- int[]转ArrayList<Intrger>

```
    List<Integer> output = new ArrayList<Integer>();
    for (int num : nums) {
        output.add(num);
    }
```

- Note：判断容器和数组为空使用list != null && list.size() != 0
- 当边界条件可能出现大于Integer.MAX_VALUE或小于Integer.MIN_VALUE时，可将发生溢出的语句进行移项防止溢出，如：

```
if (res * 10 + x % 10 > Integer.MAX_VALUE )
//转化为
if (res > (Integer.MAX_VALUE - x % 10)/10)
```

数组

二分查找

- 仅适用于有序表
- 注意取等号并

```
    public int searchInsert(int[] nums, int target) {
        int high = nums.length - 1, low = 0, mid;
        //方法1，左闭右闭
        while (low <= high){
            mid = low + (high - low) / 2;//为偶数时向下取整
            if (nums[mid] == target){
                return mid;
            } else if(nums[mid] > target){
                high = mid - 1;
            } else{
                low = mid + 1;
            }
        }
        if (求左边界)   return high;
        if (求右边界)   return low;

        //方法2，左闭右开
        while (low < high){
            mid = low + (high - low) / 2;//为偶数时向下取整
            if (nums[mid] == target){
                return mid;
            } else if(nums[mid] > target){
                high = mid;
            } else{
                low = mid + 1;
            }
        }
        //方法3，左开右闭
        while (low < high){
            mid = low + (high - low + 1) / 2;//为偶数时向下取整
            if (nums[mid] == target){
                return mid;
            } else if(nums[mid] > target){
                high = mid + 1;
            } else{
                low = mid;
            }
        }
    }
```

35.搜索插入位置

34.在排序数组中查找元素的第一个和最后一个位置

69.x 的平方根

367.有效的完全平方数

快慢指针：进行遍历

- 可对数组进行原地操作
- 使用应注意快指针左边是什么，慢指针的左边又是什么

```
public int removeDuplicates(int[] nums) {
        int slowIndex = 0;
        for (int fastIndex = 0; fastIndex < nums.length; fastIndex++){
            if (慢指针移动条件){
                nums[slowIndex++] = nums[fastIndex];
            }
        }
        return slowIndex;
    }
```

27. 移除元素

双指针法：两边向中间移动

Note：求左边界left右移条件和==条件合并，求右边界right左移条件和==条件合并

```
    public int[] sortedSquares(int[] nums) {
        int right = nums.length - 1;
        int left = 0;
        while (left <= right) {
            if (left右移条件) {
                left右移并对相关数据进行处理
            } else if (right左移条件) {
                right左移并对相关数据进行处理
            } else{
                其他情况处理
            }
        }
        return result;
    }
```

26.删除排序数组中的重复项

283.移动零

844.比较含退格的字符串

977.有序数组的平方

排序+双指针

作用1：去重遍历

作用2：降低时间复杂度(O(n^3)->O(n^2))

Note：排序是去除重复的有效操作（也可与回溯结合）

```
public int[] sortedSquares(int[] nums) {
    Arrays.sort(nums);
    int right = nums.length - 1;
    int left = 0;
    while (left <= right) {
        if (left右移条件) {
            left++;
        } else if (right左移条件) {
            right--;
        } else{
            //进行去重
            while(right > left && nums[right] == nums[right - 1])
                right--;
            while(right > left && nums[left] == nums[left + 1])
                left++;
            right--;
            left++;
        }
    }
    return result;
}
```

滑动窗口：求满足条件的最长（短）连续数组

- 求二者较小者用Math.min，一般使用左闭右开
- 数组中有负数时一般用不了滑动窗口

```
Set<Character> set = new HashSet<>();
int low = 0, maxLen = 0;
for (int i = 0; i < s.length(); i++){
    while (set.contains(s.charAt(i))){
        set.remove(s.charAt(low));
        low++;
    }
    set.add(s.charAt(i));
    maxLen = Math.max(set.size(), maxLen);
}
return maxLen;
```

904.水果成篮

76.最小覆盖子串

209.长度最小的子数组

前缀和：求解连续数组问题

prefixSum[b~c] = predixSum[a~c] - prefixSum[a~b]

Note：数组题的常见思路就是前缀和、滑动窗口和dp

模拟：螺旋矩阵

54.螺旋矩阵

链表

设置虚拟头结点：需要对实际头结点进行删除或插入操作

```
ListNode dummy = new ListNode();
dummy.next = head;
```

双指针法：前链的尾结点、后链的头结点进行暂存，操作链的节点进行操作和移动

```
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode cur = head;
        ListNode temp = null;
        while (cur != null) {
            temp = cur.next;// 保存后链的头结点，防止断链
            cur.next = prev;
            prev = cur;
            cur = temp;
        }
        return prev;
    }
```

- 206.反转链表

递归：移动到链表的尽头，通过递归的弹出完成当前节点的前移

```
///24. 两两交换链表中的节点
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null){//结束条件
            return head;
        }
        ListNode thirdNode = swapPairs(head.next.next);//步长为2，传入点数为2个
        ListNode secondNode = head.next;//对传入的点进行拷贝
        secondNode.next = head;
        head.next = thirdNode;//进行连接与断开
        return secondNode;
    }
```

```
///206. 反转链表
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null){
            return head;
        }

        ListNode start = reverseList(head.next);//步长为1，传入点数为1个
        ListNode secondNode = head.next;//对传入的点进行拷贝
        secondNode.next = head;
        head.next = null;//进行连接与断开
        return start;
    }
```

树

先(中、后)序遍历

```
    public void preOrder(TreeNode root){
        if (root == null){
            return;
        }
        preRes.add(root.val);
        preOrder(root.left);
        preOrder(root.right);
    }

    public void inOrder(TreeNode root){
        if (root == null){
            return;
        }
        inOrder(root.left);
        inRes.add(root.val);
        inOrder(root.right);
    }

    public void postrOrder(TreeNode root){
        if (root == null){
            return;
        }
        postrOrder(root.left);
        postrOrder(root.right);
        postRes.add(root.val);
    }
```

层序遍历

```
    public List<List<Integer>> levelOrder(TreeNode root) {
        Deque<TreeNode> deque = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        List<Integer> floor = new ArrayList<>();

        if (root == null)
            return res;
        deque.offer(root);
        while(!deque.isEmpty()){
           int len = deque.size();

           while (len > 0){
                TreeNode node = deque.poll();
                floor.add(node.val);
                if (node.left != null)
                    deque.offer(node.left);
                if (node.right !=  null)
                    deque.offer(node.right);
                len--;
           }

            res.add(new ArrayList<Integer>(floor));
            floor.clear();
        }

        return res;
    }
```

贪心

分配问题

455、分饼干

- 排序
- 最优分配

135、分糖果

- 两次遍历分配

122、股票交易2

- 寻找极值

区间问题

435、不重叠区间

- 排序
- 遍历进行区间移除

452、最少箭

- 排序
- 遍历查看是否溢出区间

406、身高创建队列

- 排序
- 遍历并寻找插入位置

605、存放花

- 通过寻找区间的开始与结束划分区间

763、划分字母区间

- 统计信息（最后出现）
- 通过寻找区间的开始与结束划分区间

回溯

- 全排列

```
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new LinkedList<>();
        List<Integer> path = new LinkedList<>();
        boolean[] used = new boolean[nums.length];

        backtracking(nums.length, used, nums, res, path);

        return res;
    }

    public void backtracking(int len, boolean[] used, int[] nums,
List<List<Integer>> res, List<Integer> path){
        if (path.size() == len){
            res.add(new LinkedList<Integer>(path));
            return;
        }

        for (int i = 0; i < len; i++){
            if (!used[i]){
                path.add(nums[i]);
                used[i] = true;
                backtracking(nums.length, used, nums, res, path);
                path.remove(path.size()-1);
                used[i] = false;
            }
        }
    }
```

```
List<List<Integer>> res = new ArrayList<>();
List<Integer> path = new ArrayList<>();
public List<List<Integer>> combine(int inputLen, int outputLen) {
    backtracking(inputLen,outputLen,0);
    return res;
}
```

```
public void backtracking(int inputLen, int outputLen, int startIndex){
    if (path.size() == outputLen){
        res.add(new ArrayList<>(path));
        return;
    }
    for (int i=startIndex;i<=inputLen;i++){
        path.add(i);
        backtracking(inputLen,outputLen,i);
        path.remove(path.size()-1);
    }
}
```

动态规划

1. 建立动态规划数组
2. 动态规划数组初始化
3. 确定递推公式
4. 确定遍历顺序
5. 判断选择有用数据

0-1背包问题（物品数量为1个，求满足条件的物品组合种类的相关结果）

```
    public static void testWeightBagProblem(int[] weight, int[] value, int bagWeight){
        int wLen = weight.length;
        //定义dp数组：dp[j]表示背包容量为j时，能获得的最大价值
        int[] dp = new int[bagWeight + 1];
        //非必要，进行dp数组的初始化
        dp[0] = 1;
        //遍历顺序：先遍历物品，再遍历背包容量
        for (int i = 0; i < wLen; i++){
            for (int j = bagWeight; j >= weight[i]; j--){
                dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
            }
        }
        //打印dp数组
        for (int j = 0; j <= bagWeight; j++){
            System.out.print(dp[j] + " ");
        }
    }
```

完全背包问题（物品数量为无限个，求满足条件的物品组合种类的相关结果）

先遍历物品，再遍历背包(该方法不区分21和12)

```java
private static void testCompletePack(){
    int[] weight = {1, 3, 4};
    int[] value = {15, 20, 30};
    int bagWeight = 4;
    int[] dp = new int[bagWeight + 1];
    for (int i = 0; i < weight.length; i++){ // 遍历物品
        for (int j = weight[i]; j <= bagWeight; j++){ // 遍历背包容量
            dp[j] = Math.max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    for (int maxValue : dp){
        System.out.println(maxValue + "   ");
    }
}
```

先遍历背包，再遍历物品(该方法区分21和12)

```java
private static void testCompletePackAnotherWay(){
    int[] weight = {1, 3, 4};
    int[] value = {15, 20, 30};
    int bagWeight = 4;
    int[] dp = new int[bagWeight + 1];
    for (int i = 1; i <= bagWeight; i++){ // 遍历背包容量
        for (int j = 0; j < weight.length; j++){ // 遍历物品
            if (i - weight[j] >= 0){
                dp[i] = Math.max(dp[i], dp[i - weight[j]] + value[j]);
            }
        }
    }
    for (int maxValue : dp){
        System.out.println(maxValue + "   ");
    }
}
```

环形树形动态规划（打家劫舍问题）

带状态的动态规划（股票问题）

股票问题的i为天数，j为该天数的状态（买入，卖出，今日卖出，冷冻期等）

子序列问题

诸如求公共子序列、子数组或回文的最长长度问题，使用二维dp（如果只输入了一个变量，那它既是i又是j）

单调栈

```
Deque<Integer> queue = new LinkedList<>();
int[] res = new int[temperatures.length];

for (int i = 0; i < temperatures.length; i++){
    while (!queue.isEmpty() && temperatures[i] > temperatures[queue.peek()]){
        res[queue.peek()] = i - queue.peek();
        queue.pop();
    }
    queue.push(i);
}
```

深度优先遍历

- 递归方法

```
public void DFSSearch(int[][] grid) {
        for (int x=0;x<grid.length;x++)
        {
            for (int y=0;y<grid[0].length;y++)
            {
                DFS(grid,x,y);
            }
        }
    }

public static int[] xAdj = {1,0,-1,0};
public static int[] yAdj = {0,1,0,-1};

public static void DFS(int[][] grid,int x, int y)
{
    if(x==-1||y==-1||x==grid.length||y==grid[0].length||grid[x][y] == 0)
    {
        return;
    }
    else
    {
        grid[x][y] = 0;
        for(int i=0;i<4;i++)
        {
            DFS(grid,x+xAdj[i],y+yAdj[i]);
        }
    }
}
```

- 栈方法

```
public void DFSSearch(int[][] grid) {
    for (int x=0;x<grid.length;x++)
    {
        for (int y=0;y<grid[0].length;y++)
        {
                DFS(grid,x,y);
        }
    }
}
```

```
public static int[] xAdj = {1,0,-1,0};
public static int[] yAdj = {0,1,0,-1};

public static void DFS(int[][] grid,int x, int y)
{
    if (grid[x][y] == 1)
    {
        return ;
    }
    int[] cur ={x,y},next={x,y};
    Deque<int[]> stack = new LinkedList<>();
    stack.push(cur);
    while(!stack.isEmpty())
    {
        cur = stack.pop();
        if(cur[0]>=0||cur[1]>=0||cur[0]<=grid.length-1||cur[1]<=grid[0].length-1||grid[cur[0]][cur[1]] == 1)
        {
            grid[cur[0]][cur[1]] = 0;
            for(int i=0;i<4;i++)
            {
                next[0] = cur[0]+xAdj[i];
                next[1] = cur[1]+yAdj[i];
                stack.push(new int[]{next[0], next[1]});
            }
        }
    }
}
```

广度优先遍历

```
public void DFSSearch(int[][] grid) {
    for (int x=0;x<grid.length;x++)
    {
        for (int y=0;y<grid[0].length;y++)
        {
            if (grid[x][y] == 1)
            {
                BFS(grid,x,y);
            }
        }
    }
}

public static int[] xAdj = {1,0,-1,0};
public static int[] yAdj = {0,1,0,-1};

public static void BFS(int[][] grid,int x, int y)
{
    int[] cur ={x,y},next={x,y};
    Queue<int[]> queue = new LinkedList<>();
    queue.offer(cur);
    while(!queue.isEmpty())
    {
        cur = queue.poll();
       if(cur[0]>=0||cur[1]>=0||cur[0]<=grid.length-1||cur[1]<=grid[0].length-1||grid[cur[0]][cur[1]] == 1)
        {
            grid[cur[0]][cur[1]] = 0;
            for(int i = 0; i < 4; i++)
            {
                next[0] = cur[0]+xAdj[i];
                next[1] = cur[1]+yAdj[i];
                queue.offer(new int[]{next[0], next[1]});
            }
        }
    }
}
```

搜索最短路径一般使用广搜，广搜只要搜到了终点，那么一定是最短的路径

位运算

1、将数字二进制的最后一位1变为0

```
n = n &(n-1);
```

2、字符串中出现奇数次的字符

```
int ret = 0;
for (char ch: s) {
    ret ^= ch;
}
```

字符串

KMP算法

```
public int strStr(String haystack, String needle) {
        int[] next = getNext(needle);
        int j = 0;
        for (int i = 0; i < haystack.length(); i++){
            while(j > 0 && haystack.charAt(i) != needle.charAt(j)){
                j = next[j-1];
            }
            if (haystack.charAt(i) == needle.charAt(j)){
                j++;
                if (j == needle.length()){
                    return i - needle.length() + 1;
                }
            }
        }

        return -1;
    }
    //匹配规则i=0时-》next[0]=0;i>0时-》next[i] = max(n|needle[0:n] == needle[len+n-1:len-1])
    public int[] getNext(String needle){
        int[] next = new int[needle.length()];
        int j = 0;//j指的是匹配字符串的前缀末尾索引
        for (int i = 1; i < needle.length(); i++){//i指的是匹配字符串的后缀末尾索引
            while(j > 0 && needle.charAt(i) != needle.charAt(j)){//当前缀末尾与后缀末尾不匹配时，前缀末尾一直向前寻找上一个前缀末尾
                j = next[j-1];
            }
            if (needle.charAt(i) == needle.charAt(j)){//当前缀末尾与后缀末尾匹配时，前缀末尾向后移动
                j++;
            }
            next[i] = j;
        }

        return next;
    }
```

并查集

求解两个变量是否属于一个集合

```
    public class UnionFind{
        public int[] father;

        public UnionFind(int n){
            father = new int[n];
            for (int i = 0; i < n; i++){
                father[i] = i;
            }
        }

        public int find(int x){
            if (x == father[x]){
                return x;
            }
            father[x] = find(father[x]);
            return father[x];
        }

        public void union(int x, int y){
            int xFather = find(x), yFather = find(y);
            if (xFather == yFather){
                return ;
            }
            father[xFather] = yFather;
        }

        public boolean isConnect(int x, int y){
            return find(x) == find(y);
        }
    }
```

字典树

```java
class Trie {
    public class Node{
        boolean isEnd = false;
        Node[] next = new Node[26];
    }

    public Node root;
    public Trie() {
        root = new Node();
    }

    public void insert(String word) {
        Node cur = root;
        for (int i = 0; i < word.length(); i++){
            int index = word.charAt(i)-'a';
            if (cur.next[index] == null){
                cur.next[index] = new Node();
            }
            cur = cur.next[index];
        }
        cur.isEnd = true;
    }

    public boolean search(String word) {
        Node cur = root;
        for (int i = 0; i < word.length(); i++){
            int index = word.charAt(i)-'a';
            if (cur.next[index] != null){
                cur = cur.next[index];
            }
            else{
                return false;
            }
        }
        return cur.isEnd;
    }

    public boolean startsWith(String prefix) {
        Node cur = root;
        for (int i = 0; i < prefix.length(); i++){
            int index = prefix.charAt(i)-'a';
            if (cur.next[index] != null){
                cur = cur.next[index];
            }
            else{
                return false;
            }
        }
        return true;
    }
}
```

其他

快速幂

```
   public static double FastPow(int x,int y){
        double res= 1;
        while (y!=0){
            if(y%2 == 1){ //指数为奇数，也可以利用位运算：(y&1)==1 （与操作）： 判断 n 二进制最右一位是否为 1
                res *= x;
            }
            y=y/2;  //指数循环二分,y>>=1 （移位操作）： n 右移一位（可理解为删除最后一位,即除以2）。
            x=x*x;  //底数平分

        }
        return res;
    }
```