孙子节点有四个
    int sumEvenGrandparent(TreeNode* root) {
        int val_sum = 0;
        dfs(root, val_sum);
        return val_sum;
    }

    void dfs(TreeNode* root, int& val_sum) {
        if (root == NULL) return;

        if (root->val % 2 == 0) {
            if (root->left) {
                if (root->left->left)   val_sum += root->left->left->val;
                if (root->left->right)   val_sum += root->left->right->val;
            }
            if (root->right) {
                if (root->right->left)   val_sum += root->right->left->val;
                if (root->right->right)   val_sum += root->right->right->val;
            }
        }

        if (root->left)     dfs(root->left, val_sum);
```

```
        if (root->right)    dfs(root->right, val_sum);

    }
};
```

235.二叉搜索树的最近公共祖先

https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

3.1					
参看答案					

与后面一题的区别，此处借助二叉搜索的特点，就更能找到p、q的分布

后面一题则需要不断后续遍历的方式，找到对应的节点在哪，并不断返回直到出现分布符合不在同一侧的情况。

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == NULL)   return NULL;
        if (p->val > root->val && q->val > root->val)
            return lowestCommonAncestor(root->right, p, q);
        else if (p->val < root->val && q->val < root->val)
            return lowestCommonAncestor(root->left, p, q);
        else
            return root; // 分散两边 或者 出现p、q其中一个在root上，另一个在左、右子树

    }
};
```

236.二叉树的最近公共祖先

https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/

3.1					
参看答案					

见上一题对比。一般二叉树无法预知节点的分布，故不能采用二叉搜索树中不断逼近的方法，此时只能遍历来找值，一旦找到就不断返回节点的指针，根据两个指针在树中的位置来判断关系。同侧时，找到在上面的节点。不同侧时找到最近的公共节点。

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == NULL)   return NULL;
        if (root->val == p->val || root->val == q->val) return root;
```

```
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left == NULL)   return right;
        if (right == NULL)  return left;
        return root;

    }
};
```

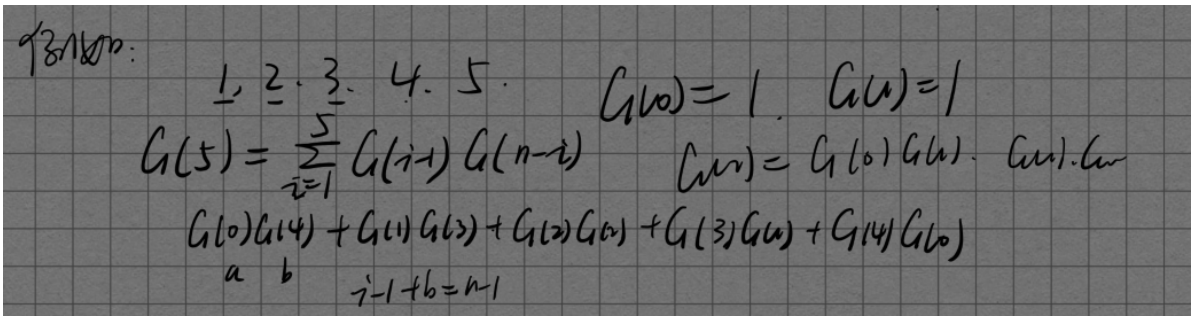
96.不同的二叉搜索树

https://leetcode-cn.com/problems/unique-binary-search-trees/

3.1					
参看答案					

这一题与下面一题类似，但是这题没有必要遍历路径，相对的，可以由下一题实际遍历的思路来直接计算结果。相反，也可以用本题来指导下一题

找到关系后，应该看出来是有重复子问题的，故使用动态规划




```
class Solution {
public:
// 利用中间节点进行分割找到关系，利用动态规划分别来求
    int numTrees(int n) {
        vector<int> dp = vector<int>(n+1, 0);
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; ++i)
            for (int j = 1; j <= i; ++j)
                dp[i] += dp[j-1]*dp[i-j];

        return dp[n];
    }
};
```

97.不同的二叉搜索树II

https://leetcode-cn.com/problems/unique-binary-search-trees-ii/

3.1					
参看答案					

```
class Solution {
public:
    vector<TreeNode*> generateTrees(int n) {
        return buildTree(1, n);
    }

    vector<TreeNode*> buildTree(int left, int right) {
        if (left > right)
            return {nullptr};
        vector<TreeNode*> temp;
        for (int i = left; i <= right; ++i) {
            vector<TreeNode*> left_list = buildTree(left, i-1);
            vector<TreeNode*> right_list = buildTree(i+1, right);

            for (auto& left: left_list)
                for (auto& right: right_list) {
                    TreeNode* node = new TreeNode(i);
                    node->left = left;
                    node->right = right;
                    temp.emplace_back(node);
                }
        }
        return temp;
    }
};
```

919.完全二叉树插入器

https://leetcode-cn.com/problems/complete-binary-tree-inserter/

3.1					
参看答案					

这种题目一般都是借用成熟的数据结构来构建，树的问题就在于需要递归比价来索引，无法直接得到，可以建立索引对应来完成。

从1开始存入节点，索引为n的节点，其左子节点索引为2n，右子节点索引为2n+1

```
class CBTInserter {
public:

```

```
// 这种题目一般都是借用成熟的数据结构来构建，树的问题就在于需要递归比价来索引，无法直接得到，可以建立索引对应来完成
    vector<TreeNode*> dict;
    CBTInserter(TreeNode* root) {
        dict.emplace_back(new TreeNode(0));
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            while(size--) {
                TreeNode* node = q.front();
                dict.emplace_back(node);
                q.pop();
                if (node->left)     q.push(node->left);
                if (node->right)    q.push(node->right);
            }
        }
    }

    int insert(int v) {
        TreeNode* node = new TreeNode(v);
        dict.emplace_back(node);
        int index = dict.size() - 1;
        int f_index = index / 2;
        if (index % 2)
            dict[f_index]->right = node;
        else
            dict[f_index]->left = node;
        return dict[f_index]->val;
    }

    TreeNode* get_root() {
        if (dict.size() == 1) // 此题可以不加，加上防止索引为空时索引溢出
            return NULL;
        return dict[1];
    }
};
```

区间问题

56.合并区间

https://leetcode-cn.com/problems/merge-intervals/

3.2					
自己做的					

先排序，可以考虑下有没有必要排序第二个元素。维护一个最长区间，依次拿区间进行对比，分析清楚包含、相交、分离的三种情况。

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> ans;
        sort(intervals.begin(), intervals.end());

        vector<int> max_inter = intervals[0];
        for (int i = 1; i < intervals.size(); ++i) {
            // if (max_inter[1] >= intervals[i][1])
            //     continue;
            if (max_inter[1] >= intervals[i][0] && max_inter[1] < intervals[i]
[1]) // 第二个字符可以不排序，包含在这里
                max_inter[1] = intervals[i][1];
            else if (max_inter[1] < intervals[i][0]) {
                ans.push_back(max_inter);
                max_inter = intervals[i];
            }
        }
        ans.push_back(max_inter);
        return ans;

    }
};
```

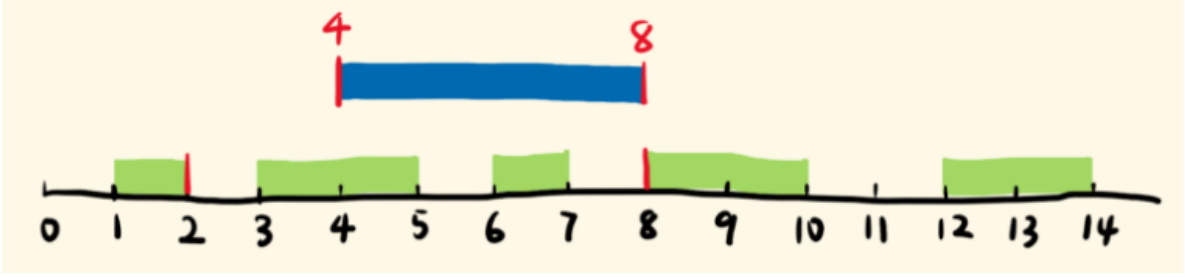
57.插入区间

[57. 插入区间 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/insert-interval/)

3.2					
参考答案					

最笨的方法：相当于按照56题的，插入以及重新排序

优化的方法：抓住原本无重叠且有序，根据插入的区间来选择合适的插入位置，减少排序消耗




```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>&
newInterval) {
        vector<vector<int>> ans;

        int i = 0;
        int size = intervals.size();
        // 在待插入区间左边的 只看待插入左边界
```

```
        while (i < size && intervals[i][1] < newInterval[0]) {
            ans.push_back(intervals[i]);
            i++;
        }
        // 与待插入区间存在交叉区域 只看待插入右边界
        while (i < size && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = min(newInterval[0], intervals[i][0]);
            newInterval[1] = max(newInterval[1], intervals[i][1]);
            i++;
        }
        ans.push_back(newInterval);
        // 待插入区间右边的
        while (i < size) {
            ans.push_back(intervals[i]);
            i++;
        }
        return ans;
    }
};
```

1288.删除被覆盖区间

[1288. 删除被覆盖区间 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/remove-covered-intervals/)

3.2					
参考答案					

和上面题目类似思路，但是直接累计结果就好了，注意排序得首位从小到大，末尾从大到小的方式，这样不会丢失被覆盖的区间，当区间完全被覆盖时count++

```
class Solution {
public:
    static bool compare(vector<int>& a, vector<int>& b) {
        if (a[0] == b[0])
            return a[1] > b[1];
        return a[0] < b[0];
    }

    int removeCoveredIntervals(vector<vector<int>>& intervals) {
        sort(intervals.begin(), intervals.end(), compare);

        int remove = 0;
        vector<int>& max_inter = intervals[0];
        for(int i = 1; i < intervals.size(); ++i) {
            if (intervals[i][1] <= max_inter[1])
                remove++;
            else if (max_inter[1] >= intervals[i][0] && max_inter[1] <
intervals[i][1])
```

```
                max_inter[1] = intervals[i][1];
            else if (max_inter[1] < intervals[i][0])
                max_inter = intervals[i];
        }
        return intervals.size()-remove;
    }
};
```

228.汇总区间

228. 汇总区间 - 力扣（LeetCode） (leetcode-cn.com)

3.2					
自己做					

遍历更新就好了

```
class Solution {
public:
    vector<string> summaryRanges(vector<int>& nums) {
        if (nums.empty())   return {};

        vector<string> ans;
        vector<int> inter(2, nums[0]);

        for(int i = 0; i < nums.size()-1; ++i) {
            if (nums[i] + 1 == nums[i+1])
                inter[1] = nums[i+1];
            else {
                if (inter[0] == inter[1])
                    ans.emplace_back(to_string(inter[0]));
                else
                    ans.emplace_back(to_string(inter[0]) + "->" +
to_string(inter[1]));
                inter[0] = nums[i+1];
                inter[1] = nums[i+1];
            }
        }
        if (inter[0] == inter[1])
            ans.emplace_back(to_string(inter[0]));
        else
            ans.emplace_back(to_string(inter[0]) + "->" + to_string(inter[1]));
        return ans;
    }
};
```

986.区间列表的交集

986. 区间列表的交集 - 力扣（LeetCode） (leetcode-cn.com)

	3.2				
	参考历史记录				

有点NMS的意味，分别各拿一个，算重合区间，存在则记录，否则看谁长，选短的那一端递增。

```
class Solution {
public:
    vector<vector<int>> intervalIntersection(vector<vector<int>>& firstList,
vector<vector<int>>& secondList) {
        vector<vector<int>> ans;
        int size1 = firstList.size(), size2 = secondList.size();
        int i = 0, j = 0;
        while (i < size1 && j < size2) {
            vector<int>& a = firstList[i];
            vector<int>& b = secondList[j];
            int left = max(a[0], b[0]);
            int right = min(a[1], b[1]);
            if (right - left >= 0)
                ans.emplace_back(vector<int>{left, right});
                // cout << left << right << endl;
            if (b[1] > a[1])
                i++;
            else
                j++;
        }
        return ans;
    }
};
```

435.无重叠区间

435. 无重叠区间 - 力扣（LeetCode） (leetcode-cn.com)

3.2					
参考历史记录					

贪心算法，有重合时应该尽量去除右边界长的（保留右边界小的）。

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty())  return 0;

        sort(intervals.begin(), intervals.end());
        int end = intervals[0][1];
        int count = 0;
```

```
        for(int i = 1; i < intervals.size(); ++i) {
            if (intervals[i][0] < end) {
                end = min(intervals[i][1], end);
                count++;
            } else
                end = intervals[i][1];
        }
        return count;
    }
};
```

795.区间子数组个数

795. 区间子数组个数 - 力扣（LeetCode） (leetcode-cn.com)

3.2					
看答案的					

首先是组合的数目采用 前N项和计算；其次是利用集合相减的方式来计算中间集合的大小。

```
class Solution {
public:
    int numSubarrayBoundedMax(vector<int>& A, int L, int R) {
        return countN(A, R) - countN(A, L-1);
    }

    int countN(vector<int>& A, int T) {
        int count = 0;
        int value = 0;
        for (int n:A) {
            value = n <= T ? value+1 : 0;
            count += value;
        }
        return count;
    }
};
```

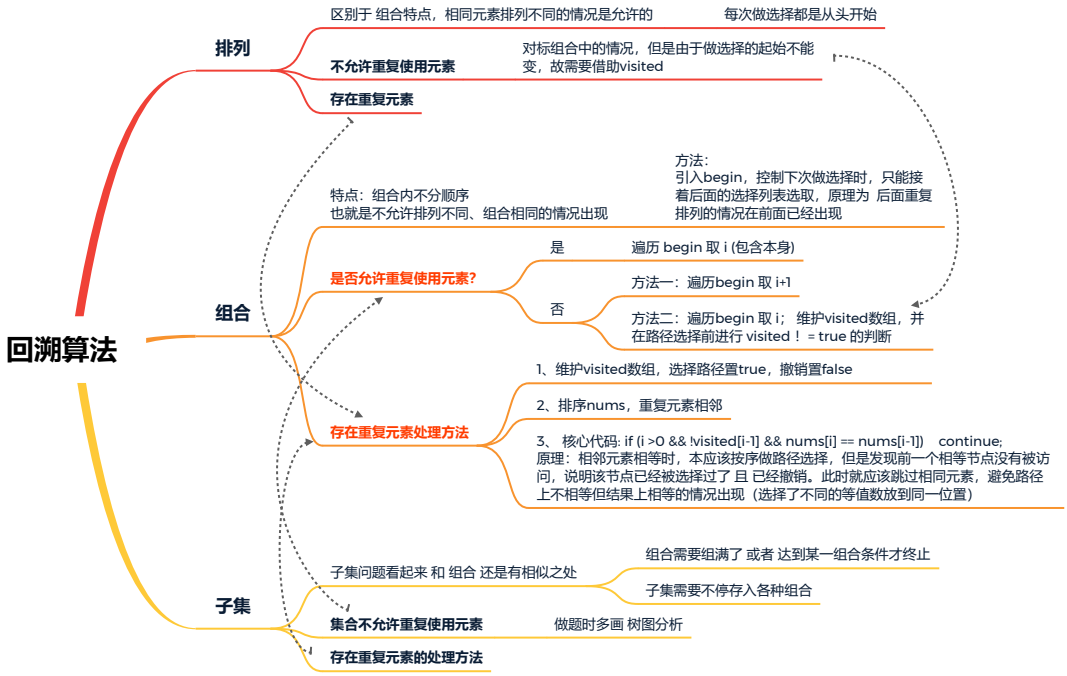
回溯算法

总结：

```
#***************************** 回溯算法框架 ***********************************

# ** 注意回溯 实际问题要适当 剪枝减低计算量；要经常结合树图分析**
    result = []
    def backtrack(路径, 路径列表):
        if 满足结束条件:
            result.add(路径)
            return
        # 做选择
        路径.add()
        # 遍历
        backtrack(路径, 路径列表)
        # 撤销选择
        路径.del()
```

具体可能遇到以下问题：



39.组合总和

[39. 组合总和 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/combination-sum/)

3.3					
看答案					

首先求解目标和 target的问题，注意用减法，避免引入 加法做和

其次得注意这是组合问题，参照组合问题特性解决。

本题可以重复使用元素，维护begin，每次从 i 开始做选择就好，这时包含了本身

本类型题目注意剪枝，明确不满足条件没必要带入递归当中，但是注意本题要想直接break，需要先排序

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<int> path;
        sort(candidates.begin(), candidates.end());
        backtrack(candidates, 0, target, path);
        return ans;

    }

    void backtrack(vector<int>& candidates, int begin, int target, vector<int>&
path) {
        if (target == 0) {
            ans.emplace_back(path);
            return;
        } else if (target < 0)
            return;
        // 做选择
        for (int i = begin; i < candidates.size(); ++i) {
            if (target-candidates[i] < 0) // 剪枝，因为已经排序了，前面为负，后面肯定更
是
                break;
            path.push_back(candidates[i]);
            backtrack(candidates, i, target-candidates[i], path);
            path.pop_back();
        }
    }
};
```

40.组合总和 II

[40. 组合总和 II - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/combination-sum-ii/)

3.3					
看答案					

39题基础上，元素只能使用一次，存在重复元素等两个问题，可以按照思维导图分别实现。

只能使用一次的问题，使用begin = i + 1实现就好；重复元素问题 排序、维护visited、if (i>0 && !visited[i] && nums[i] == nums[i-1])

```
class Solution {
public:
// 这一题相对于 39题 多了重复元素的问题，可以先排序+visited解决
    vector<vector<int>> ans;
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<int> path;
        vector<bool> visited(candidates.size(), false);
        sort(candidates.begin(), candidates.end());
```

```
        backtrack(candidates, visited, 0, target, path);
        return ans;

    }

    void backtrack(vector<int>& candidates, vector<bool>& visited, int begin,
int target, vector<int>& path) {
        if (target == 0) {
            ans.emplace_back(path);
            return;
        } else if (target < 0)
            return;

        for (int i = begin; i < candidates.size(); ++i) {
            if (target - candidates[i] < 0) // 剪枝
                break;
            // if (visited[i] == true) // 此处与下面i+1 相对，原理都为避免范围重复元素
选一个即可
            //     continue;
            if (i > 0 && !visited[i-1] && candidates[i] == candidates[i-1])
                continue;

            visited[i] = true;
            path.push_back(candidates[i]);
            backtrack(candidates, visited, i+1, target-candidates[i], path);
            visited[i] = false;
            path.pop_back();
        }
    }
};
```

216.组合总和 III

[216. 组合总和 III - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/combination-sum-iii/)

3.3					
看答案					

这一题似乎是个九叉树的问题，开始被 k 弄晕，本以为要循环3次，每次从1~9中间取值……最后发现完全可以直接遍历1~9 直到 path长度等于 k 就好了，从终止条件来约束遍历的过程。

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> combinationSum3(int k, int n) {
        vector<int> path;
        vector<bool> visited(k, false);
        backtrack(k, n, 1, visited, path);
```

```
        return ans;
    }

    void backtrack(int k, int n, int begin, vector<bool>& visited, vector<int>&
path) {
        if (n == 0 && path.size() == k) {
            ans.emplace_back(path);
            return;
        } else if (n < 0)
            return;

        for (int i = begin; i <= 9; ++i) {
            if (n - i < 0)  break;
            if (visited[i-1])
                continue;
            path.push_back(i);
            visited[i-1] = true;
            backtrack(k, n-i, i, visited, path);
            path.pop_back();
            visited[i-1] = false;
        }
    }
};
```

78.子集

78. 子集 - 力扣（LeetCode） (leetcode-cn.com)

3.3					
看答案					

子集问题，避免排列重复的情况，引入begin来限制 做选择 的起点。同时做一次选择就可以视为一个子集读入，注意空集合也是子集。

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<int> path;
        ans.push_back({});
        backtrack(nums, 0, path);
        return ans;
    }

    void backtrack(vector<int>& nums, int begin, vector<int>& path) {
        if (begin == nums.size()) {
            return;
        }
```

```
        for (int i = begin; i < nums.size(); ++i) {
            path.push_back(nums[i]);
            ans.emplace_back(path);
            backtrack(nums, i+1, path);
            path.pop_back();
        }
    }
};
```

90.子集 II

[90. 子集 II - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/subsets-ii/)

3.3					
看答案					

可能包含重复元素的子集问题，在子集问题基础上剪枝，主要剪掉重复元素的问题。维护visited，使用if (i >0 && !visited[i-1] && nums[i]==nums[i-1])

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<int> path;
        vector<bool> visited(nums.size(), false);
        ans.push_back({});
        sort(nums.begin(), nums.end());
        backtrack(nums, visited, 0, path);
        return ans;
    }

    void backtrack(vector<int>& nums, vector<bool>& visited, int begin,
vector<int>& path) {
        if (begin == nums.size())
            return;

        for (int i = begin; i < nums.size(); ++i) {
            if (i > 0 && !visited[i-1] && nums[i] == nums[i-1])
                continue;
            path.push_back(nums[i]);
            visited[i] = true;
            ans.emplace_back(path);
            backtrack(nums, visited, i+1, path);
            path.pop_back();
            visited[i] = false;
        }
    }
};
```

46.全排列

[46. 全排列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/permutations/)

3.3					
自己做					

全排列问题，每一层从头做选择，但是需要维护 visited 来防止重复元素

```
// 全排列问题用回溯算法解决
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> permute(vector<int>& nums) {
        vector<int> path;
        vector<bool> visited(nums.size(), false);
        backtrack(nums, visited, path);
        return ans;
    }
    void backtrack(vector<int>& nums, vector<bool>& visited, vector<int>& path)
{
        // 结束条件
        if (path.size() == nums.size()) {
            ans.emplace_back(path);
            return ;
        }

        for (int i = 0; i < nums.size(); ++i) {
            // nums[i] already visited, then continue
            if (visited[i])
                continue;
            path.push_back(nums[i]);
            visited[i] = true;
            backtrack(nums, visited, path);
            path.pop_back();
            visited[i] = false;
        }

    }
};
```

47.全排列 II

47. 全排列 II - 力扣（LeetCode） (leetcode-cn.com)

3.3					
自己做					

包含重复元素的全排列问题，维护 visited 和 if(i>0 && !visited[i-1] && nums[i] == nums[i-1])

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<int> path;
        vector<bool> visited(nums.size(), false);
        sort(nums.begin(), nums.end());
        backtrack(nums, visited, path);
        return ans;
    }

    void backtrack(vector<int>& nums, vector<bool>& visited, vector<int>& path)
{
        if (path.size() == nums.size()) {
            ans.emplace_back(path);
            return;
        }

        for (int i = 0; i < nums.size(); ++i) {
            if (visited[i])
                continue;

            if (i > 0 && !visited[i-1] && nums[i]==nums[i-1])
                continue;
            path.push_back(nums[i]);
            visited[i] = true;
            backtrack(nums, visited, path);
            path.pop_back();
            visited[i] = false;
        }
    }
};
```

131.分割回文串

131. 分割回文串 - 力扣（LeetCode） (leetcode-cn.com)

3.3					
参考档案					

利用 begin 和 i 类似于作为双指针的方式来不断截取字符串，重点是终止条件

```
class Solution {
public:
    vector<vector<string>> ans;
    vector<vector<string>> partition(string s) {
        vector<string> path;
        backtrack(s, 0, path);
        return ans;
    }
    void backtrack(string& s, int begin, vector<string>& path) {
        if (begin == s.size()) {
            ans.emplace_back(path);
            return;
        }

        for (int i = begin; i < s.size(); ++i) {
            if (!check(s, begin, i))
                continue;
            path.emplace_back(s.substr(begin, i-begin+1));
            backtrack(s, i+1, path);
            path.pop_back();
        }
    }

    bool check(string& s, int left, int right) {
        while (left < right) {
            if (s[left] != s[right])
                return false;
            left++;
            right--;
        }
        return true;
    }
};
```

93.复原IP地址

93. 复原 IP 地址 - 力扣（LeetCode） (leetcode-cn.com)

3.3					
自己做					

和上一题有点像的，其实也属于字符串的分段，不够得加好剪枝与合法性判断

```cpp
class Solution {
public:
    vector<string> ans;
    vector<string> restoreIpAddresses(string s) {
        string path;
        backtrack(s, 0, 0, path);
        return ans;
    }

    void backtrack(string& s, int begin, int section, string& path) {
        if (path.size() == s.size() + 4 && section == 4) { // 所得长度刚好为s+4(4个点) 且分区必须满足 4
            path.pop_back();                                // 可以不需要 == 4 之前已经被剪枝了，不可能满足长度相等情况下还分区不同。
            ans.emplace_back(path);                         // 如果没有对分区剪枝，可能会出现许多小数点填补导致长度相等
            return;
        }

        for (int i = begin; i < s.size(); ++i) {
            if (section > 3) // 分片超过四个了，剪枝
                break;
            if (i-begin >= 1 && s[begin] == '0') // 长度(i-begin+1) 大于一个，且以0开头，如果出现，后面必不可得到，剪枝
                break;
            if (stoi(s.substr(begin, i-begin+1)) > 255) // 分片数值大于255， 剪枝
                break;

            string temp = path;                         // 便于撤销选择
            path += (s.substr(begin, i-begin+1) + ".");
            backtrack(s, i+1, section+1, path);
            path = temp;
        }
    }
};
```

动态规划

总结：

1、一般形式 ： 求最值

2、核心问题 ： 穷举、空间换时间

3、三要素 ： `重叠子问题、最优子结构、状态转移方程`

4、状态转移方程思维框架： 明确**状态** -> 定义dp数组/函数的含义 -> 明确 **选择** -> 明确 base case

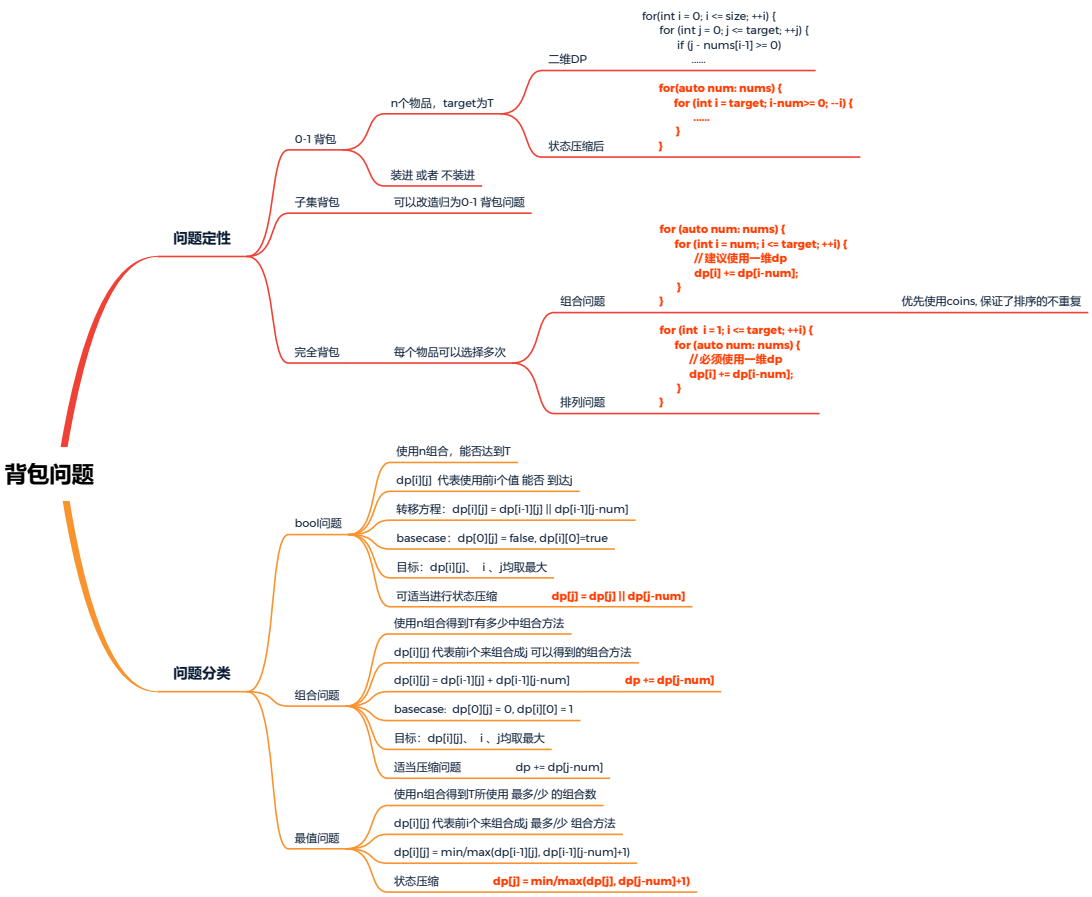
```
# 实际做题时，应该按照思考流程来，首先是 dp的定义，明确dp数组的含义非常重要，合理的定义能够帮助快速明确转移方程
# 在建立的定义上，寻找状态转移关系
# 寻找 base case 和 最终目标
# 完整构造程序
```

背包问题

总结来看：

- 找到背包问题的关键，选项与目标值（必要时需要转换问题）
- 确定是 0-1 背包 还是 完全背包
- 确定是 bool问题？ 最值问题？ 组合问题？
- 确定是组合问题还是排列问题

背包问题关键就是有多个选项，且有一个给给定目标值（或者通过转换问题得到），如果遇到物品只有一次选择机会为0-1背包问题，物体可重复多次选择的时候，为完全背包问题。针对0-1背包问题可以采用二维dp进行求解，再进行空间压缩的方式。针对完全背包，使用一维dp进行求解。嵌套循环设计时，物品在外面循环控制了物品选取的顺序性（不包含排列问题），物体是否可重复使用由转移方程中 i-1 还是 i 决定，i-1 表示已经针对最后一个做了决策，结果得依据之前的 i - 1 个物品了，但是 i 表示结果还是与 i 有关。



416.分割等和子集

https://leetcode-cn.com/problems/partition-equal-subset-sum/

3.8					
总结后自己做					

这一题为 0-1背包中的bool问题，分成两个子集转换为某一组合的target为 sum / 2；可先用思路较为清晰的二维DP，后进行空间压缩

```
// 二维 DP
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for (int num: nums)
            sum += num;
        if (sum % 2)    return false;
        int target = sum/2;
        int size = nums.size();
        vector<vector<bool>> dp(size+1, vector<bool>(target+1, false));
        dp[0][0] = true;
        for (int i = 1; i <= size; ++i) {
            for (int j = 0; j <= target; ++j) {
                if (j - nums[i-1] >= 0)
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                else
                    dp[i][j] = dp[i-1][j];
            }
        }
        return dp[size][target];
    }
};

// 一维DP 空间压缩
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for (int num: nums)
            sum += num;
        if (sum % 2)    return false;
        int target = sum/2;
        int size = nums.size();
        vector<bool> dp(target+1, false);
        dp[0] = true;
        for (int num: nums) {
            for (int j = target; j >= num; --j) {
                dp[j] = dp[j] || dp[j-num];
            }
        }
        return dp[target];
    }
};
```

494.目标和

https://leetcode-cn.com/problems/target-sum/

3.8					
总结后自己做					

这一题为 0-1背包中的组合问题，确定正数部分为 x，负数部分为 y。

$$
\begin{aligned}
x + y &= \sum num \\
x - y &= S \\
Target &= \frac{\sum num + S}{2}
\end{aligned}
$$

此时定性为 0-1背包（直接一维DP），组合问题（nums在外循环）

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int S) {
        int sum = 0;
        for (int num: nums)
            sum += num;
        if (sum < S)    return 0;
        if ((S+sum) % 2)    return 0;

        int target = (S+sum)/2;
        vector<int> dp(target+1, 0);
        dp[0] = 1;
        for (int num: nums) {
            for (int j = target; j >= num; --j) {
                dp[j] += dp[j-num];
            }
        }
        return dp[target];
    }
};
```

377.组合总和 IV

https://leetcode-cn.com/problems/combination-sum-iv/

3.8					
总结后自己做					

比较明显的完全背包， 求解组合的数量的问题，但是涉及到了排列过程故将nums循环置内

```
class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        vector<int> dp(target+1, 0);
        dp[0] = 1;
        for (int i = 0; i <= target; ++i) {
            for (int num: nums) {
                if (i - num >= 0 && dp[i] < INT_MAX-dp[i-num])
                    dp[i] += dp[i-num];
            }
        }
        return dp[target];
    }
};
```

474.一和零

https://leetcode-cn.com/problems/ones-and-zeroes/

3.8					
总结后自己做					

首先是 0-1背包问题，target为一个二维的数，求解最值问题（注意本题不是求解组合的个数问题，组合的个数一定是确定值，不存在什么最值问题，本题要求求解在 组合中元素个数最大值）

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

        for (string str: strs) {
            vector<int> count = countOnesZeros(str);
            for (int i = m; i >= count[0]; --i) {
                for (int j = n; j >= count[1]; --j) {
                    dp[i][j] = max(dp[i][j], dp[i-count[0]][j-count[1]] + 1);
                }
            }
        }
        return dp[m][n];
    }

    vector<int> countOnesZeros(string& s) {
        vector<int> count(2, 0);
```

```
        for (char c: s)
            count[c-'0']++;
        return count;
    }
};
```

518.零钱兑换 II

https://leetcode-cn.com/problems/coin-change-2/

3.8					
总结后自己做					

完全背包 ------> 内循环无需逆序

求解组合的个数（不允许排列） -----> nums在外

组合问题 -----> 存储组合的种数

```
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        vector<int> dp(amount+1, 0);
        dp[0] = 1;
        for (int coin: coins) {
            for (int j = coin; j <= amount; ++j)
                dp[j] += dp[j-coin];
        }
        return dp[amount];
    }
};
```

139.单词拆分

https://leetcode-cn.com/problems/word-break/

3.8					
参考答案					

这一题很奇怪，但是得转换思路

将字符串长度看成背包容量，子字符串看成商品，由于允许重复出现，为完全背包问题。 首先确定dp[i] 含义为 s[0~i-1]字符串 满足条件与否。在s[0~i-1] 中 找到 j 拆分该串为s[0~j-1] s[j~i-1] 两段，由于s[0~j-1] 的合法性早已经存储，需要计算 s[j~i-1] 的合法性（是否在dict中）如果两个条件满足，则s[0~i-1] 满足条件，对应dp[i]=true，等加入下一个字符的时候，又是不断 分段来确定合法性，好在左边

部分的合法性一直都在，不需要处理。

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> dict(wordDict.begin(), wordDict.end());
        int size = s.size();
        vector<bool> dp(size+1, false);
        dp[0] = true;
        for (int i = 1; i <= size; ++i) {
            for (int j = 0; j < i; ++j) {
                string str = s.substr(j, i-j);
                if (dp[j] && dict.count(str)) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[size];
    }
};
```

其它问题

860.柠檬水找零

https://leetcode-cn.com/problems/lemonade-change/

3.5					
自己做					

这一题实际是使用贪心算法实现，但是可以类别后面的零钱找零问题

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0, ten = 0; // 20的直接忽略，找不开
        int nums = 0;
        for (int bill: bills) {
            if (bill == 5)  five++;
            else if (bill == 10)    ten++;
            nums += bill;

            if (bill == 20) {
                if (ten >= 1 && five >= 1) {
                    ten--;
                    five--;
                } else if (five >= 3)
                    five -= 3;
                else
                    return false;
            }
            if (bill == 10) {
                if (five >= 1)
                    five--;
                else
                    return false;
            }
        }
        return true;
    }
};
```

322.零钱兑换

https://leetcode-cn.com/problems/coin-change/

3.5					
参考答案					

这一题实际是使用贪心算法实现，但是可以类别后面的零钱找零问题

```
// 动态规划
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int size = coins.size();
        vector<int> dp(amount+1, amount+1);
        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for(int coin: coins) {
                if (i - coin < 0)   continue;
                dp[i] = min(dp[i], dp[i-coin]+1);
```

```
            }
        }
        return (dp[amount] == amount+1 ? -1: dp[amount]);
    }
};

// 回溯方法 + 记忆化搜索
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> memo(amount+1, 0);

        return backtrack(coins, amount, memo);
    }

    int backtrack(vector<int>& coins, int amount, vector<int>& memo) {
        if (amount == 0) {
            return 0;
        } else if (amount < 0)
            return -1;

        if (memo[amount] != 0) return memo[amount];
        int min = INT_MAX;
        for (int coin: coins) {
            int res = backtrack(coins, amount-coin, memo);
            if (res >= 0 && res < min)
                min = res + 1;
        }

        memo[amount] = (min == INT_MAX)? -1: min;
        return memo[amount];
    }
};
```

10.正则表达式匹配

https://leetcode-cn.com/problems/regular-expression-matching/

3.6					
参考答案					

可以用递归的方式来实现，或者使用动态规划，具体见代码

```
// 动态规划
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size()+1, n = p.size()+1;
        // dp[i][j] 表示 s[0~i-1] 与 p[0~j-1]的匹配结果
```

```
        vector<vector<bool>> dp(m, vector<bool>(n, false));
        // 分析状态转移方程之后，需要确定边界条件，心里需要有这个想法
        // 第一列 除 第一个全是false
        // 第一行初始化，* 出现之前必须得有个字符，且利用 *消字符时，前面必须为True 防止ab*
的情况
        dp[0][0] = true;
        for (int j = 1; j < n; ++j) {
            if (p[j-1] == '*' && dp[0][j-2])
                dp[0][j] = true;
        }
        // 按照转移方程来确定，但是需要注意一个地方就是索引问题，抓住 dp[i][j] 表示 s[0~i-
1] 与 p[0~j-1]的匹配结果
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (p[j-1] == s[i-1] || p[j-1] == '.') {
                    dp[i][j] = dp[i-1][j-1];
                } else if (p[j-1] == '*') {
                    if (p[j-2] == s[i-1] || p[j-2] == '.')
                        dp[i][j] = dp[i][j-2] || dp[i-1][j];
                    else
                        dp[i][j] = dp[i][j-2];
                } else
                    dp[i][j] = false;
            }
        }
        // 终止边界
        return dp[m-1][n-1];
    }
};

// 回溯方法
class Solution {
public:
    bool isMatch(string s, string p) {
        vector<vector<bool>> memo(s.size()+1, vector<bool>(p.size()+1, false);
        return dp(s, p, 0, 0);

    }

    bool dp(string& s, string& p, int i, int j) {
        int m = s.size(), n = p.size();
        // 终止条件
        if (j == n)
            return i == m;
        if (i == m) {
            // 排除奇数的情况
            if ((n - j) % 2 == 1)
                return false;
            // 其实类似于动态规划中，dp的第一行，后续必须出现 字符* 成对出现才能消除
            for (; j+1 < p.size(); j += 2) {
                if (p[j+1] != '*')
                    return false;
            }
            return true;
        }

        if (s[i] == p[j] || p[j] == '.') {
            if (j < n-1 && p[j+1] == '*')
```

```
                return dp(s, p, i+1, j) || dp(s, p, i, j+2);
            else
                return dp(s, p, i+1, j+1);
        } else {
            if (j < n-1 && p[j+1] == '*')
                return dp(s, p, i, j+2);
            else
                return false;
        }

    }
};
```

前缀和问题

```
/*   模板  */

int function(vector<int>& nums, 一些约束) {
    int size = A.size();
        // 第一个int对应存入的值， 根据题意选定第二个int
    unordered_map<int, int> dict;
    dict[0] = 1; //
    int nsum = 0;
    int ans = 0;
    for (int i = 1; i <= size; ++i) {
        nsum += A[i-1];
        int target = nsum - S;
        // 满足条件了，依据原问题，确定是 return 还是 增加数
        if (dict.count(target))
            ans += dict[target];
        // 依据问题选择合适的更新
        dict[nsum]++;
    }
    return ans;
}
```

在求解子数组 和 子串问题 上，由于问题的特殊性（为连续的），有些问题可以采用 滑动窗口 和 前缀和的思想，能够大幅度减低暴力枚举产生的消耗。本部分着重讲述前缀和的思想，关于滑动窗口的部分，可以参考XXXXXXXX。

`前缀和` 核心思想 为解决 `快速得到某个子数组的和` 问题，将原本需要不断遍历求和的过程，直接采用两段区间相减的方式快速得到。具体计算如下：

$$
\begin{gathered}
nums[0,1,\ldots,n-1] \\
\text{定义前}n\text{项和为：}nSum[i]=nums[0]+nums[1]+\ldots+nums[i-1] \\
\text{针对}0,1,\ldots,i-1,i,i+1,\ldots,n-1 \\
\text{有}sum(i\ldots n-1)=nSum[n]-nSum[i]
\end{gathered}
$$

最基本的区间计算如以上过程，实际题目中，可能出现：

寻找满足某一条件的区间个数

解决方法：

1. 创建 unordered_map<int, int> dict 用于存储 前n项和值 与 数目
2. 更新 dict[0] = 1；很明显需要包含起点从头开始的子区间，否则会丢情况
3. 转换问题，寻找target，例如类似找到某一区间元素和为K的问题，问题定义为：
4. 找到 `nums[i, ... , j]` ,满足 `sum(nums[i, ... j]) = K`，结合前n项计算得到`nSum[j] - nSum[i] = k`，转换一下为，在当前右边界的基础上`nSum[j]`，找到`nSum[i] = nSum[j] - k`，也就是说能在 `0~j-1` 范围内找到几个`i`能够满足上述条件，即以`nSum[j]`为结尾的子数组就有几个满足原问题条件的结果。又进一步分析知道，由于`nSum[i]` 这一前n项和问题，在求解`nSum[j]`之后都已经计算过，直接借用哈希表能够快速实现查找与更新，因为为了计算满足条件的数目，很自然的hashmap 的 key 为nSum，val 为该nSum存在的次数。

寻找满足某条件的子区间： 唯一 / 其中一个/ 存在性问题

解决方法：

本质问题和上述求个数问题差不多，但是部分细节需要注意：

1. 创建 unordered_map<int, int> 用于存储 前n项和值 与 索引
2. 按照严格nSum定义，合理的更新和添加dict[0]
3. 关于 for 循环是从 0 还是 1 开始，本质问题都不大，只要能够返回正确的索引即可，但是不同问题处理上可能稍微有差异。例如 只是找到某一区间和为k 从 0 开始时，没有严格按照前n项和定义来，简化了计算量，但是相差不大，但是建议还是建议严格按照nSum定义，循环从1开始取，因为某一问题的dict[0] 还是比较重要的，特别是**涉及到区间长度限制**的时候，选用严格nSUm定义能够清晰边界条件。

724.寻找数组的中心下标

https://leetcode-cn.com/problems/find-pivot-index/

3.12					
参考做出来					

0, ... , i-1, i, i+1, ... ,n-1

题目也就是找i使得：nSum[i] = nSum[n] - nSum[i+1]

转换下：nSum[i] + nSum[i+1] = nSum[n]

找到两个连续的前n项和 的和 等于数组之和，由于至于两个相邻量有关，没必要维护数组，用 cur 和 pre 两个变量即可实现

```
class Solution {
public:
    int pivotIndex(vector<int>& nums) {
        int size = nums.size();
        int sumN = 0, sum_i = 0, sum_i1 = 0;
        for (int i = 0; i < size; ++i)
            sumN += nums[i];
```

```
        // 因为可以看到实际与两个值有关，可以进一步优化下空间
        for (int i = 0; i < size; ++i) {
            sum_i1 += nums[i];
            if (sum_i + sum_i1 == sumN)
                return i;
            sum_i = sum_i1;
        }
        return -1;
    }
};
```

1.两数之和

https://leetcode-cn.com/problems/two-sum/

3.12					
参考做出来					

本质不是前缀和问题，但是解题思路一样

unordered_map<int, int> 第一个为 nums[i]， 第二个存储 index

题目也就是找i使得：nums[i] + nums[j] = k

转换下：nums[i] = k - nums[j]

先求 nums[j] 的时候用hashmap存储下来，便于nums[i] 的 O(1) 复杂度下查找

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int size = nums.size();
        unordered_map<int, int> dict;
        for (int i = 1; i <= size; ++i) {
            if (dict.count(target-nums[i-1]))
                return {dict[target-nums[i-1]], i-1};
            dict[nums[i-1]] = i-1;
        }
        return {};
    }
};
```

560.和为K的子数组

https://leetcode-cn.com/problems/subarray-sum-equals-k/

3.12					
参考做出来					

target = nSum[i] - k

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        int size = nums.size();
        unordered_map<int, int> dict;
        dict[0] = 1;

        int cur_sum = 0;
        int ans = 0;
        for (int i = 0; i < size; ++i) {
            cur_sum += nums[i];
            int target = cur_sum - k;
            if (dict.count(target))
                ans += dict[target];
            dict[cur_sum]++;
        }
        return ans;
    }
};
```

1248.统计 优美子数组

https://leetcode-cn.com/problems/count-number-of-nice-subarrays/

3.12					
参考做出来					

target = odd[i] - k

将 求和问题 变成 奇数 计数问题

```
class Solution {
public:
    int numberOfSubarrays(vector<int>& nums, int k) {
        int size = nums.size();
        // val - > nums 存出值val 对应出现的次数（出现在前n项和中，目的就是找到前n项和中出现这个值，然后直接就可以进行区间左边的划分）
        unordered_map<int, int> dict;
        dict[0] = 1;

        int cur_num = 0;
        int ans = 0;
        for (int i = 0; i < size; ++i) {
```

```
            cur_num += nums[i] & 1;
            int target = cur_num - k;
            if (dict.count(target))
                ans += dict[target];
            dict[cur_num]++;
        }
        return ans;
    }
};
```

974.和可被K整除的子数组

https://leetcode-cn.com/problems/subarray-sums-divisible-by-k/

3.12					
参考做出来					

中间区间能够被K整除 ----> 右边界n项和 - 左边界n项和 可以被整除 ----->

左边界n项和 的余数 与 右边界n项和 余数相同 ---->

余数求解，由于c++在被除数为负的时候与平时的思路不同

余数：`(nsum % K + K) % K`

```
class Solution {
public:
    int subarraysDivByK(vector<int>& A, int K) {
        int size = A.size();
        unordered_map<int, int> dict;
        dict[0] = 1;

        int ans = 0;
        int nsum = 0;
        for (int i = 0; i < size; ++i) {
            nsum += A[i];
            int target = (nsum % K + K) % K;
            if (dict.count(target))
                ans += dict[target];
            dict[target]++;
        }
        return ans;
    }
};
```

523.连续的子数组和

https://leetcode-cn.com/problems/continuous-subarray-sum/

3.12					
参考做出来					

倍数问题与上一题一样：

中间区间能够被K整除 ----> 右边界n项和 - 左边界n项和 可以被整除 ----->

左边界n项和 的余数 与 右边界n项和 余数相同 ---->

余数求解，由于c++在被除数为负的时候与平时的思路不同

余数：`(nsum % K + K) % K`

长度问题：

使用严格的n项和定义，满足了dict[0] = 0 ，方便了左边界问题

这题还有自己的一些特别情况，看代码

```
class Solution {
public:
    bool checkSubarraySum(vector<int>& nums, int k) {
        int size = nums.size();

        if (k == 0) {
            for (int i = 0; i < size-1; ++i)
                if (nums[i] == 0 && nums[i+1] == 0)
                    return true;
            return false;
        }

        unordered_map<int, int> dict;
        dict[0] = 0;
        long long nsum = 0;
        for (int i = 1; i <= size; ++i) {
            nsum += nums[i-1];
            int target = (nsum % k + k) % k;
            if (dict.count(target) && i - dict[target] > 1)
                return true;
            // 保持最前一次的索引，即多个相同val时，保持index为最小的，保证长度
            if (!dict.count(target))
                dict[target] = i;
        }
        return false;

    }
};
```

930.和相同的二元子数组

https://leetcode-cn.com/problems/binary-subarrays-with-sum/

3.12					
参考做出来					

标准的问题，可以来对1计数，但是由于全是1 也可以直接求和来解决。

```
class Solution {
public:
    int numSubarraysWithSum(vector<int>& A, int S) {
        int size = A.size();
        // 统计次数
        unordered_map<int, int> dict;
        dict[0] = 1;
        int nsum = 0;
        int ans = 0;
        for (int i = 0; i < size; ++i) {
            nsum += A[i];
            int target = nsum - S;
            if (dict.count(target))
                ans += dict[target];
            dict[nsum]++;
        }
        return ans;
    }
};
```

437.路径综合III

https://leetcode-cn.com/problems/path-sum-iii/

3.18					
参考做出来					

算法思想
回溯 + 前缀和

前缀和的概念：
一个节点的前缀和就是该节点到根之间的路径和。
前缀和的意义：
因为对于同一路径上的两个节点，上面的节点是下面节点的祖先节点，所以其前缀和之差即是这两个节点间的路径和（不包含祖先节点的值）。
哈希map的使用：
key是前缀和， value是该前缀和的节点数量，记录数量是因为有出现复数路径的可能。
回溯的意义：
因为只有同一条路径上的节点间（节点和其某一祖先节点间）的前缀和做差才有意义。所以当前节点处理完之后，需要从map中移除这一个前缀和，然后再进入下一个分支路径。

```
/**
```

```
 * 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> dict;
    int count = 0;
    int pathSum(TreeNode* root, int sum) {
        dict[0] = 1;
        backtrack(root, 0, sum);
        return count;

    }

    void backtrack(TreeNode* node, int cur_sum, int sum) {
        if (node == nullptr)
            return;
        cur_sum += node->val;
        if (dict.count(cur_sum - sum))
            count += dict[cur_sum-sum];
        dict[cur_sum] ++;
        backtrack(node->left, cur_sum, sum);
        backtrack(node->right, cur_sum, sum);
        dict[cur_sum] --;
    }
};
```

单调栈

单调栈利用巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持单调。

在一些问题中，例如 下一个更大的元素 中，装入栈的元素 可以选择正向装和倒着装。区别在于正向装的的时候，每次出现比栈顶大的元素时，会一直弹出，这些弹出的元素下一个最大值就是此时参与比较的值。但是这种方式会导致输出存在空隙，不是连续输出，对于有些问题可能不够友好，而相对的，由后向前是持续输出的。

有些问题中，存入的可以不是值，而是索引，方便一些问题的需求。

关于**栈的形式也要灵活使用**，只要可以后端插入和删除的容器均可利用栈的思想，例如 vector、deque、string等，相比stack，某些条件下这些容器所具有的特殊性更加适合需求

```
// 单调栈模板（后入）
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> ans(nums.size()); // 存放答案的数组
    stack<int> s;
    for (int i = nums.size() - 1; i >= 0; i--) {   // 倒着往栈里放
        while (!s.empty() && s.top() <= nums[i]) { // 只有当前值小于栈顶，才能将栈顶视为该元素的后一个最大值，否则需要弹出
            s.pop();   // 弹出
        }
        ans[i] = s.empty() ? -1 : s.top(); // 如果栈为空 说明没有比当前大的值
        s.push(nums[i]);
    }
    return ans;
}
```

496.下一个更大元素 I

https://leetcode-cn.com/problems/next-greater-element-i/

3.16					
参考做出来					

```
// 前面弹出来
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        vector<int> ans;
        stack<int> stk;
        unordered_map<int, int> dict;
        int size1 = nums1.size();
        int size2 = nums2.size();
        for (int i = 0; i < size2; ++i) {
            while (!stk.empty() && nums2[i] > stk.top()) {
                dict[stk.top()] = nums2[i];
                stk.pop();
            }
            stk.push(nums2[i]);
        }
        while (!stk.empty()) {
            dict[stk.top()] = -1;
            stk.pop();
        }
        for (int i = 0; i < size1; ++i) {
            ans.push_back(dict[nums1[i]]);
        }
        return ans;
```

```
    }
};
// 后面弹出来 更加简洁对应
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        stack<int> stk;
        unordered_map<int, int> dict;

        for (int i = nums2.size()-1; i >= 0; --i) {
            while (!stk.empty() && stk.top() <= nums2[i])
                stk.pop();
            dict[nums2[i]] = stk.empty() ? -1 : stk.top();
            stk.push(nums2[i]);
        }

        vector<int> ans;
        for (int num : nums1)
            ans.push_back(dict[num]);
        return ans;
    }
};
```

739.每日温度

https://leetcode-cn.com/problems/daily-temperatures/

3.16					
复习做出来					

```
class Solution {
public:
    // 需要等待的天数 记录索引就好
    vector<int> dailyTemperatures(vector<int>& T) {
        stack<int> stk;
        vector<int> ans(T.size(), 0);

        for (int i = T.size()-1; i >= 0; --i) {
            while (!stk.empty() && T[stk.top()] <= T[i])
                stk.pop();
            ans[i] = stk.empty() ? 0 : stk.top()-i;
            stk.push(i);
        }
        return ans;
    }
};
```

496.下一个更大元素 II

https://leetcode-cn.com/problems/next-greater-element-ii/

3.16					
复习做出来					

```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n = nums.size();
        stack<int> stk;
        vector<int> ans(n, -1);

        for (int i = 2*n-1; i >= 0; --i) {
            while (!stk.empty() && stk.top() <= nums[i%n])
                stk.pop();
            ans[i%n] = stk.empty() ? -1 : stk.top();
            stk.push(nums[i%n]);
        }
        return ans;
    }
};
```

42.接雨水

https://leetcode-cn.com/problems/trapping-rain-water/

3.17					
复习做出来					

```
class Solution {
public:
    int trap(vector<int>& height) {
        int size = height.size();
        stack<int> stk;
        int water = 0;
```

```
        for (int i = 0; i < size; ++i) {
            while (!stk.empty() && height[i] > height[stk.top()]) {
                int bottom = stk.top();
                stk.pop();
                if (!stk.empty())
                    // 底部不断被填平
                    water += (i - stk.top()-1) * (min(height[stk.top()],
height[i]) - height[bottom]);
            }
            stk.push(i);
        }
        return water;
    }
};
```

84.柱形图的最大矩形

https://leetcode-cn.com/problems/largest-rectangle-in-histogram/

3.17					
复习做出来					

核心为针对每个条状图，找到左边和右边小于它的区域，以该区间作为宽，以该区间的高度作为高，可以获得面积的，并维护一个变量存储面积最大值来不断更新

暴力的方法很明确了，就是针对每个条形图，往左右阔边来找边界。

巧妙的是利用单调栈，这样保证 当输入数值大于栈顶元素，表示该栈顶所表示的框右边界到了，至于左边界，由于单调栈的性质，只要弹出栈顶元素，下一个就是它的左边界。注意之前的栈顶元素对应的高度就是 矩形框的高 ，两个边界的距离就是矩形框的宽，以此可以计算面积。

设置哨兵可以避免 栈 空与非空的判断

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        stack<int> stk;
        vector<int> temp(heights.size() + 2, 0);
        for (int i = 0; i < heights.size(); ++i)
            temp[i+1] = heights[i];
        stk.push(0);

        int max_area = 0;

        for (int i = 1; i < temp.size(); ++i) {
            while (temp[stk.top()] > temp[i]) {
                int height = temp[stk.top()];
                stk.pop();
                max_area = max(max_area, height * (i - stk.top() - 1));
```

```
            }
            stk.push(i);
        }
        return max_area;
    }
};
```

85.最大矩形

https://leetcode-cn.com/problems/maximal-rectangle/

3.17					
复习做出来					

在84题的基础上，按行来进行矩形面积的测量

```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.size() == 0)
            return 0;
        int row = matrix.size();
        int col = matrix[0].size();
        vector<int> heights(col+2, 0);

        int max_area = 0;
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                if (matrix[i][j] == '1')
                    heights[j+1] ++;
                else
                    heights[j+1] = 0;
            }
            max_area = max(max_area, maxArea(heights));
        }
        return max_area;

    }

    int maxArea(vector<int>& heights) {
        stack<int> stk;
        stk.push(0);
        int max_area = 0;

        for (int i = 1; i < heights.size(); ++i) {
            while (heights[stk.top()] > heights[i]) {
                int height = heights[stk.top()];
                stk.pop();
                max_area = max(max_area, height * (i - stk.top() - 1));
            }
            stk.push(i);
        }
```

```
        return max_area;
    }
};
```

221.最大正方形

https://leetcode-cn.com/problems/maximal-rectangle/

3.17					
复习做出来					

在84题的基础上，按行来进行矩形面积的测量

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.size() == 0)
            return 0;
        int row = matrix.size();
        int col = matrix[0].size();
        vector<int> heights(col+2, 0);

        int max_area = 0;
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                if (matrix[i][j] == '1')
                    heights[j+1]++;
                else
                    heights[j+1] = 0;
            }
            max_area = max(max_area, maxArea(heights));
        }
        return max_area;
    }

    int maxArea(vector<int>& heights) {
        stack<int> stk;
        stk.push(0);
        int max_area = 0;

        for (int i = 1; i < heights.size(); ++i) {
            while (heights[stk.top()] > heights[i]) {
                int height = heights[stk.top()];
                stk.pop();
                if (height <= (i-stk.top()-1))
                    max_area = max(max_area, height * height);
                else
                    max_area = max(max_area, (i-stk.top()-1) * (i-stk.top()-1));

            }
```

```
            stk.push(i);
        }
        return max_area;
    }
};
```

239.滑动窗口最大值

https://leetcode-cn.com/problems/sliding-window-maximum/submissions/

3.17					
复习做出来					

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        vector<int> ans;

        for (int i = 0; i < nums.size(); ++i) {
            if (i < k-1) {
                while (!dq.empty() && dq.back() < nums[i])
                    dq.pop_back();
                dq.push_back(nums[i]);
            } else {
                while (!dq.empty() && dq.back() < nums[i])
                    dq.pop_back();
                dq.push_back(nums[i]);
                ans.push_back(dq.front());
                if (nums[i-k+1] == dq.front())
                    dq.pop_front();
            }
        }
        return ans;
    }
};
```

异位词

242.有效的字母异位词

https://leetcode-cn.com/problems/valid-anagram/

3.18					
复习做出来					

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.size() != t.size())
            return false;
        vector<int> dict(26, 0);
        for (char c : s)
            dict[c-'a']++;
        for (char d : t)
            dict[d-'a']--;
        // 此处因为开始进行了长度判断，排除了可能因为子集的问题造成判断不足
        //（比如t中把s出现的全减掉了，但是t还包含s中没有的，造成判断过少）
        for (char c : s)
            if (dict[c-'a'] != 0)
                return false;
        return true;
    }
};
```

49.字母异位词分组

https://leetcode-cn.com/problems/group-anagrams/

3.18					
复习做出来					

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> dict;
        vector<vector<string>> ans;
        for (string& str : strs) {
            string temp = str;
            sort(temp.begin(), temp.end());
            dict[temp].emplace_back(str);
        }
        for (auto& it : dict) {
```

```
            ans.emplace_back(it.second);
        }
        return ans;
    }
};
```

438.找到字符串中所有字母异位词

https://leetcode-cn.com/problems/find-all-anagrams-in-a-string/

3.18					
复习做出来					

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        if (s.size() < p.size())
            return {};
        unordered_map<char, int> need, windows;
        vector<int> ans;
        for (char c : p)    need[c]++;
        int left = 0, right = 0;
        int valid = 0;
        while (right < s.size()) {
            char c = s[right];
            right++;
            // 只更新关注的，就算有其他字符存进来，此时是不能满足长度条件的
            if (need.count(c)) {
                windows[c]++;
                if (windows[c] == need[c])
                    valid++;
            }
            // 收缩
            while (valid == need.size()) {
                if (right - left == p.size())
                    ans.push_back(left);
                char d = s[left];
                left++;
                // 更新
                if (need.count(d)) {
                    if (windows[d] == need[d])
                        valid--;
                windows[d]--;
                }
            }
        }
        return ans;
```

```
    }
};
```

1347、制造字母异位词的最小步骤数

https://leetcode-cn.com/problems/minimum-number-of-steps-to-make-two-strings-anagram/

3.18					
复习做出来					

```
class Solution {
public:
    int minSteps(string s, string t) {
        vector<int> dict(26, 0);
        for (char c : s)
            dict[c-'a']++;

        int count = 0;
        for (char c : t) {
            if (dict[c-'a'] == 0)
                count++;
            else // 保证上面dict[c]判断条件一直是0 即如果会继续被减还是会计数
                dict[c-'a']--;
        }
        return count;
    }
};
```