蓝桥杯复习提纲

算法

枚举

排序

快速排序

```cpp
void QuickSort(vector<int> array, int left, int right) {
    if (left >= right) return;
    int key = array[left];
    int low = left;
    int high = right;
    while (low < high) {
        while (low < high && array[high] >= key) {
            high--;
        }
        array[low] = array[high];
        while (low < high && array[low] <= key) {
            low++;
        }
        array[high] = array[low];
    }

    array[low] = key;
    QuickSort(array, left, low - 1);
    QuickSort(array, low + 1, right);
}
```

堆排序

```cpp
void SiftDown(vector<int> array, int i, int n){
    int left = 2*i+1,right = 2*i+2,min = i;
    if(left<n&&array[min]<array[left]){
        min = left;
    }
    if(right<n&&array[min]<array[right]){
        min = right;
    }
    if(min!=i){
        int t = array[min];
        array[min] = array[i];
        array[i] = t;
        SiftDown(array, min, n);
    }
}
void BuildHeap(int *array,int n){
    int p = n/2-1;
    for(int i = p;i>=0;i--){
        SiftDown(array, i, n);
    }
}
void HeapSort(int *array,int n){
    BuildHeap(array,n);
    for(int i = n-1;i>0;i--){
        int t = array[0];
        array[0] = array[i];
        array[i] = t;
        SiftDown(array, 0, i);
    }
}
```

归并排序

```
//基数排序
int  getMaxBit(vector<int> array, int n){//得到元素序列中最大数的位数
    int max = 1;
    int k = 10;
    for(int i = 0;i<n;i++){
        while(array[i]>=k){
            k*=10;
            max++;
        }
    }
    return max;
}
void RadixSort(int *array,int size) {
    int n;
    int max = getMaxBit(array, size);
    int maxNum = 1;
    for(int i = 1;i<max;i++){
        maxNum *=10;
    }
    for(int i=1;i<=maxNum;i=i*10) {
        int tmp[15][10]={0};//分配操作:建立一个15行，10列的数组，每一列分别代表0~9位数，15行代表能存放的总个数
        for(int j=0;j<size;j++) {
            n=(array[j]/i)%10;
            tmp[j][n]=array[j];
        }
        int k=0;//收集操作：将二维数组中的数据自左至右、自上至下收集到数组中
        for(int p=0;p<10;p++)
            for(int q=0;q<size;q++) {
                if(tmp[q][p]!=0)
                    array[k++]=tmp[q][p];
            }
        }
}
```

快速排序

```
quickSort(array,0,len-1);
void quickSort(int s[], int l, int r){
    if (l< r){
        int i = l, j = r, x = s[l];
        while (i < j){
            while(i < j && s[j]>= x) // 从右向左找第一个小于x的数
                j--;
            if(i < j)
                s[i++] = s[j];
            while(i < j && s[i]< x) // 从左向右找第一个大于等于x的数
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quickSort(s, l, i - 1); // 递归调用
        quickSort(s, i + 1, r);
    }
}
```

搜索

二分查找

```
lower_bound(store.begin(), store.end(), num)
```

```
int binarySearch(int[] nums, int target) {
    int left = 0, right = ...;

    while(...) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            ...
        } else if (nums[mid] < target) {
            left = ...
        } else if (nums[mid] > target) {
            right = ...
        }
    }
    return ...;
}
```

寻找左侧边界的二分查找

```
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length; // 注意
    // int right = nums.length - 1;

    while (left < right) { // 注意
    // while(left <= right)
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid; // 注意
            // right = mid - 1;
        }
    }
    return left;
}
int binary_search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if(nums[mid] == target) {
            // 直接返回
            return mid;
        }
    }
    // 直接返回
    return -1;
}

int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定左侧边界
            right = mid - 1;
        }
    }
    // 最后要检查 left 越界的情况
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}


int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定右侧边界
            left = mid + 1;
        }
    }
    // 最后要检查 right 越界的情况
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

34. 在排序数组中查找元素的第一个和最后一个位置

```
class Solution {
public:
    int left_bound(vector<int> &nums, int target) {
        if (nums.size() == 0) return -1;
        int left = 0;
        int right = nums.size()-1; // 注意

        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1; // 注意
                // right = mid - 1;
            }
        }
        if(left == nums.size() || nums[left] != target) return -1;
        return left;
    }
    int right_bound(vector<int> nums, int target) {
        if (nums.size() == 0) return -1;
        int left = 0;
        int right = nums.size()-1; // 注意

        while (left <= right) {
            int mid = (left + right) / 2;
```

```cpp
            if (nums[mid] == target) {
                left = mid + 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1; // 注意
            }
        }
        if(left == 0 || nums[left-1] != target) return -1;
        return left-1;
    }
    vector<int> searchRange(vector<int>& nums, int target) {
        return {left_bound(nums, target), right_bound(nums, target)};
    }
};
```

BFS:

752. 打开转盘锁

```cpp
class Solution {
public:
    int openLock(vector<string>& deadends, string target) {
        unordered_map<string, bool> dead;
        for(auto d : deadends) dead[d] = true;

        // char c = '1';
        // cout << (char)('0' + (c - '0' - 1 + 10) % 10) << endl;
        // c = '1';
        // cout << (char)('0' + (c - '0' + 1) % 10) << endl;

        if(dead.count("0000")) return -1;

        set<string> vis;
        queue<pair<string, int> > q;
        q.push({"0000", 0});
        vis.insert("0000");
        while(!q.empty()){
            auto u = q.front(); q.pop();
            string s = u.first;
            int cnt = u.second;
            if(s == target) return cnt;

            for(int i = 0; i < 4; i++){
                char c = s[i], t = s[i];
                s[i] = (char)('0' + (c - '0' + 1) % 10);
                if(!vis.count(s) && !dead.count(s)){
                    q.push({s, cnt+1});
                    vis.insert(s);
                }
                s[i] = (char)('0' + (c - '0' - 1 + 10) % 10);
                if(!vis.count(s) && !dead.count(s)){
                    q.push({s, cnt+1});
                    vis.insert(s);
                }
                s[i] = t;
            }
        }
        return -1;
    }
};
```

双向BFS：

```
class Solution {
public:
    int openLock(vector<string>& deadends, string target) {
        unordered_map<string, bool> dead;
        for(auto d : deadends) dead[d] = true;

        if(dead.count("0000")) return -1;
        set<string> q1, q2;
        set<string> vis;

        q1.insert("0000");
        q2.insert(target);

        int ans = 0;
        while(!q1.empty() && !q2.empty()){
            set<string> t;
            for(string s : q1){
                if(q2.count(s)) return ans;
                vis.insert(s);
                for(int i = 0; i < 4; i++){
                    char c = s[i], t1 = s[i];

                    s[i] = (char)('0' + (c - '0' + 1) % 10);
                    if(!vis.count(s) && !dead.count(s)) t.insert(s);

                    s[i] = (char)('0' + (c - '0' - 1 + 10) % 10);
                    if(!vis.count(s) && !dead.count(s)) t.insert(s);

                    s[i] = t1;
                }
            }
            q1 = q2;
            q2 = t;
            ans ++;
        }
        return -1;
    }
};
```

双指针

链表

141. 环形链表

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head) return false;
        ListNode *slow = head, *fast = head;
        while(fast != nullptr && fast->next != nullptr){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) return true;
        }
        return false;
    }
};
```

142. 环形链表 II

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {

        ListNode *slow = head, *fast = head;
        while(fast != nullptr && fast->next != nullptr){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast){
                slow = head;
                while(slow != fast){
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
       return NULL;

    }
};
```

83. 删除排序链表中的重复元素

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(head == nullptr) return nullptr;
        ListNode* fast = head, *slow = head;
        while(fast != nullptr){
            if(fast->val != slow->val){
                slow = slow->next;
                slow->val = fast->val;
            }
            fast=fast->next;
        }
        slow->next = nullptr;
        return head;
    }
};
```

数组

26. 删除排序数组中的重复项

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.size() == 0) return 0;

        int slow = 0, fast = 0;
        while(fast < nums.size()){
            if(nums[slow] != nums[fast]){
                slow++;
                nums[slow] = nums[fast];
            }
            fast++;
        }
        return slow + 1;
    }
};
```

27. 移除元素

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        if(nums.size() == 0) return 0;
        int fast = 0, slow = 0;
        while(fast < nums.size()){
            if(nums[fast] != val){
                // 注意和上一道题目两行代码的顺序
                nums[slow] = nums[fast];
                slow++;
            }
            fast++;
        }
        // 不同
        return slow;
    }
};
```

283. 移动零

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

滑动窗口

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;
```

```cpp

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

76. 最小覆盖子串

```cpp
class Solution {
public:
    unordered_map<char, int> cnt, req;
    bool judge(){
        for(auto c : req){
            if(cnt[c.first] < c.second) return false;
        }
        return true;
    }
    string minWindow(string s, string t) {
        for(auto c : t) req[c]++;
        int l = 0, r = 0;
        int ans = 0x3f3f3f3f, ansL = -1, ansR = -1;
        while(r < s.size()){
            if(req.find(s[r]) != req.end()) cnt[s[r]]++;
            while(judge() && l <= r){
                if(ans > r-l+1){
                    ans = r-l+1;
                    ansL = l; ansR = r;
                }
                if(req.find(s[l]) != req.end()) cnt[s[l]]--;
                ++l;
            }
            r++;
        }

        return ansL == -1 ? string() : s.substr(ansL, ans);
    }
};
```

3. 无重复字符的最长子串

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int l = 0, r = 0;
        int ans = 0;
        map<char, int> p;
        for(r; r < s.size(); r++){
            p[s[r]]++;
            while(p[s[r]] > 1){
                p[s[l++]]--;
            }
            ans = max(ans, r-l+1);
        }
        return ans;
    }
};
```

回溯

46. 全排列

```
do{

} while(next_permutation(nums.begin(), nums.end()))
```

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> result;
        do {
            result.emplace_back(nums);
        } while (next_permutation(nums.begin(), nums.end()));
        return result;
    }
};
```

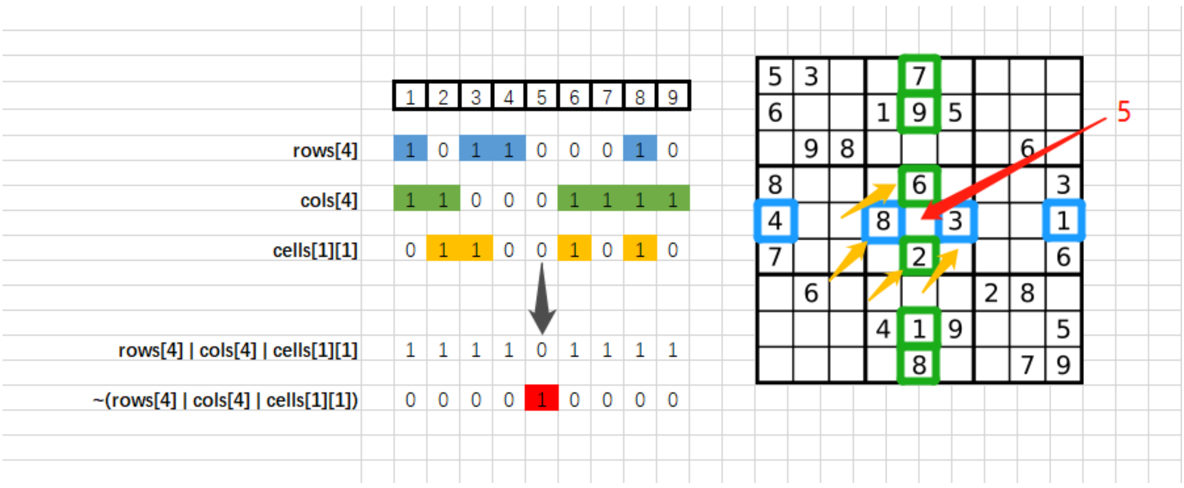
51. N 皇后

```cpp
// 二维
class Solution {
public:
    vector<vector<string> > ans;
    int n;
    bool check(vector<string> &queen, int row, int col){
        for(int i=0; i<row; i++){
            if(queen[i][col] == 'Q') return false;
        }
        for(int i=row-1, j=col-1; i>=0 && j>=0; j--, i--){
            if(queen[i][j]=='Q') return false;
        }
        for(int i=row-1, j=col+1; i>=0 && j<n; j++, i--){
            if(queen[i][j] == 'Q') return false;
        }
        return true;
    }
    void helper(vector<string> &queen, int row){
        if(row == n){
            ans.emplace_back(queen);
            return;
        }
        for(int i=0; i<n; i++){
            if(check(queen, row, i)){
                queen[row][i] = 'Q';
                helper(queen, row+1);
                queen[row][i] = '.';
            }
        }
    }
    vector<vector<string>> solveNQueens(int n) {
        this->n = n;
        vector<string> queen(n, string(n, '.'));
        helper(queen, 0);
        return ans;
    }
};
```

```cpp
// 一维

class Solution {
public:
    vector<vector<string> > ans;
    int n;
    bool check(vector<int> &queen, int row, int col){
        for(int i=0; i<row; i++){
            if(queen[i]==col || abs(queen[i]-col) == abs(i-row)) return false;
        }
        return true;
    }
    void helper(vector<int> &queen, int row){
        if(row == n){
            vector<string> tmp(n, string(n, '.'));
            for(int i=0; i<n; i++){
                tmp[i][queen[i]] = 'Q';
            }
            ans.emplace_back(tmp);
            return;
        }
        for(int i=0; i<n; i++){
            if(check(queen, row, i)){
                queen[row]=i;
                helper(queen, row+1);
                queen[row]=-1;
            }
        }
    }
    vector<vector<string>> solveNQueens(int n) {
        this->n = n;
        vector<int> queen(n, -1);
        helper(queen, 0);
        return ans;
    }
};

```

37. 解数独

```
bitset:
状态压缩
1. 使用 bitset<9> 来压缩存储每一行、每一列、每一个 3x3 宫格中 1-9 是否出现
2. 这样每一个格子就可以计算出所有不能填的数字，然后得到所有能填的数字 getPossibleStatus()
3. 填入数字和回溯时，只需要更新存储信息
4. 每个格子在使用时，会根据存储信息重新计算能填的数字

回溯
1. 每次都使用 getNext() 选择能填的数字最少的格子开始填，这样填错的概率最小，回溯次数也会变少
2. 使用 fillNum() 在填入和回溯时负责更新存储信息
3. 一旦全部填写成功，一路返回 true ，结束递归
```


1 2 3 4 5 6 7 8 9
rows[4] 1 0 1 1 0 0 0 1 0
cols[4] 1 1 0 0 0 1 1 1 1
cells[1][1] 0 1 1 0 0 1 0 1 0
rows[4] | cols[4] | cells[1][1] 1 1 1 1 0 1 1 1 1
~(rows[4] | cols[4] | cells[1][1]) 0 0 0 0 1 0 0 0 0
5 3 7
6 1 9 5
9 8 6
8 6 3
4 8 3 1
7 2 6
6 2 8
4 1 9 5
8 7 9
5

```
class Solution {
```

```cpp
public:
    vector<bitset<9> > rows, cols;
    vector<vector<bitset<9> > > cells;

    bitset<9> getPosibleStatus(int x, int y){
        return ~(rows[x] | cols[y] | cells[x/3][y/3]);
    }

    vector<int> getNext(vector<vector<char> > &board){

        vector<int> ans;
        int minCnt = 0x3f;
        for(int i=0; i<board.size(); i++){
            for(int j=0; j<board[i].size(); j++){
                if(board[i][j] != '.') continue;
                auto cur = getPosibleStatus(i, j);
                int c = cur.count();
                if(c < minCnt){
                    minCnt = c;
                    ans = {i, j};
                }
            }
        }

        return ans;
    }
    void fillNum(int x, int y, int n, bool flag){
        rows[x][n] = flag ? 1 : 0;
        cols[y][n] = flag ? 1 : 0;
        cells[x/3][y/3][n] = flag ? 1 : 0;
    }

    bool helper(vector<vector<char> > &board, int cnt){
        if(cnt == 0) return true;

        auto next = getNext(board);
        auto bits = getPosibleStatus(next[0], next[1]);

        for(int i=0; i<bits.size(); i++){
            if(!bits.test(i)) continue;
            int x = next[0], y = next[1];
            fillNum(x, y, i, true);
            board[x][y] = i+'1';
            if(helper(board, cnt-1)) return true;
            board[x][y] = '.';
            fillNum(x, y, i, false);
        }

        return false;
    }
    void solveSudoku(vector<vector<char>>& board) {
        rows = vector<bitset<9> > (9, bitset<9>());
        cols = vector<bitset<9> > (9, bitset<9>());
        cells = vector<vector<bitset<9> > > (3, vector<bitset<9> >(3,
bitset<9>()));


        int cnt = 0;
        for(int i=0; i<board.size(); i++){
            for(int j=0; j<board[i].size(); j++){
                cnt += (board[i][j] == '.');
                if(board[i][j] == '.') continue;

                int n=board[i][j] - '1';
                rows[i] |= (1<<n);
                cols[j] |= (1<<n);
                cells[i/3][j/3] |= (1<<n);
            }
        }

        helper(board, cnt);
    }
};
```

529. 扫雷游戏

```
class Solution {
public:
    int m, n;
    int dist[8][2] = {1, 0, -1, 0, 0, 1, 0, -1, 1, -1, -1, 1, 1, 1, -1, -1};
    void bfs(vector<vector<char>>& board, int sr, int sc){
        queue<pair<int, int> > q;
        bool vis[55][55];
        memset(vis, false, sizeof vis);
        vis[sr][sc] = true;
        q.push({sr, sc});
        while(!q.empty()){
            auto u = q.front(); q.pop();
            int x=u.first, y=u.second;
            int cnt=0;
            for(int i=0; i<8; i++){
                int x1=x+dist[i][0];
                int y1=y+dist[i][1];
                if(x1<0||y1<0||x1>=m||y1>=n) continue;
                cnt += board[x1][y1] == 'M';
            }

            if(cnt){
                board[x][y] = cnt+'0';
            }else{
                board[x][y] = 'B';

                for(int i=0; i<8; i++){
                    int x1=x+dist[i][0];
                    int y1=y+dist[i][1];
                    if(x1<0||y1<0||x1>=m||y1>=n || vis[x1][y1] || board[x1]
[y1]!='E') continue;
                    q.push({x1, y1});
                    vis[x1][y1] = true;
                }
            }
        }
    }
    vector<vector<char>> updateBoard(vector<vector<char>>& board,
vector<int>& click) {
        m=board.size(); n=board[0].size();
        if(board[click[0]][click[1]] == 'M'){
            board[click[0]][click[1]] = 'X';
        }else{
            bfs(board, click[0], click[1]);
        }
        return board;
    }
};
```

679. 24 点游戏

```
class Solution {
public:
    static constexpr int TARGET = 24;
    static constexpr double EPSILON = 1e-6;
    static constexpr int ADD = 0, MULTIPLY = 1, SUBTRACT = 2, DIVIDE = 3;
    bool judgePoint24(vector<int>& nums) {
        vector<double> ans;
        for(int num : nums) ans.emplace_back(static_cast<double>(num));
        return helper(ans);
    }

    bool helper(vector<double> nums){
        if(nums.size() == 0) return false;
        if(nums.size() == 1) return fabs(TARGET-nums[0])<=EPSILON;

        int sz = nums.size();

        for(int i=0; i<sz; i++){
            for(int j=0; j<sz; j++){
                if(i==j) continue;
                vector<double> l;
                for(int k=0; k<sz; k++){
                    if(k==j || k==i) continue;
                    l.emplace_back(nums[k]);
                }

                for (int k = 0; k < 4; k++) {
                    /*
                    加法和乘法都满足交换律，因此如果选择的运算操作是加法或乘法，
                    则对于选出的 22 个数字不需要考虑不同的顺序，
                    在遇到第二种顺序时可以不进行运算，直接跳过。
                    */
                    if (k < 2 && i > j)  continue;
                    if (k == ADD) l.emplace_back(nums[i] + nums[j]);
                    else if (k == MULTIPLY) l.emplace_back(nums[i] *
nums[j]);
                    else if (k == SUBTRACT) l.emplace_back(nums[i] -
nums[j]);
                    else if (k == DIVIDE) {
                        // 除法为零
                        if (fabs(nums[j]) < EPSILON) continue;
                        l.emplace_back(nums[i] / nums[j]);
                    }
```

```cpp
                    if (helper(l)) return true;
                    l.pop_back();
                }
            }
        }
        return false;
    }
};
```

488. 祖玛游戏

```
1. 如果后面的球与前面的球颜色不一样，在这里尝试插入一个后面颜色的球
2. 如果相邻的两个球颜色相同，考虑在中间插入一个其他颜色的球，将他们分割
```

代码：

```cpp
class Solution {
public:
    int findMinStep(string board, string hand) {
        cnt = hand.size();
        for (auto c : hand) {
            h[c - 'A']++;
        }

        dfs(board, 0);
        return ans == INT_MAX ? -1 : ans;
    }

    void dfs(string board, int step) {
        shoot(board);

        if (board.empty()) {
            ans = min(ans, step);
        }
        if (step == cnt) return;
        if (step >= ans) return;

        set<pair<int, char>> ins;
        for (int i = 0; i < board.size(); i++) {
            int t = board[i] - 'A';
            if (i == 0 || board[i] != board[i - 1]) {
                if (h[t] != 0) {
                    ins.insert({ i, 'A' + t });
                }
            }
            if (i != 0 && board[i] == board[i - 1]) {
                for (int j = 0; j < h.size(); j++) {
                    if (j == t || h[j] == 0) continue;
                    ins.insert({ i, 'A' + j });
                }
            }
        }

        for (auto[i, c] : ins) {
            h[c - 'A']--;
            board.insert(i, 1, c);
```

```
                dfs(board, step + 1);
                board.erase(i, 1);
                h[c - 'A']++;
            }
        }

        void shoot(string& board) {
            for (int i = 0; i < (int)board.size() - 2; i++) {
                int j = i + 1;
                while (j < board.size() && board[i] == board[j]) j++;
                if (j - i < 3) {
                    i = j - 1;
                    continue;
                }
                board.erase(i, j - i);
                shoot(board);
                break;
            }
        }

    private:
        int ans = INT_MAX;
        int cnt = 0;
        vector<int> h = vector<int>(26, 0);
    };

```

计数

进制转换

```
void dec2bin(int num){
    stack<int> bin;
    while(num!=0){
        bin.push(num % 2);
        num /= 2;
    }
    return bin;
}
```

卡特兰数

```
int n, f[19]={1,1};
cin >> n;
for( int i=2;i<=n;++i ){
    for( int j=0;j<i;++j ){
        f[i] += f[j]*f[i-j-1];
    }
}
```

求二进制中一的个数

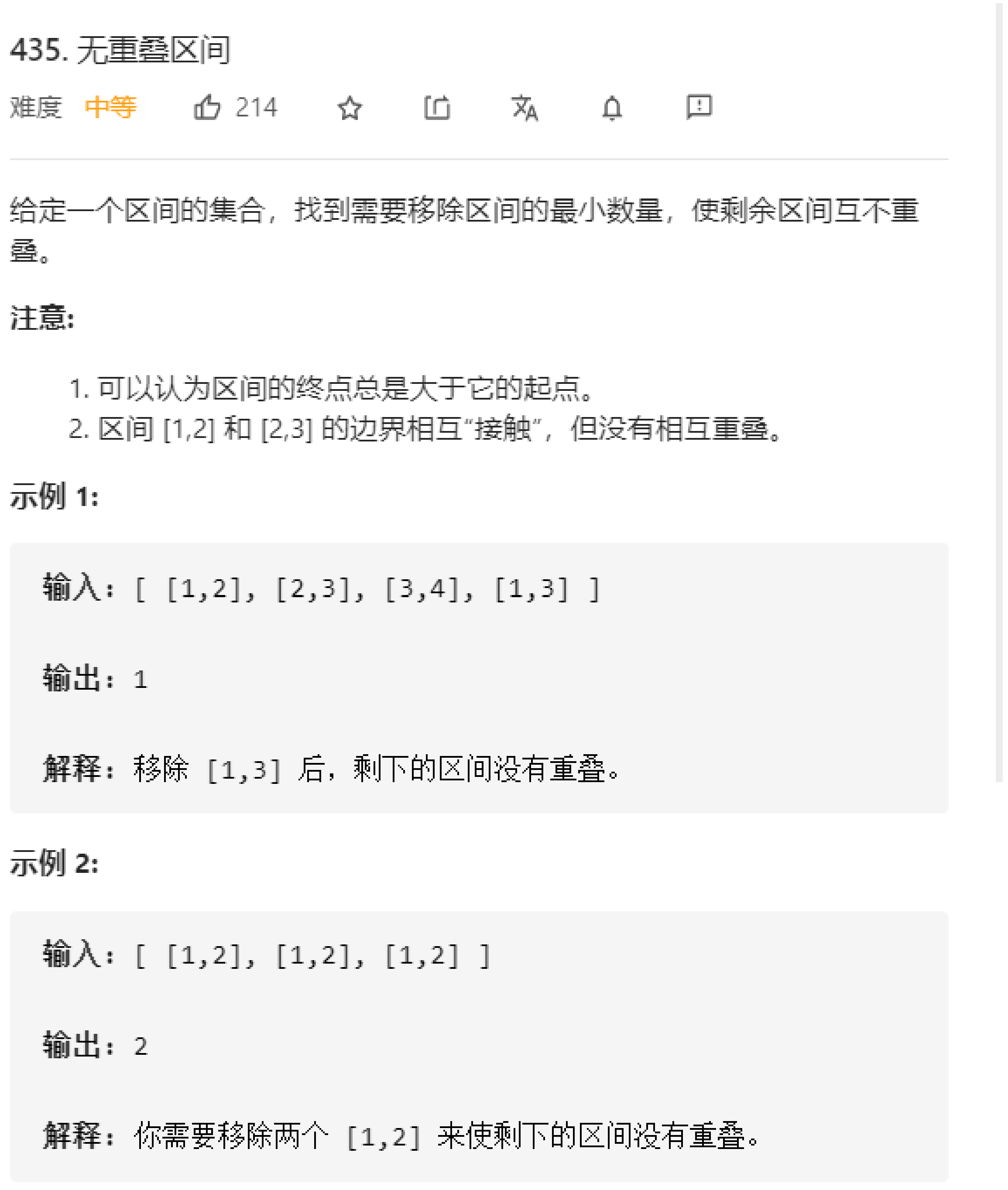
```
//去掉最低位一个1
x &= (x-1)
```

```
/*利用位移运算来求解3，不会引起死循环，而且循环次数少，有几个1就循环几次*/
int NumberOf1(int n) {
    int count = 0;
    while (n){
        count++;
        n = (n - 1) & n;
    }
    return count;
}
```

贪心

区间调度

[435. 无重叠区间]



435. 无重叠区间

难度 中等 214

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

1. 可以认为区间的终点总是大于它的起点。
2. 区间 [1,2] 和 [2,3] 的边界相互"接触"，但没有相互重叠。

示例 1:

```
输入: [ [1,2], [2,3], [3,4], [1,3] ]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。
```

示例 2:

```
输入: [ [1,2], [1,2], [1,2] ]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。
```

代码:

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

        sort(intervals.begin(), intervals.end(), [](vector<int> &a, vector<int> &b) -> bool {
            return a[1] < b[1];
        });

        int end = intervals[0][1];
        int res = -1;
        for(auto G:intervals){
            if(G[0] < end) res++;
            else end = G[1];
        }
        return res;
    }
};
```

区间问题

1288. 删除被覆盖区间

```
class Solution {
public:
    int removeCoveredIntervals(vector<vector<int>>& intervals) {
        sort(begin(intervals), end(intervals),
            [](const vector<int> &o1, const vector<int> &o2) {
                return o1[0] == o2[0] ? o2[1] < o1[1] : o1[0] < o2[0];
            }
        );

        int count = 0;
        int prev_end = 0;
        for (auto curr : intervals) {
            if (prev_end < curr[1]) {
                ++count;
                prev_end = curr[1];
            }
        }
        return count;
    }
};
```

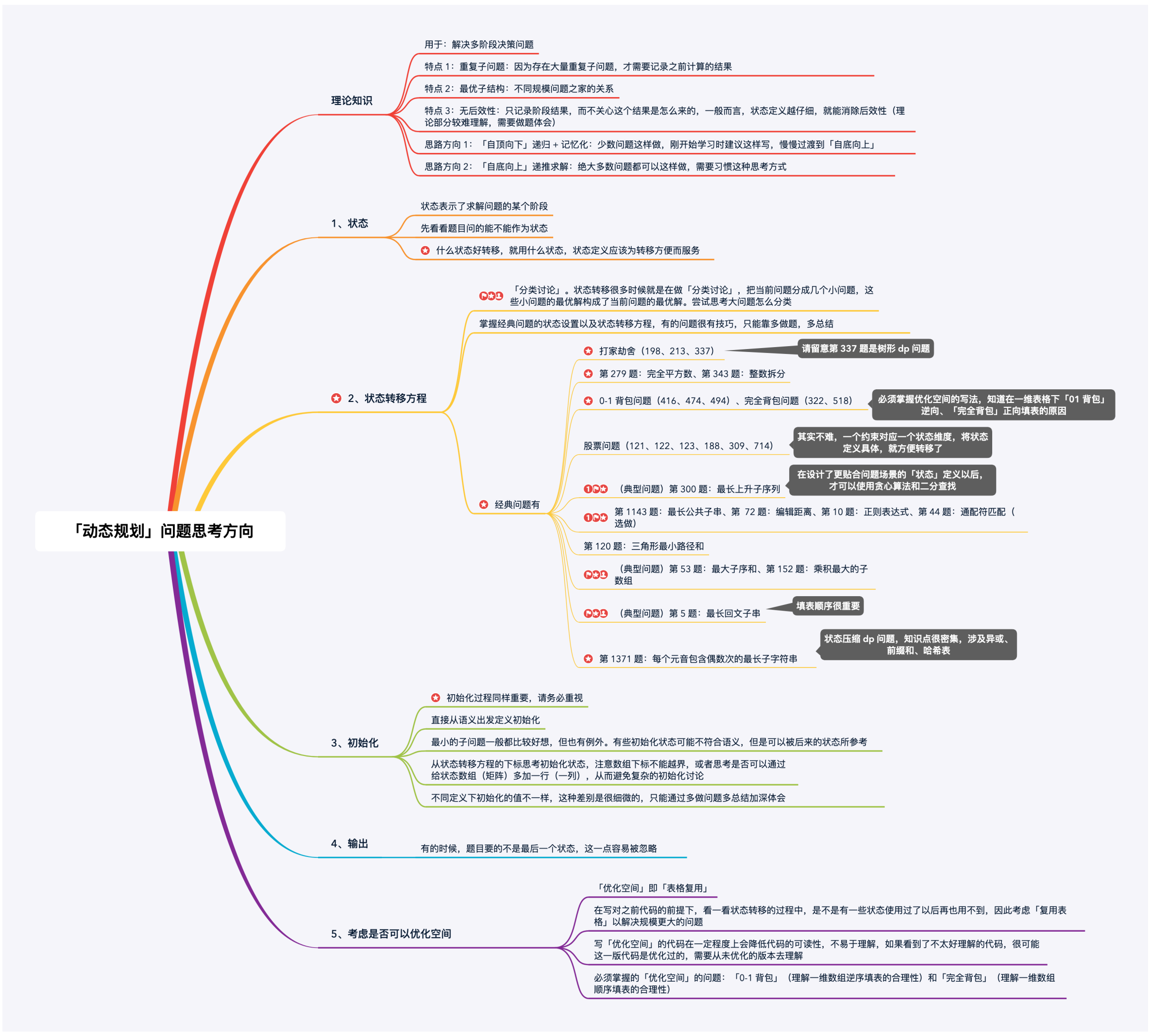
56. 合并区间

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if(intervals.size() == 0) return {};
        sort(intervals.begin(), intervals.end(), [](const vector<int> &a, const vector<int> &b){
            return a[0] == b[0] ? a[1] > b[1] : a[0] < b[0];
        });

        int prevEnd = 0;
        vector<vector<int> > ans;
        ans.emplace_back(intervals[0]);
        for(auto v : intervals){
            auto &last = ans.back();
            if(last[1] >= v[0]) last[1] = max(last[1], v[1]);
            else ans.emplace_back(v);
        }
        return ans;
    }
};
```

986. 区间列表的交集

```cpp
class Solution {
public:
    vector<vector<int>> intervalIntersection(vector<vector<int>>& A, vector<vector<int>>& B) {
        int i = 0, j = 0;
        vector<vector<int> > ans;

        while(i < A.size() && j < B.size()){
            auto i1 = A[i];
            auto j1 = B[j];

            if (j1[1] >= i1[0] && i1[1] >= j1[0]) ans.push_back({max(i1[0], j1[0]), min(i1[1], j1[1])});

            if(i1[1] < j1[1]) i++;
            else j++;
        }
        return ans;
    }
};
```

动态规划


「动态规划」问题思考方向
理论知识
用于：解决多阶段决策问题
特点 1：重复子问题：因为存在大量重复子问题，才需要记录之前计算的结果
特点 2：最优子结构：不同规模问题之家的关系
特点 3：无后效性：只记录阶段结果，而不关心这个结果是怎么来的，一般而言，状态定义越仔细，就能消除后效性（理论部分较难理解，需要做题体会）
思路方向 1：「自顶向下」递归 + 记忆化：少数问题这样做，刚开始学习时建议这样写，慢慢过渡到「自底向上」
思路方向 2：「自底向上」递推求解：绝大多数问题都可以这样做，需要习惯这种思考方式
1、状态
状态表示了求解问题的某个阶段
先看看题目问的能不能作为状态
什么状态好转移，就用什么状态，状态定义应该为转移方便而服务
2、状态转移方程
「分类讨论」。状态转移很多时候就是在做「分类讨论」，把当前问题分成几个小问题，这些小问题的最优解构成了当前问题的最优解。尝试思考大问题怎么分类
掌握经典问题的状态设置以及状态转移方程，有的问题很有技巧，只能靠多做题，多总结
经典问题有
打家劫舍（198、213、337）
请留意第 337 题是树形 dp 问题
第 279 题：完全平方数、第 343 题：整数拆分
0-1 背包问题（416、474、494）、完全背包问题（322、518）
必须掌握优化空间的写法，知道在一维表格下「01 背包」逆向、「完全背包」正向填表的原因
股票问题（121、122、123、188、309、714）
其实不难，一个约束对应一个状态维度，将状态定义具体，就方便转移了
（典型问题）第 300 题：最长上升子序列
在设计了更贴合问题场景的「状态」定义以后，才可以使用贪心算法和二分查找
第 1143 题：最长公共子串、第 72 题：编辑距离、第 10 题：正则表达式、第 44 题：通配符匹配（选做）
第 120 题：三角形最小路径和
（典型问题）第 53 题：最大子序和、第 152 题：乘积最大的子数组
（典型问题）第 5 题：最长回文子串
填表顺序很重要
第 1371 题：每个元音包含偶数次的最长子字符串
状态压缩 dp 问题，知识点很密集，涉及异或、前缀和、哈希表
3、初始化
初始化过程同样重要，请务必重视
直接从语义出发定义初始化
最小的子问题一般都比较好想，但也有例外。有些初始化状态可能不符合语义，但是可以被后来的状态所参考
从状态转移方程的下标思考初始化状态，注意数组下标不能越界，或者思考是否可以通过给状态数组（矩阵）多加一行（一列），从而避免复杂的初始化讨论
不同定义下初始化的值不一样，这种差别是很细微的，只能通过多做问题多总结加深体会
4、输出
有的时候，题目要的不是最后一个状态，这一点容易被忽略
5、考虑是否可以优化空间
「优化空间」即「表格复用」
在写对之前代码的前提下，看一看状态转移的过程中，是不是有一些状态使用过了以后再也用不到，因此考虑「复用表格」以解决规模更大的问题
写「优化空间」的代码在一定程度上会降低代码的可读性，不易于理解，如果看到了不太好理解的代码，很可能这一版代码是优化过的，需要从未优化的版本去理解
必须掌握的「优化空间」的问题：「0-1 背包」（理解一维数组逆序填表的合理性）和「完全背包」（理解一维数组顺序填表的合理性）

背包系列

思路：

```
for (int i = 1; i <= n; i++) {
    // 01背包：逆序遍历
    // for (int j = m; j >= w[i]; j--) {
    // 完全背包：正向遍历
    for (int j = w[i]; j <= m; j++) {
        maxValue[j] = max(maxValue[j], maxValue[j - w[i]] + v[i]);
    }
}
```

494. 目标和

01背包(内外层循环的顺序)

```
class Solution {
public:
    int subsets(vector<int>& nums, int target){
        vector<int> dp(target+10, 0);
        dp[0] = 1;
        for(int i = 0; i < nums.size(); i++){
            for(int j = target; j >= 0; j--){
                if(j - nums[i] >= 0) dp[j] += dp[j-nums[i]];
```

```cpp
                else dp[j] = dp[j];
            }
        }
        return dp[target];
    }
    int findTargetSumWays(vector<int>& nums, int S) {
        int sum = 0;
        for(int num : nums) sum += num;
        // 2 * A = S + sum(nums) A是正数集合 B是负数集合
        if(sum < S || (sum+S)%2 == 1) return 0;
        return subsets(nums, (sum+S)/2);
    }
};
```

416. 分割等和子集

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

322. 零钱兑换

```cpp
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> dp(amount+10, 0x3f3f3f3f);
        dp[0] = 0;
        for(int i = 1; i <= amount; i++){
            for(auto &coin : coins)
                if(i - coin >= 0) dp[i] = min(dp[i], dp[i-coin]+1);
        }
        return dp[amount] == 0x3f3f3f3f ? -1 : dp[amount];
    }
};
```

股票系列：

121. 买卖股票的最佳时机

思路：

```
dp[-1][k][0] = 0
解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0 。
dp[-1][k][1] = -infinity
解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。
dp[i][0][0] = 0
解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0 。
dp[i][0][1] = -infinity
解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。
```

状态转移方程：

```
base case：
dp[-1][k][0] = dp[i][0][0] = 0
dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

代码：

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

    int maxProfit(int K, vector<int>& prices) {
        if(!prices.size()) return 0;
        if(K >= prices.size()/2) return maxProfit_inf(prices);

        vector<vector<vector<int> > > dp(prices.size() + 2,
vector<vector<int> >(K+1, vector<int>(2, 0)));
        for(int i = 0; i <= K; i++){
            dp[0][i][0] = 0; dp[0][i][1] = -0x3f3f3f3f;
        }

        for(int i = 1; i <= prices.size(); i++){
            for(int k = 1; k <= K; k++){
                dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i-
1]);
                dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i-
1]);
            }
        }
        return dp[prices.size()][K][0];
    }
};
```

打家劫舍系列

198. 打家劫舍

思路：

```
dp[i] = max(dp[i+1], dp[i+2]+nums[i]);
            选上一个， 或者选当前和上上一个
```

代码：

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if(nums.size() == 0) return 0;
        vector<int> dp(nums.size()+2, 0);
        for(int i = nums.size()-1; i>=0; i--){
            dp[i] = max(dp[i+1], dp[i+2]+nums[i]);
        }
        return dp[0];
    }
};
```

字符串DP

28. 实现 strStr()

KMP / DP

```
const int maxn = 1e5+10;
int dp[maxn][256];
class Solution {
public:
    void kmp(string pattern){
        int n = pattern.size();

        dp[0][pattern[0]] = 1;
        int X = 0;
        for(int i = 1; i < pattern.size(); i++){
            for(int j = 0; j < 256; j++)
                dp[i][j] = dp[X][j];
            dp[i][pattern[i]] = i + 1;
            X = dp[X][pattern[i]];
        }
    }
    int strStr(string haystack, string needle) {
        if(needle.size() == 0) return 0;
        memset(dp, 0, sizeof dp);
        int j = 0;
        kmp(needle);
        for(int i = 0; i < haystack.size(); i++){
            auto c = haystack[i];
            j = dp[j][c];
            if(j == needle.size()) return i - needle.size() + 1;
        }

        return -1;
    }
};
```

72. 编辑距离

```
int dp[1000][1000];
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        memset(dp, 0x3f, sizeof dp);
        for(int i = 0; i <= m; i++) dp[i][0] = i;
        for(int j = 0; j <= n; j++) dp[0][j] = j;
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++){
                if(word1[i-1] == word2[j-1]) dp[i][j] = dp[i-1][j-1];
                else dp[i][j] = min(dp[i-1][j], min(dp[i][j-1], dp[i-1][j-1])) + 1;
            }
        }
        return dp[m][n];
    }
};
```

651.四键键盘

假设你有一个特殊的键盘包含下面的按键：

`Key 1: (A)`：在屏幕上打印一个 'A'。

`Key 2: (Ctrl-A)`：选中整个屏幕。

`Key 3: (Ctrl-C)`：复制选中区域到缓冲区。

`Key 4: (Ctrl-V)`：将缓冲区内容输出到上次输入的结束位置，并显示在屏幕上。

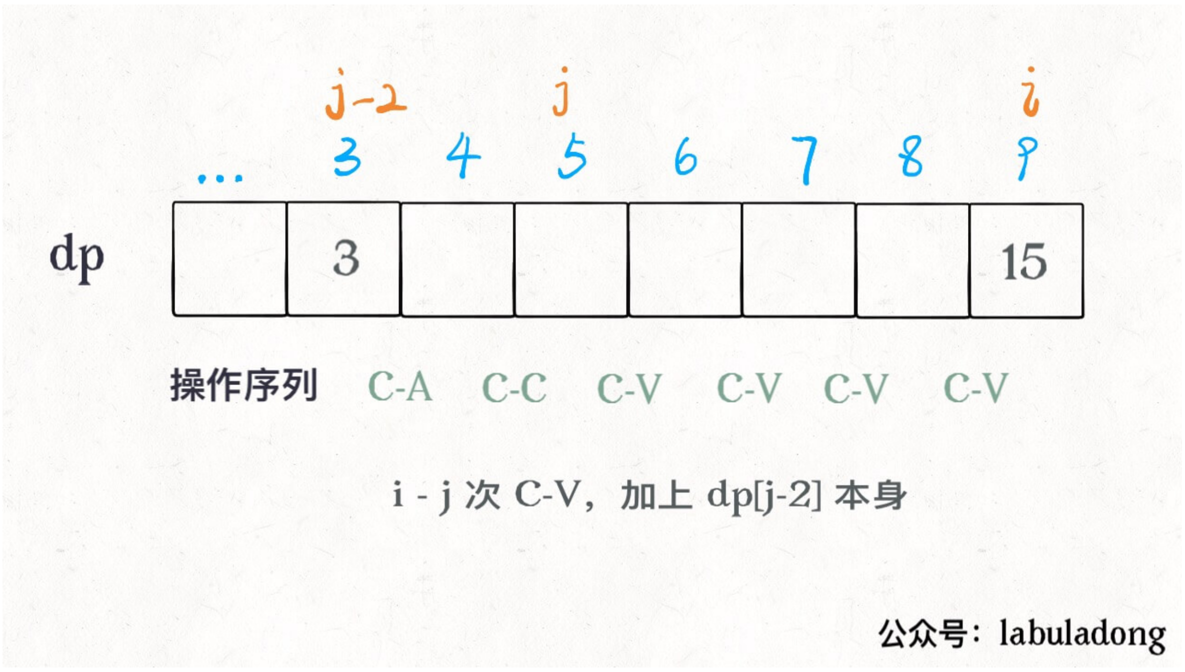
现在，你只可以按键 **N** 次（使用上述四种按键），请问屏幕上最多可以显示几个 'A'呢？

样例 1:

```
输入: N = 3
输出: 3
解释:
我们最多可以在屏幕上显示三个'A'通过如下顺序按键：
A, A, A
```

样例 2:

```
输入: N = 7
输出: 9
解释:
我们最多可以在屏幕上显示九个'A'通过如下顺序按键：
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
```


j-2
j
i
... 3 4 5 6 7 8 9
dp
3
15
操作序列 C-A C-C C-V C-V C-V C-V
i - j 次 C-V，加上 dp[j-2] 本身
公众号：labuladong

代码：

```
public int maxA(int N) {
    int[] dp = new int[N + 1];
    dp[0] = 0;
    for (int i = 1; i <= N; i++) {
        // 按 A 键
        dp[i] = dp[i - 1] + 1;
        for (int j = 2; j < i; j++) {
            // 全选 & 复制 dp[j-2]，连续粘贴 i - j 次
            // 屏幕上共 dp[j - 2] * (i - j + 1) 个 A
            dp[i] = Math.max(dp[i], dp[j - 2] * (i - j + 1));
        }
    }
    // N 次按键之后最多有几个 A?
    return dp[N];
}
```

5. 最长回文子串

```
class Solution {
public:
    string longestPalindrome(string s) {
        vector<vector<bool> > dp(s.size(), vector<bool>(s.size(), false));

        for(int i = 0; i < s.size(); i++) dp[i][i] = true;

        int begin = 0, maxSize = 1;
        for(int j = 1; j < s.size(); j++){
            for(int i = 0; i < j; i++){
                if(s[i] == s[j]){
                    if(j - i < 3) dp[i][j] = true;
                    else dp[i][j] = dp[i+1][j-1];

                }
                else dp[i][j] = false;

                if(dp[i][j] && maxSize < j-i+1){
                    maxSize = j-i+1;
                    begin = i;
                }
            }
        }

        return s.substr(begin, maxSize);
    }
};
```

516. 最长回文子序列

代码：

```
int dp[1010];
class Solution {
public:
    // int longestPalindromeSubseq(string s) {
```

```
    //     vector<vector<int> > dp(s.size()+10, vector<int>(s.size()+10, 0));
    //     for(int i = 0; i < s.size(); i++) dp[i][i] = 1;

    //     for(int i = s.size()-2; i >= 0; i--){
    //         for(int j = i+1; j < s.size(); j++){
    //             if(s[i] == s[j]) dp[i][j] = dp[i+1][j-1] + 2;
    //             else dp[i][j] = max(dp[i][j-1], dp[i+1][j]);
    //         }
    //     }
    //     return dp[0][s.size()-1];
    // }
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        for(int i = 0; i < n; i++) dp[i] = 1;

        for(int i = n-2; i >= 0; i--){
            int pre = 0;
            for(int j = i+1; j < n; j++){
                int t = dp[j];
                if(s[i] == s[j]) dp[j] = pre + 2;
                else dp[j] = max(dp[j-1], dp[j]);
                pre = t;
            }
        }
        return dp[s.size()-1];
    }
};
```

博弈

[877. 石子游戏]

```
class Solution {
public:
    bool stoneGame(vector<int>& piles) {
        int n = piles.size();
        vector<vector<int> > dp(n+1, vector<int>(n+1, 0));
        // dp[i][j] 定义：区间 piles[i..j] 内先手可以获得的净胜分
        for(int i = 0; i < n; i++) dp[i][i] = piles[i];
        for(int i = n-2; i>=0; i--){
            for(int j=i+1; j<n; j++){
                dp[i][j] = max(piles[i]-dp[i+1][j], piles[j]-dp[i][j-1]);
            }
        }
        return dp[0][n-1] > 0;
    }
};
```

区间DP

[312. 戳气球]

```cpp
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        int n = nums.size();
        vector<int> points(n+2);
        for(int i = 1; i <= n; i++) points[i] = nums[i-1];
        points[0] = points[n+1] = 1;
        vector<vector<int> > dp(n+2, vector<int>(n+2, 0));

        for(int i = n; i >= 0; i--){
            for(int j = i+1; j <= n + 1; j++){
                for(int k = i+1; k < j; k++){
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] +
points[i]*points[k]*points[j]);
                }
            }
        }
        return dp[0][n+1];
    }
};
```

树状DP

834. 树中距离之和

```cpp
class Solution {
public:
    vector<vector<int> > G;
    vector<int> ans;
    vector<int> dp, sz;
    void dfs(int u, int fa){
        dp[u] = 0;
        sz[u] = 1;
        for(auto v : G[u]){
            if(v == fa) continue;
            dfs(v, u);
            dp[u] += dp[v] + sz[v];
            sz[u] += sz[v];
        }
    }

    void dfs2(int u, int fa){
        ans[u] = dp[u];
        for(auto v : G[u]){
            if(v == fa) continue;
            int pu = dp[u], pv = dp[v];
            int su = sz[u], sv = sz[v];

            dp[u] -= dp[v] + sz[v];
            sz[u] -= sz[v];

            dp[v] += dp[u] + sz[u];
            sz[v] += sz[u];

            dfs2(v, u);
            dp[u] = pu; dp[v] = pv;
            sz[u] = su; sz[v] = sv;
        }
    }
    vector<int> sumOfDistancesInTree(int N, vector<vector<int>>& edges) {
        G = vector<vector<int> > (N);
        ans = vector<int>(N);
        dp = vector<int>(N, 0);
        sz = vector<int>(N, 0);
        for(auto e : edges){
            G[e[0]].emplace_back(e[1]);
            G[e[1]].emplace_back(e[0]);
        }
        dfs(0, -1);
        // for(auto v : dp) cout << v << " ";
        dfs2(0, -1);
        return ans;
    }
};
```

状压DP

1349. 参加考试的最大学生数

```
int dp[10][1<<8];

//dp[i][bits] 表示前i行中，第i行作为情况为bits的最大答案
//bits 011011011 1: 有人座，  0: 无人座
class Solution {
public:
    int lowbit(int x){
        return x&-x;
    }
    int getcount(int x){
        int res=0;
        while(x>0){
            ++res;
            x-=lowbit(x);
        }
        return res;
    }
    int maxStudents(vector<vector<char>>& seats) {
        int n=seats.size(), m=seats[0].size();

        memset(dp, -1, sizeof(dp));
        dp[0][0] = 0;

        int lim = (1<<m);
        for(int i=1; i<=n; i++){
            for(int cur=0; cur<lim ;cur++){
                for(int pre = 0; pre < lim; pre++){

                    if(dp[i-1][pre] == -1)
                        continue;

                    bool flg=0;
                    for(int j=0; j<m; j++){
                        if(((cur>>j)&1) == 0) continue;
                        if(seats[i-1][j] == '#') flg=1;
                        if(j>=1 && ((cur>>(j-1))&1)) flg=1;
                        if(j<m-1 && ((cur>>(j+1))&1)) flg=1;
                        if(j >= 1 && ((pre>>(j-1))&1)) flg=1;
                        if(j<m-1 && ((pre >> (j+1))&1)) flg=1;
                    }
                    if(flg){
                        continue;
                    }
                    dp[i][cur] = max(dp[i][cur], dp[i-1][pre] +
getcount(cur));
                }
            }
        }

        int ans=0;
        for(int i=0; i<lim; i++)
            ans = max(ans, dp[n][i]);
        return ans;
    }
};
```

968. 监控二叉树

```cpp
/**
 * 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:
    /* 状态转移:
    a: root必须放置摄像头的情况下，覆盖整棵树需要的摄像头数目。
    b: 覆盖整棵树需要的摄像头数目，无论 root 是否放置摄像头。
    c: 覆盖两棵子树需要的摄像头数目，无论节点 root 本身是否被监控到。
    */

    vector<int> dfs(TreeNode* root){
        if(!root) return {INT_MAX/2, 0, 0};
        auto l = dfs(root->left);
        auto r = dfs(root->right);
        int a = l[2] + r[2] + 1;
        int b = min(a, min(l[1]+r[0], l[0]+r[1]));
        int c = min(a, l[1]+r[1]);
        // cout << a << " " << b << " " << c << endl;
        return {a, b, c};
    }
    int minCameraCover(TreeNode* root) {
        if(!root) return 0;
        auto ans = dfs(root);
        return ans[1];
    }
};
```

53. 最大子序和

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        vector<int> dp(nums.size(), -0x3f3f);
        int res = nums[0];
        dp[0] = nums[0];
        for(int i = 1; i < nums.size(); i++){
            dp[i] = max(nums[i], nums[i] + dp[i-1]);
            res = max(res, dp[i]);
        }
        return res;
    }
};
```

300. 最长上升子序列

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

1143. 最长公共子序列

```
int dp[1010];
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.size(), n = text2.size();

```

```cpp
        //vector<vector<int> > dp(m+1, vector<int>(n+1, 0));
        for(int i = 0; i <= n; i++) dp[i] = 0;

        // for(int i = 1; i <= m; i++){
        //     for(int j = 1; j <= n; j++){
        //         if(text1[i-1] == text2[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
        //         else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        //     }
        // }

        for(int i = 1; i <= m; i++){
            int pre = 0;
            for(int j = 1; j <= n; j++){
                int t = dp[j];
                if(text1[i-1] == text2[j-1]) dp[j] = pre + 1;
                else dp[j] = max(dp[j], dp[j-1]);
                pre = t;
            }
        }
        return dp[n];
    }
};
```

1567. 乘积为正数的最长子数组长度

```cpp
const int maxn = 1e5+10;
int dp[maxn][2];
class Solution {
public:
    int getMaxLen(vector<int>& nums) {
        memset(dp, 0, sizeof dp);
        int ans = 0;
        for(int i=1; i<=nums.size(); i++){
            if(nums[i-1] > 0){
                dp[i][0] = dp[i-1][0] + 1;
                dp[i][1] = dp[i-1][1] ? dp[i-1][1]+1 : 0;
            }
            if(nums[i-1] < 0){
                dp[i][0] = dp[i-1][1] ? dp[i-1][1]+1 : 0;
                dp[i][1] = dp[i-1][0] + 1;
            }
            //cout << dp[i][0] << " " << dp[i][1] << endl;
            ans = max(ans, dp[i][0]);
        }
        return ans;
    }
};
```

数论

博弈

巴什博弈

简述

什么是**巴什博弈**：只有一堆n个物品，两个人轮流从这堆物品中取物， 规定每次至少取一个，最多取m个。最后取光者得胜。

分析

我们称先进行游戏的人为先手，另一个人为后手。

1、如果$n = m + 1$，那么由于一次最多只能取m个，所以，无论先手拿走多少个，后手都能够一次拿走剩余的物品，后者取胜。
2、如果$n = (m + 1) * r + s$ (r为任意自然数，$s \leq m$,先手要拿走s个物品，如果后手拿走$k(k \leq m)$个，那么先手再拿走$m + 1 - k$个，结果剩下$(m + 1) * (r - 1)$个，以后保持这样的取法，那么先取者肯定获胜。我们得到如下结论:**要保持给对手留下$(m + 1)$的倍数，就能最后获胜。**

必胜态必败态

只要n不能整除$m + 1$,那么必然是先手取胜，否则后手取胜。

变形

如果我们规定最后取光者输，那么又会如何呢?

$(n - 1)\%(m + 1) == 0$则后手胜利 先手会重新决定策略，所以不是简单的相反的。

威佐夫博奕

简述

威佐夫博弈(Wythoff Game)：有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

分析

我们用$（a_k，b_k）（a_k \leq b_k, k = 0，1，2，..., n)$表示两堆物品的数量并称其为局势，如果甲面对$（0，0）$，那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是：$（0，0）$、$（1，2）$、$（3，5）$、$（4，7）$、$（6，10）$、$（8，13）$、$（9，15）$、$（11，18）$、$（12，20）$。 可以看出：$a_0 = b_0 = 0$, a_k是未在前面出现过的最小自然数,而 $b_k = a_k + k$。

必胜态必败态

满足$a_k = k *（1 + 5\surd）/2，b_k = a_k + k$，后手必胜，否则先手必胜。

Nim博弈

简述

通常的Nim游戏的定义是这样的：有若干堆石子，每堆石子的数量都是有限的，合法的移动是: 选择一堆石子并拿走若干颗（不能不拿），如果轮到某个人时所有的石子堆都已经被拿空了，则判负（因为他此刻没有任何合法的移动）。

必胜态必败态

对于一个Nim游戏的局面$(a1, a2, \ldots, an)$，它是$P - position$当且仅当$a_1 \oplus a_2 \oplus \ldots \oplus a_n = 0$，其中$\oplus$ 表示异或(xor)运算。

排列组合

数论

LCM / GCD

```
int gcd(int a, int b) {
    return b ? gcd(b , a % b) : a;
}

int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
```

BigNum

扩展欧几里得

`定理：`对于不完全为 0 的非负整数 a，b，gcd（a，b）表示a，b 的最大公约数，必然存在整数对 x，y，使得 gcd（a，b）$= a * x + b * y$。

```
int e_gcd(int a, int b, int &x, int &y){
    if(!b){
        x=1; y=0;
        return a;
    }
    int gcd = e_gcd(b, a % b, y, x);
    y -= a / b * x;
    return gcd;
}
```

素数筛

```
void get_prime(vector<int> &prime, int n){
    vector<bool> is_prime(n + 1, true);
    if (n < 2) return;
    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            prime.push_back(i);
            for (int j = i * i; j <= n; j += i) is_prime[j] = false;
        }
    }
}
```

快速幂

```
#define ll long long
ll pow (ll x, ll n){
    ll ans = 1.0;
    while(n > 0){
        if(n & 1) ans *= x;
        n /= 2;
        x *= x;
    }
    return ans;
}
```

费马小定理

`费马小定理`: p为质数， a为任意自然数，则$a^p \equiv a \ (mod \ p)$

逆元

$a * x \equiv 1 (mod \ p)$

中文描述: a乘一个数x并模p等于1

如果要求结果mod一个大质数，若原本的结果中有除法，比如除以a,那就可以乘以a的逆元来替代

扩展欧几里得求逆元:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

void exgcd(ll a,ll b,ll& d,ll& x,ll& y) {
    if(!b) { d = a; x = 1; y = 0; }
    else{ exgcd(b, a%b, d, y, x); y -= x*(a/b); }
}

ll inv(ll a, ll p) {
    ll d, x, y;
    exgcd(a, p, d, x, y);
    return d == 1 ? (x + p) % p : -1;
}

int main()
{
    ll a,p;
    while(1)
    {
        scanf("%lld %lld",&a,&p);
        printf("%lld\n",inv(a,p));
    }
}
```

费马小定理求逆元：

```
ll power_mod(ll a, ll b, ll mod)
{
    ll ans = 1;
    while (b)
    {
        if (b & 1)
            ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
inv2 = power_mod(a, mod - 2, mod);
```

欧拉定理求逆元：

```
int eurler_phi(int n)
{
    int res = n;
    for(int i = 2; i * i <= n; i++){
        if(n % i == 0){
            res = res / i * (i - 1);
            while(n % i == 0) n /= i;
        }
    }
    if(n != 1) res = res / n * (n - 1);
    return res;
}
```

约瑟夫环

```cpp
class Solution {
public:
    int lastRemaining(int n, int m) {
        int f = 0;
        for (int i = 2; i != n + 1; ++i)
            f = (m + f) % i;
        return f;
    }
};
```

递归：

```cpp
int josephus(int n, int m) {
    if(n == 1)  return 0;
    else return (josephus(n - 1, m) + m) % n;
}
```

图论

最小生成树

Kruskal

```cpp
int n,m;
struct edge{
    int x,y,v;
}e[maxm];
int cmp(edge a,edge b)
{
    return a.v<b.v;
}
int fa[maxn];
int find(int x)
{
    if(x==fa[x])
        return x;
    x=find(fa[x]);
    return fa[x];
}
int Kruskal()
{
    for(int i=1;i<=n;i++) fa[i]=i;
    sort(e+1,e+m+1,cmp);
    int cnt=0,ans=0;
    for(int i=1;i<=m;i++)
    {
        int fx=find(e[i].x),fy=find(e[i].y);
        if(fx==fy)  continue;
        fa[fx]=fy;
        ++cnt;
        ans+=e[i].v;
        if(cnt==n-1) break;

    }
    return ans;
}
```

Prim

```
#include<iostream>
#include<vector>
#include <climits>
using namespace std;
int main(){
    int n;
    cin>>n;
    vector<vector<int>>m(n,vector<int>(n));
    vector<int>lowest(n),vertex(n,0);
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cin>>m[i][j];
            if(!n){
                lowest[j]=m[i][j];//将0号节点插入最小构造树中
            }
        }
    }
    vertex[0]=1;//标记0号节点
    int totalcost=0;
    for(int i=0;i<n;i++){
        int mincost=INT_MAX,index=0;
        for(int j=0;j<n;j++){//寻找与树中已有点相连的最短边
            if(!vertex[j]&&lowest[j]&&lowest[j]<mincost){
                mincost=lowest[j];
                index=j;
            }
        }
        vertex[index]=1;//标记新加入树的点
        totalcost+=lowest[index];
        for(int j=0;j<n;j++){
            if(m[index][j]&&!vertex[j]&&(m[index][j]<lowest[j]||!lowest[j]))
{//利用新加入树的点刷新最短边
                lowest[j]=m[index][j];
            }
        }
    }
    cout<<totalcost<<endl;
}
```

拓扑排序

```
void toporder(int n) {
    queue<int>q;
    for (int i = 0; i < n; i++) {
        if (!deg[i]) q.push(i);
        // deg[i] 表示第i个点的度数，这里先把度数为0的点加入队列
    }
    while (!q.empty()) {
        int u = q.front(); q.pop();
        // u是排到的点，这里根据情况写
        for (int i = 0; i < (int)G[u].size(); i++) {
            int v = G[u][i];
            if (!--deg[v]) {
                q.push(v);
            }
        }
    }
}
```

最短路

Floyd

```
#include<stdio.h>
#define MAX 100000
int main()
{
    int n;
    int arcs[10][10],path[10][10];//pat[i][j]=k 表示从i到j会经过k
    FILE *fp=fopen("floyd_data.txt","r");
    if(fp==NULL)
    {
        printf("open file error\n");
        return 0;
    }
    scanf("%d",&n);
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            fscanf(fp,"%d",&arcs[i][j]);
            path[i][j]=j; //初始化
        }
    }
    for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
    if(arcs[i][k]+arcs[k][j]<arcs[i][j])
    {
        arcs[i][j]=arcs[i][k]+arcs[k][j];
        path[i][j]=k;
    }
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            printf("%d->%d:%d    ",i,j,arcs[i][j]);
            int t=i;
            while(t!=j)
            {
                printf("%d--",t);
                t=path[t][j];
            }
            printf("%d",t);
            printf("\n");
        }

    }
    if(fclose(fp)!=0) printf(" close file error\n");
}
```

Dijkstra

```
int n,m,s,dis[maxn];
bool vis[maxn];

priority_queue<pa,vector<pa>,greater<pa> > q;

struct edge{
    int val,to;
};
vector<edge> e[maxn];

void dijkstra(){ //dis[i] 表示从起点到i的最短距离
    for(int i=1;i<=n;i++) dis[i]=1000000001;
    dis[s]=0;
    q.push(make_pair(0, s));
    while(!q.empty()) {
        int x=q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x]=1;
        for(int i=0;i<e[x].size();i++) {
            int y=e[x][i].to;
            if(dis[x]+e[x][i].val<dis[y]) {
                dis[y]=dis[x]+e[x][i].val;
                q.push(make_pair(dis[y], y));
            }
        }
    }
}

```

SPFA

```
int n,m,s,dis[maxn];
bool vis[maxn];

struct edge{
    int val,to;
};
vector<edge> e[maxn];
```

```

queue<int> q;

void SPFA(){
    for(int i=1;i<=n;i++) dis[i]=1000000001;
    dis[s]=0;
    q.push(s);
    vis[s]=1;
    while(!q.empty()) {
        int x=q.front();
        q.pop();
        for(int i=0;i<e[x].size();i++) {
            int y=e[x][i].to;
            if(dis[x]+e[x][i].val<dis[y]) {
                dis[y]=dis[x]+e[x][i].val;
                if(!vis[y]) {
                    q.push(y);
                    vis[y]=1;
                }
            }
        }
        vis[x]=0;
    }
}
```

缩点

tarjan

```
栈：当前dfs路径上的点
low[x]:x能到达的点中最小的dfn


dfs(x,t)
    将x入栈
    dfn[x]=t
    low[x]=t
    for(x,y)
        if(!vis[y])
            dfs(y,t+1),low[x]=min(low[x],low[y]);
        else
            if(y在栈中（在路径上））
                low[x]=min(low[x],low[y])
    if(low[x]=dfn[x])
        将栈出到x
```

网络流

最大流 / 最小割

#####

连通分量

二分图

```cpp
bool Hungary(int now)//now是当前顾客
{
    for(int i=0;i<edge[now].size();i++)//遍历当前顾客喜欢的每道菜
    {
        int to=edge[now][i];
        if(!vis[to])
        {
            vis[to]=1;
            if(!food[to]||Hungary(food[to]))//如果这道菜还没分配，或者可以重新分配
            {
                food[to]=now;//那这道菜就属于这个顾客
                return 1;//这个顾客有菜吃
            }
        }
    }
    return 0;//这个顾客没菜吃
}
//主程序
int ans=0;
for(int i=1;i<=n;i++)
{
    memset(vis,0,sizeof(vis));
    if(Hungary(i)) ans++;
}
```

P3386 二分图最大匹配

```cpp
#include<bits/stdc++.h>
using namespace std;
vector<int> par(2005);
bool vis[2005];
vector<vector<int> > G(2005);
int n, m, e;
void init(){
    cin >> n >> m >> e;
    for(int i=0; i<e; i++){
        int x, y;
        cin >> x >> y;
        if (x>=1&&y>=1&&x<=n&&y<=m)
            G[x].push_back(y);
        //G[y].push_back(x);
    }
}
bool dfs(int k){
    for(int i=0; i<G[k].size(); i++){
        int t=G[k][i];
        if(!vis[t]){
            vis[t]=1;
            if(!par[t] || dfs(par[t])){
                par[t]=k;
                //par[k]=t;
                return true;
            }
            }
        }
    return false;
}
int main(){
    init();
    int ans=0;
    for(int i=1; i<=n; i++){
        memset(vis, 0, sizeof(vis));
        if(dfs(i))
            ans++;
    }
    cout << ans << endl;
    return 0;
}
```

字符串

> 处理空格 用`stringstream`.

回文串

字典树

模板:

```cpp
class Trie {
private:
    bool isWord = false;
    Trie* next[26] = {nullptr};
public:
    /** Initialize your data structure here. */
    Trie() {

    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie* root = this;
        for(auto c : word){
            if(root->next[c-'a'] == nullptr) root->next[c-'a'] = new Trie();
            root = root->next[c-'a'];
        }
        root->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie* root = this;
        for(auto c : word){
            if(root->next[c-'a'] == nullptr) return false;
            root = root->next[c-'a'];
        }
        return root->isWord;
    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    bool startsWith(string prefix) {
        Trie* root = this;
        for(auto c : prefix){
            if(root->next[c-'a'] == nullptr) return false;
            root = root->next[c-'a'];
        }
        return true;
    }
};
```

KMP

```
void getNext(string s, vector<int> &next){
    int i=0, j=-1;
    next[0] = -1;
    while(i < s.size()){
        if(j == -1 || s[i] == s[j]){
            i++; j++;
            next[i] = j;
        }
        else j = next[j];
    }
}
int kmp(string s, string pattern){
    int i = 0, j = 0;
    vector<int> next(pattern.size());
    getNext(pattern, next);

    while(i < s.size() && j < (int)pattern.size()){
        if(j == -1 || s[i] == pattern[j]){
            i++; j++;
        }
        else j = next[j];
    }
    //匹配完成
    if(j >= pattern.size()) return i-j;
    else return -1;
}
```

DP:

```
const int maxn = 1e5+10;
int dp[maxn][256];
```

```cpp
class Solution {
public:
    void kmp(string pattern){
        int n = pattern.size();

        dp[0][pattern[0]] = 1;
        int X = 0;
        for(int i = 1; i < pattern.size(); i++){
            for(int j = 0; j < 256; j++)
                dp[i][j] = dp[X][j];
            dp[i][pattern[i]] = i + 1;
            X = dp[X][pattern[i]];
        }
    }
    int strStr(string haystack, string needle) {
        if(needle.size() == 0) return 0;
        memset(dp, 0, sizeof dp);
        int j = 0;
        kmp(needle);
        for(int i = 0; i < haystack.size(); i++){
            auto c = haystack[i];
            j = dp[j][c];
            if(j == needle.size()) return i - needle.size() + 1;
        }

        return -1;
    }
};
```

Manacher

```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Solution {
public:
    string longestPalindrome(string s) {
        // 特判
        int size = s.size();
        if (size < 2) {
            return s;
        }

        // 得到预处理字符串
        string str = "#";
        for (int i = 0; i < s.size(); ++i) {
            str += s[i];
            str += "#";
        }
        // 新字符串的长度
        int strSize = 2 * size + 1;
        // 数组 p 记录了扫描过的回文子串的信息
        vector<int> p(strSize, 0);

        // 双指针，它们是一一对应的，须同时更新
        int maxRight = 0;
        int center = 0;

        // 当前遍历的中心最大扩散步数，其值等于原始字符串的最长回文子串的长度
        int maxLen = 1;
        // 原始字符串的最长回文子串的起始位置，与 maxLen 必须同时更新
        int start = 0;

        for (int i = 0; i < strSize; ++i) {
            if (i < maxRight) {
                int mirror = (2 * center) - i;
                // 这一行代码是 Manacher 算法的关键所在，要结合图形来理解
                p[i] = min(maxRight - i, p[mirror]);
            }

            // 下一次尝试扩散的左右起点，能扩散的步数直接加到 p[i] 中
            int left = i - (1 + p[i]);
            int right = i + (1 + p[i]);

            // left >= 0 && right < sLen 保证不越界
            // str.charAt(left) == str.charAt(right) 表示可以扩散 1 次
            while (left >= 0 && right < strSize && str[left] == str[right]) {
                p[i]++;
                left--;
                right++;

            }

            // 根据 maxRight 的定义，它是遍历过的 i 的 i + p[i] 的最大者
            // 如果 maxRight 的值越大，进入上面 i < maxRight 的判断的可能性就越大，这样就可以重复利用之前判断过的回文信息了
            if (i + p[i] > maxRight) {
                // maxRight 和 center 需要同时更新
                maxRight = i + p[i];
                center = i;
            }
            if (p[i] > maxLen) {
                // 记录最长回文子串的长度和相应它在原始字符串中的起点
                maxLen = p[i];
                start = (i - maxLen) / 2;
            }
        }
        return s.substr(start, maxLen);
    }
};
```

数据结构

哈希

710. 黑名单中的随机数

```cpp
class Solution {
public:
    int sz;
    unordered_map<int, int> pos;
    Solution(int N, vector<int>& blacklist) {
        sz = N - blacklist.size();

        for(auto b : blacklist) pos[b] = 0x3f;

        int last = N-1;
        for(auto b : blacklist){
            if(b >= sz) continue;
            while(pos.count(last)) last--;
            pos[b] = last;
            last--;
        }
    }

    int pick() {
        int index = rand() % sz;
        if(pos.count(index)) return pos[index];

        return index;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(N, blacklist);
 * int param_1 = obj->pick();
 */
```

链表

双向链表

146. LRU缓存机制

```cpp
struct DeListNode{
    int key, value;
    DeListNode* prev;
    DeListNode* next;
    DeListNode(): key(0), value(0), prev(nullptr), next(nullptr) {}
    DeListNode(int _key, int _value): key(_key), value(_value), prev(nullptr), next(nullptr) {}
};

class LRUCache {
public:
    int capacity;
    int cnt;
    DeListNode *head;
    DeListNode *tail;
    unordered_map<int, DeListNode*> val;
    LRUCache(int capacity) {
        head = new DeListNode();
        tail = new DeListNode();
        head -> next = tail;
        tail -> prev = head;
        this->capacity = capacity;
        cnt = 0;
    }

    int get(int key) {
        if(val.count(key) > 0){
            DeListNode* deq = val[key];
            moveTohead(deq);
            return deq->value;
        }
        else return -1;
    }

    void put(int key, int value) {
        if(val.count(key) > 0) {
            DeListNode* deq = val[key];
            deq->value = value;
            moveTohead(deq);
        }
        else{
            DeListNode* deq = new DeListNode(key, value);
            val[key] = deq;
            addTohead(deq);
            cnt++;
            if(cnt > capacity){
                DeListNode *node = removeTail();
                val.erase(node->key);
                cnt--;
            }
        }
    }

    void addTohead(DeListNode *node){
        node -> prev = head;
        node -> next = head -> next;
        head -> next -> prev = node;
        head -> next = node;
    }
    void removeNode(DeListNode *node){
        node -> prev -> next = node -> next;
        node -> next -> prev = node -> prev;
    }
    void moveTohead(DeListNode *node){
        removeNode(node);
        addTohead(node);
    }
    DeListNode* removeTail(){
        DeListNode* node = tail -> prev;
        removeNode(node);
        return node;
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

图

堆

295. 数据流的中位数

```
class MedianFinder {
public:
    priority_queue<int> lo;
    priority_queue<int, vector<int>, greater<int> > hi;
    /** initialize your data structure here. */
    MedianFinder() {

    }

    void addNum(int num) {
        lo.push(num);
        hi.push(lo.top());
        lo.pop();


        if(lo.size() < hi.size()){
            lo.push(hi.top());
            hi.pop();
        }
    }

    double findMedian() {
        return (lo.size()+hi.size()) & 1 ? 1.0 * lo.top() : (lo.top() +
hi.top()) * 0.5;
    }
};
```

```

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */
```

树

Morris遍历

模板：

```cpp
TreeNode *getSuccessor(TreeNode *root){
        TreeNode* node = root->left;
        while(node->right && node->right!=root) node = node->right;
        return node;
    }
void morrisTraversal(TreeNode *root){
    TreeNode* node = root;
    while(node){
        if(node->left==nullptr){
            helper(); //其他处理
            node = node -> right;
        }
        else{
            TreeNode* succ = getSuccessor(node);
            if(succ->right == nullptr){
                succ->right = node;
                node = node->left;
            }
            else{
                succ->right = nullptr;
                helper();
                node = node->right;
            }
        }
    }
}
```

538. 把二叉搜索树转换为累加树

给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Tree)，使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

例如：

```
输入: 原始二叉搜索树:
              5
            /   \
           2     13

输出: 转换为累加树:
             18
            /   \
          20     13
```

```
/**
 * 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:
    int sum = 0;
    TreeNode* getSuccessor(TreeNode* node){
        TreeNode* succ = node->right;
        while(succ->left!=NULL && succ->left!=node) succ = succ->left;
        return succ;
    }
    TreeNode* convertBST(TreeNode* root) {
        TreeNode* node = root;

        while(node != NULL){
            if(node->right==NULL){
                sum += node->val; // 处理和
                node->val = sum;
                node=node->left;
            }
            else{
                TreeNode* succ = getSuccessor(node);
                if(succ->left == NULL){
                    succ->left = node;
                    node = node->right;
                }else{
                    succ->left = NULL;
                    sum += node->val;
                    node->val = sum;
                    node=node->left;
                }
            }
        }
        return root;
    }
};
```

501. 二叉搜索树中的众数

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

假定 BST 有如下定义：

- 结点左子树中所含结点的值小于等于当前结点的值
- 结点右子树中所含结点的值大于等于当前结点的值
- 左子树和右子树都是二叉搜索树

例如：
给定 BST `[1,null,2,2]`,

```
   1
    \
     2
    /
   2
```

`返回[2]`.

```cpp
/**
 * 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:
    vector<int> ans;
    int base, cnt=0, maxCnt=0;

    /*
    * 处理函数
    * 中序遍历后相等的数一定相邻
    */
    void helper(int x){
        // base 指当前保存的‘众数’
        if(x == base) cnt++;
        else{
            // 换‘众数’
            cnt = 1;
            base = x;
        }

        if(cnt == maxCnt) ans.emplace_back(x);
        if(cnt > maxCnt){
            maxCnt = cnt;
            ans = vector<int>{base};
        }
    }

    TreeNode *getSuccessor(TreeNode *root){
        TreeNode* node = root->left;
        while(node->right && node->right!=root) node = node->right;
        return node;
    }
    void morrisTraversal(TreeNode *root){
        TreeNode* node = root;
        while(node){
            //cout << node << endl;
            if(node->left==nullptr){
                helper(node->val);
                node = node -> right;
            }
            else{
                TreeNode* succ = getSuccessor(node);
                if(succ->right == nullptr){
                    succ->right = node;
                    node = node->left;
                }
                else{
                    succ->right = nullptr;
                    helper(node->val);
                    node = node->right;
                }
            }
        }
    }
    vector<int> findMode(TreeNode* root) {
        morrisTraversal(root);
        return ans;
    }
};
```

后续遍历

145. 二叉树的后序遍历

给定一个二叉树，返回它的 后序 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3
输出: [3,2,1]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

```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:
    void helper(vector<int> &ans, TreeNode* node){
        vector<int> tmp;
        while(node){
            tmp.emplace_back(node->val);
            node = node->right;
        }
        reverse(tmp.begin(), tmp.end());

        for(auto x:tmp){
            ans.emplace_back(x);
        }
    }
    TreeNode *getSuccessor(TreeNode *root){
        TreeNode* node = root->left;
        while(node->right && node->right!=root) node = node->right;
        return node;
    }
    vector<int> morrisTraversal(TreeNode *root){
        TreeNode* node = root;
        vector<int> ans;
        while(node){
            // if(node->left==nullptr){
            //     helper(); //其他处理
            //     node = node -> right;
            // }
            if(node->left != nullptr){
                TreeNode* succ = getSuccessor(node);
                if(succ->right == nullptr){
                    succ->right = node;
                    node = node->left;
                    continue;
                }
                else{
                    succ->right = nullptr;
                    helper(ans, node->left);
                }
            }
            node = node->right;
        }
        helper(ans, root);
        return ans;
    }
    vector<int> postorderTraversal(TreeNode* root) {
        return morrisTraversal(root);
    }
};
```

构造二叉树

[105. 从前序与中序遍历序列构造二叉树]

例:

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

二叉树:

```
    3
   / \
  9  20
    /  \
   15   7
```

```
/**
 * 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:
    vector<int> pre, in;
    map<int, int> pos;
    TreeNode* build(int pre_l, int pre_r, int in_l, int in_r){
        if(pre_l > pre_r) return nullptr;

        int pre_val = pre[pre_l];
        int p = pos[pre_val];

        int num = p-in_l;

        TreeNode* root = new TreeNode(pre_val);

        root-> left = build(pre_l+1, pre_l+num, in_l, p-1);
        root -> right = build(pre_l+num+1, pre_r, p+1, in_r);
        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        pre = preorder; in = inorder;
        for(int i = 0; i < inorder.size(); i++) pos[inorder[i]] = i;
        return build(0, preorder.size()-1, 0, inorder.size()-1);
    }
};
```

106. 从中序与后序遍历序列构造二叉树

例:

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

二叉树:

```
    3
   / \
  9  20
    /  \
   15   7
```

```
/**
 * 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:
    vector<int> in, post;
    map<int, int> pos;
    int rt;
    TreeNode* build(int l, int r){
        if(l > r) return nullptr;

        int post_val = post[rt];
        int p = pos[post_val];

        TreeNode* root = new TreeNode(post_val);
        rt--;
        root -> right = build(p+1, r);
        root-> left = build(l, p-1);

        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        in = inorder; post = postorder;
        rt = inorder.size()-1;
        for(int i = 0; i < inorder.size(); i++) pos[inorder[i]] = i;
        return build(0, rt);
    }
};
```

889. 根据前序和后序遍历构造二叉树

返回与给定的前序和后序遍历匹配的`任何二叉树`。

例：

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

二叉树：

```
      1
     /  \
    2    3
   / \  / \
  4   5 6   7
```

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode constructFromPrePost(int[] pre, int[] post) {
        int N = pre.length;
        if (N == 0) return null;
        TreeNode root = new TreeNode(pre[0]);
        if (N == 1) return root;

        int L = 0;
        for (int i = 0; i < N; ++i)
            if (post[i] == pre[1])
                L = i+1;

        root.left = constructFromPrePost(Arrays.copyOfRange(pre, 1, L+1),
                                         Arrays.copyOfRange(post, 0, L));
        root.right = constructFromPrePost(Arrays.copyOfRange(pre, L+1, N),
                                          Arrays.copyOfRange(post, L, N-1));
        return root;
    }
}
```

序列化

449. 序列化和反序列化二叉搜索树

297. 二叉树的序列化与反序列化

```java
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(root == nullptr) return "#";

        return to_string(root->val) + " " + serialize(root->left) + " " +
serialize(root->right);
    }

    TreeNode* build(stringstream &data){
        string s;
        data >> s;
        if(s == "#") return nullptr;

        TreeNode* root = new TreeNode(stoi(s));
        root -> left = build(data);
        root -> right = build(data);

        return root;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        stringstream s(data);
        return build(s);
    }
};

// Your Codec object will be instantiated and called as such:
// Codec* ser = new Codec();
// Codec* deser = new Codec();
// string tree = ser->serialize(root);
// TreeNode* ans = deser->deserialize(tree);
// return ans;
```

遍历

116. 填充每个节点的下一个右侧节点指针

```
/*
// Definition for a Node.
class Node {
public:
    int val;
```

```cpp
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};
*/

class Solution {
public:
    Node* connect(Node* root) {
        if(root == nullptr) return nullptr;

        connect(root->left, root->right);
        return root;
    }
    void connect(Node* node1, Node* node2){
        if(!node1 || !node2) return;

        node1->next = node2;
        connect(node1->left, node1->right);
        connect(node2->left, node2->right);

        connect(node1->right, node2->left);
    }
};
```

114. 二叉树展开为链表

```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:
    void flatten(TreeNode* root) {
        if(root == nullptr) return;

        flatten(root->left);
        flatten(root->right);

        TreeNode* left = root->left, *right = root->right;

        root->left = nullptr;
        root->right = left;

        TreeNode* p = root;
        while(p->right != nullptr) p = p->right;

        p->right = right;
    }
};
```

二叉搜索树

98. 验证二叉搜索树

```cpp
/**
 * 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:
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, nullptr, nullptr);
    }
    bool isValidBST(TreeNode *root, TreeNode* min, TreeNode* max){
        if(!root) return true;
        if(min && root->val <= min->val) return false;
        if(max && root->val >= max->val) return false;

        return isValidBST(root->left, min, root) && isValidBST(root->right, root, max);
    }
};
```

701. 二叉搜索树中的插入操作

```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* insertIntoBST(TreeNode* root, int val) {
        if(!root) return new TreeNode(val);

        if(root->val > val) root->left = insertIntoBST(root->left, val);
        if(root->val < val) root->right = insertIntoBST(root->right, val);

        return root;
    }
};
```

450. 删除二叉搜索树中的节点

```
/**
 * 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:
    int predecessor(TreeNode *root){
        root = root->left;
        while(root->right) root=root->right;
        return root->val;
    }
    int successor(TreeNode *root){
        root=root->right;
        while(root->left) root=root->left;
        return root->val;
    }
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(root==NULL) return NULL;

        if(root->val < key){
            root->right = deleteNode(root->right, key);
        }else if(root->val > key){
            root->left = deleteNode(root->left, key);
        }else{
            if(root->left==NULL && root->right==NULL) root = NULL;
            else if(root->right){
                root->val=successor(root);
                root->right=deleteNode(root->right, root->val);
            }else{
                root->val=predecessor(root);
```

```
                    root->left=deleteNode(root->left, root->val);
                }
            }
            return root;
        }
};
```

LCA

```
/**
 * 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* ans;
    bool dfs(TreeNode* root, TreeNode* p, TreeNode* q){
        if(root == NULL) return false;
        bool l = dfs(root->left, p, q);
        bool r = dfs(root->right, p, q);
        if((l&&r) || (( l || r ) && (root->val == q->val || root->val == p->val)))
            ans = root;
        return (l||r) || (root->val == p->val || root->val == q->val);
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        dfs(root, p, q);
        return ans;
    }
};

//2
class Solution {
public:
    unordered_map<int, TreeNode*> fa;
    unordered_map<int, bool> vis;
    void dfs(TreeNode* root){
        if (root->left != nullptr) {
            fa[root->left->val] = root;
            dfs(root->left);
        }
        if (root->right != nullptr) {
            fa[root->right->val] = root;
            dfs(root->right);
        }
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
        fa[root->val] = nullptr;
        dfs(root);
        while (p != nullptr) {
            vis[p->val] = true;
            p = fa[p->val];
        }
        while (q != nullptr) {
            if (vis[q->val]) return q;
            q = fa[q->val];
        }
        return nullptr;
    }
};
```

栈

单调栈

解决：下一个最大值

316. 去除重复字母

```
class Solution {
public:
    string removeDuplicateLetters(string s) {
        if(s.size() == 0) return "";
        unordered_map<char, int> p;
        for(int i = 0; i < s.size(); i++) p[s[i]] = i;

        stack<char> st;
        set<char> vis;
        for(int i = 0; i < s.size(); i++){
            if(vis.count(s[i])) continue;
            while(!st.empty() && st.top() > s[i]){
                if(p[st.top()] <= i) break;
                vis.erase(st.top());
                st.pop();
            }
            st.push(s[i]);
            vis.insert(s[i]);
        }
        string ans;
        while(!st.empty()){
            ans += st.top();
            st.pop();
        }
        reverse(ans.begin(), ans.end());
```

```cpp
        return ans;
    }
};
```

503. 下一个更大元素 II

```cpp
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {
        vector<int> num(nums);
        for(auto x : nums) num.emplace_back(x);

        vector<int> ans(nums.size());

        stack<int> st;
        for(int i = num.size()-1; i >= 0; i --){
            while(!st.empty() && st.top() <= num[i]) st.pop();
            if(i < nums.size()) ans[i] = st.empty() ? -1 : st.top();
            st.push(num[i]);
        }

        return ans;
    }
};
```

队列

单调队列

滑动串口

239. 滑动窗口最大值

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

        }
        return res;
    }
};
```

优先队列

```
#define P pair<int, int>
struct cmp{
    bool operator()(const P p1, const P p2) {
        return p1.second > p2.second; //second的小值优先
    }
};

priority_queue<P, vector<P>, cmp> que;

priority_queue<int> lo;                                    // max heap
priority_queue<int, vector<int>, greater<int>> hi;    // min heap
```

460. LFU缓存

```
struct Node{
    int key, val, freq;
    Node(int _key, int _val, int _freq):key(_key), val(_val), freq(_freq){}
};
class LFUCache {
    int minfreq, capacity;
    unordered_map<int, list<Node>::iterator> key_table;
    unordered_map<int, list<Node>> freq_table;
public:
    LFUCache(int _capacity) {
        minfreq = 0;
        capacity = _capacity;
        key_table.clear();
        freq_table.clear();
    }

    int get(int key) {
        if(capacity == 0) return -1;
        auto it = key_table.find(key);
        if(it == key_table.end()) return -1;
        list<Node>::iterator node = it->second;
        int val = node->val, freq = node->freq;
```

```cpp
        freq_table[freq].erase(node);
        if(freq_table[freq].size() == 0){
            freq_table.erase(freq);
            if(minfreq == freq) minfreq += 1;
        }

        freq_table[freq+1].push_front(Node(key, val, freq+1));
        key_table[key] = freq_table[freq+1].begin();
        return val;
    }

    void put(int key, int value) {
        if(capacity == 0) return;
        auto it = key_table.find(key);
        if(it == key_table.end()){
            if(key_table.size() == capacity){
                auto it2 = freq_table[minfreq].back();
                key_table.erase(it2.key);
                freq_table[minfreq].pop_back();
                if(freq_table[minfreq].size() == 0){
                    freq_table.erase(minfreq);
                }
            }
            freq_table[1].push_front(Node(key, value, 1));
            key_table[key] = freq_table[1].begin();
            minfreq = 1;
        }
        else{
            list<Node>::iterator node = it->second;
            int freq = node->freq;
            freq_table[freq].erase(node);
            if(freq_table[freq].size() == 0){
                freq_table.erase(freq);
                if(minfreq == freq) minfreq += 1;
            }
            freq_table[freq+1].push_front(Node(key, value, freq+1));
            key_table[key] = freq_table[freq+1].begin();
        }
    }
};

/**
 * Your LFUCache object will be instantiated and called as such:
 * LFUCache* obj = new LFUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

并查集

模板

```cpp
int fa[N];
  void init(int n) {  // 不要忘记哦！！！
      for (int i = 0; i <= n; i++)
          fa[i] = i;
      }
  }
  void unin(int u, int v) {
      int fau = find(u);
      int fav = find(v);
      if (fau == fav) return;
      fa[fav] = fau;
  }
  int find(int u) {
       return fa[u] == u ? fa[u] : fa[u] = find(fa[u]);
  }
```

990. 等式方程的可满足性

```cpp
class Solution {
public:
    int root[600];
    void init(){
        for(int i = 0; i < 600; i++) root[i] = i;
    }
    int find(int x){
        return x == root[x] ? x : root[x] = find(root[x]);
    }
    void unite(int x, int y){
        x = find(x); y = find(y);
        root[x] = y;
    }
    bool equationsPossible(vector<string>& equations) {
        init();
        for (const string& str: equations) {
            if (str[1] == '=') {
                int index1 = str[0] - 'a';
                int index2 = str[3] - 'a';
                unite(index1, index2);
            }
        }
        for (const string& str: equations) {
            if (str[1] == '!') {
                int index1 = str[0] - 'a';
                int index2 = str[3] - 'a';
                if (find(index1) == find(index2)) {
                    return false;
                }
            }
        }
        return true;
    }
};
```

树状数组

```
int lowbit(int x)
{
    return x & (-x);
}
void modify(int x,int add)//一维
{
    while(x<=MAXN)
    {
        a[x]+=add;
        x+=lowbit(x);
    }
}
int get_sum(int x)
{
    int ret=0;
    while(x!=0)
    {
        ret+=a[x];
        x-=lowbit(x);
    }
    return ret;
}
void modify(int x,int y,int data)//二维
{
    for(int i=x;i<MAXN;i+=lowbit(i))
        for(int j=y;j<MAXN;j+=lowbit(j))
            a[i][j]+=data;
}
int get_sum(int x,int y)
{
    int res=0;
    for(int i=x;i>0;i-=lowbit(i))
        for(int j=y;j>0;j-=lowbit(j))
            res+=a[i][j];
    return res;
}
```

字符串

数组

文件

```
freopen("in.txt", "r", stdin);
freopen("out.txt", "w", stdout);
fclose(stdin);
fclose(stdout);
```

STL

priority_queue

```
#define P pair<int, int>
struct cmp{
    bool operator()(const P p1, const P p2) {
        return p1.second > p2.second; //second的小值优先
    }
};

priority_queue<P, vector<P>, cmp> que;

priority_queue<int> lo;                              // max heap
priority_queue<int, vector<int>, greater<int>> hi;   // min heap
```

vector

```
for(auto it = nestedList.begin(); it != nestedList.end(); it++) handle(*it);
store.insert(lower_bound(store.begin(), store.end(), num), num);
```