

## 指针和引用

Java 引用传递 `ArrayList<Integer> b = a` 可以修改自定义类型的值。Java 无地址传递，在递归遍历树的同时想修改变量，声明为全局变量可降低难度。

## 数组 int []

初始化: `int[] v = new int[];`  
`boolean[][] vv = new boolean[m][n];`  
判空: `v == null || v.length == 0` // 一维  
`vv == null || vv.length == null || vv[0].length == 0` // 二维  
截取: `subarr = Arrays.copyOfRange(arr,2,6);`

## 字符串 String

初始化: `String str = "hello world";`  
`StringBuilder sb = new StringBuilder(str);`  
类型转换: `sb.toString();`  
增: `sb.append(true); sb.insert(i, "abc");`  
删: `sb.deleteCharAt(i); sb.delete(i,j);`  
改: `sb.setCharAt(i,'z');`  
查: `char c = str.charAt(i);`  
判等: `s1.equals(s2);` 判空: `s1.isEmpty();`  
截取: `str.substring(i); str.substring(i,j);`  
拼接: `str.concat("abc");`

## 动态数组

`ArrayList<Integer> v = new ArrayList<>();`  
`isEmpty(), size()`  
`add(), remove(), get(), set(1, 100)`

## 双向链表

`LinkedList<Integer> v = new LinkedList<>();`  
`isEmpty(), size(), contains()`  
`add(), remove(), get()` // 默认尾插, 头删  
`addFirst(), removeFirst(), removeLast()`

## 哈希表 (哈希集合类似)

`Map<Integer, String> map = new HashMap<>();`  
`containsKey(), keySet(), getOrDefault()`  
`put(), remove(), get()`

## 有序表 (有序集合类似)

`Map<String> map = new TreeSet<>();`  
`put(), remove(), containsKey(), firstKey()`  
`lastKey(), floorKey(), ceilingKey()`

## 双端队列 (代替栈和队列)

`Deque<String> stack = new ArrayDeque<>();`  
`Deque<String> queue = new LinkedDeque<>();`  
`isEmpty(), size()`  
`addFirst(), removeFirst(), peekFirst()`  
`addLast(), removeLast(), peekLast()`

## 类型转换

`Integer.parseInt(s)` // String -> int  
`String.valueOf(chs)` // int, char[] -> String  
'8' - '0' // char -> int  
`Double.valueOf(i)` // int -> double  
`foo.intValue()` // double -> int  
`list = Arrays.asList(arr)` // [] -> ArrayList

## 哨兵: 哑元结点 (提高效率)

`TreeNode p = new TreeNode(-1, head);`  
`TreeNode dummy = p;` // 保存头结点位置, 不移动  
`return dummy.next;`

## Arrays 和 Collections 工具包

`Arrays.sort(nums);` // 数组排序  
`Arrays.binarySearch(nums, 23);`  
`Arrays.stream(nums).max().getAsInt();`  
`Collections.sort(list);` // 列表排序  
`list.sort(Collections.reverseOrder());` // 逆序  
`Collections.reverse(list);` // 翻转链表

## 大数计算: java.math.\*

默认 int 无符号最大值约为  $40 \times 10^8$  (32 位)。  
`BigInteger A = BigInteger.valueOf(23);`  
`BigDecimal B = BigDecimal.valueOf(1234.56);`  
`A.add(A), A.subtract(A),`  
`A.multiply(A), A.divide(A)`

## 简易哈希表的实现

基于数组实现: key 为数组下标, value 为元素值。  
`int[] map = new int[256];` // ASCII -> 下标

## ArrayList 元素去重

`ArrayList<String> list = new ArrayList<>();`  
`Set<String> set = new HashSet<>();`  
`set.addAll(list);`  
`ArrayList<String> ret = new ArrayList<>(set);`

## 取出整数中的每一位

```
while (n > 0) {
    digit = n % 10; // 取出各位
    n = n / 10; // 更新数字
} // 注意处理最值 Integer.MAX_VALUE
```

## 遍历哈希表或哈希集合

`Map<String, String> map = new HashMap<>();`  
`for (Map.Entry<String, String> entry :`  
`map.entrySet()) {`  
`int key = entry.getKey();`  
`int value = entry.getValue();`  
`}`

## 链表的遍历

```
void traverse(ListNode head) {
    // 前序遍历代码
    traverse(head.next);
    // 后序遍历代码
}
```

## 二叉树的遍历

```
void traverse(TreeNode root) {
    if (root == null) return;
    // 前序遍历代码
    traverse(root.left);
    // 中序遍历代码
    traverse(root.right);
    // 后序遍历代码
}
```

## 图 (N 叉树) 的遍历

```
class TreeNode {
    int val;
    TreeNode[] children;
}

void traverse(TreeNode root) {
    for (TreeNode child : children)
        traverse(child);
}
```

## 求二叉树的深度: 递归实现

```
// 利用这个例子学习如何使用递归的返回值
int traverse(TreeNode root) {
    // 叶节点相当于 dp 的最后一个状态
    if (root == null)
        return 0; // 实际意义: 0 层
    // 从叶结点到根结点逆向推理 left 和 right
    int left = traverse(root.left);
    int right = traverse(root.right);
    // 下一次递归利用上一次递归 return 的结果
    return left > right ? left + 1 : right + 1;
}
```

## 二叉树的层序遍历

```
Queue<Integer> queue = new LinkedList<>();
void traverse(TreeNode root) {
    if (root != null)
        queue.offer(root);
    else
        return;

    TreeNode p = queue.poll();
    while (p != null) {
        if (p.left != null)
            queue.offer(p.left);
        if (p.right != null)
            queue.offer(p.right);
        if (!queue.isEmpty())
            p = queue.poll();
    }
}
```

## ACM 模式: import java.util.Scanner

```
// 类名必须为 Main, 不含 package xxx 信息
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // 若有下一个字符 hasNext 返回真
        // 若碰到行尾符号 hasNextLine 返回真
        // 注意 hasNextXXX 与 nextXXX 须同时出现
        while (in.hasNextInt()) { // 检查
            int a = in.nextInt();
            int b = in.nextInt(); // 指针向前移动
            // 四舍五入, 保留两位小数
            String.format("%.2f", num);
        }
    }
}
```

## 位运算

```
// 以补码形式存储, 以原码形式输出到屏幕
// int 类型按道理应该写 32 位, 下面写法不严谨
9 // 原码 01001 补码 01001 反码 01001
-1 // 原码 10001 补码 11111 反码 11110
~ 9 // ~ 01001 = 10110 原码 11010 = -10
a ^ a // = 0, 判断两数是否相同
a ^ 0x1 // 对 a 的第 0 位求反
a ^ b // 求 a + b 的各位之和, 无进位
a & b // 求 a + b 各位的进位
a & 0xFE // 关闭或检查 a 的第 0 位
a | 0x01 // 开启 a 的第 0 位
n & (~ n + 1) // 获取 n 的二进制最右侧的 1
n &= (n - 1) // 抹掉 n 的二进制最后侧的 1

int add(int a, int b) {
    int sum = a; // 求两数之和的例子
    int add = b;
    while (add != 0) {
        int tmp = sum ^ add;
        add = (sum & add) << 1;
        sum = tmp;
    }
    return sum;
}
```

若出现新的运算规则, 题目实际想让自己定义运算。

## 计算递归的时间复杂度: master 公式

$T(N) = a * T(\frac{N}{b}) + O(N^d)$   
 $N$  是问题的总规模,  $a$  是递归调用的次数,  $\frac{N}{b}$  是子问题的规模,  $O(N^d)$  是除递归代码外的时间复杂度。

- $\log_b a < d \Rightarrow O(N^d)$
- $\log_b a > d \Rightarrow O(N \log_b a)$
- $\log_b a = d \Rightarrow O(N^d \log N)$

## 二分查找：递归实现

```
// 注意这里有多处使用 return
int binarySearch(int[] arr, int target,
                int left, int right) {
    if (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target)
            return mid;
        if (arr[mid] > target) // 向左查找
            return binarySearch(arr, target,
                                left, mid - 1);
        if (arr[mid] < target) // 向右查找
            return binarySearch(arr, target,
                                mid + 1, right);
    }
    return -1; // 没找到
}
```

## 希尔排序：从小到大

```
void shellSort(int[] arr) {
    for (int step = arr.length / 2;
         step >= 1; step /= 2) {
        for (int r = step; r < len; r++) {
            int tmp = arr[r]; // 把 r 放到最终位置
            int l = r - step;
            while (l >= 0 && arr[l] > tmp) {
                arr[l + step] = arr[l]; // 将 l 右移
                l -= step;
            }
            arr[l + step] = tmp; // 放置
        }
    }
}
```

## 重建大根堆：树形数组、完全二叉树、优先队列

```
// 向已有堆的末尾插入元素，重建大根堆
void heapInsert(int[] arr, int i) {
    while (arr[i] > arr[(i - 1) / 2]) {
        swap(arr, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
// 移除堆顶元素(放在末尾)，重建大根堆
heapify(int[] arr, int i, int heapSize) {
    while (i < heapSize) {
        int l = 2 * i + 1; // 左孩子指针
        int r = 2 * i + 2; // 右孩子指针
        int max = i;
        if (l < heapSize && arr[l] > arr[max])
            max = l;
        if (r < heapSize && arr[r] > arr[max])
            max = r;
        if (max == i)
            break;
        swap(arr, i, max);
        i = max;
    }
}
```

## 插入排序 $O(n^2)$ ：从小到大

```
void insertSort(int[] arr) {
    int j; // 用于扫描 i 之前的元素
    for (int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        for (j = i; j > 0 && arr[j-1] > tmp; j--)
            arr[j] = arr[j-1]; // 向后移动元素
        arr[j] = tmp;
    }
}
```

## 堆排序：从小到大（利用 heapInsert 和 heapify）

```
PriorityQueue 就是一个小根堆结构，可以直接使用。

heapSort(int[] arr) {
    if (arr == null || arr.length < 2)
        return;
    // 构建大根堆（方法一）
    //for (int i = 0; i < arr.length; i++)
    //    heapInsert(arr, i);
    // 构建大根堆（方法二，更快）
    for (int i = arr.length - 1; i >= 0; i--)
        heapify(arr, i, arr.length);
    // 每次选择并移除堆顶元素，放到末尾
    int heapSize = arr.length;
    swap(arr, 0, --heapSize);
    while (heapSize > 0) {
        heapify(arr, 0, heapSize);
        swap(arr, 0, --heapSize);
    }
}
```

### 归并排序 $O(n\log(n))$ : 分而治之 (递归)

```
mergeSort(int[] arr, int[] tmp,
          int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, tmp, left, mid);
        mergeSort(arr, tmp, mid+1, right);
        merge(arr, tmp, left, mid, right);
    }
}

merge(int[] arr, int[] tmp,
      int left, int mid, int right) {
    int pLeft = left
    int pRight = mid + 1;
    int pTmp = left;
    // 将左右子数组较小的元素依次插入到 tmp 中
    while (pLeft <= mid && pRight <= right) {
        if (arr[pLeft] <= arr[pRight])
            tmp[pTmp++] = arr[pLeft++];
        else
            tmp[pTmp++] = arr[pRight++];
    }
    // 复制剩余元素到 tmp 中
    while (pLeft <= mid)
        tmp[pTmp++] = arr[pLeft++];
    while (pRight <= right)
        tmp[pTmp++] = arr[pRight++];
    // 必须保存局部的排序结果, 否则下次还是乱序
    for (int i = left; i <= right; i++)
        arr[i] = tmp[i];
}

调用: int[] arr = new int[]{7, 3, 2, 6};
int[] tmp = new int[arr.length]; // 辅助空间
mergeSort(arr, tmp, 0, arr.length - 1);
```

### 快速排序 $O(n\log(n))$ : 从小到大

```
quickSort(int[] arr, int left, int right) {
    if (left < right) {
        int pivot = arr[left]; // 随机选基准点
        int i = left, j = right; // 不修改原变量
        while (i < j) {
            while (i < j && arr[j] > pivot)
                j--; // 从右往左: 首个比 pivot 小的值
            if (i < j) {
                arr[i] = arr[j]; // 丢失 arr[i]
                i++;
            }
            while (i < j && arr[i] < pivot)
                i++; // 从左往右: 首个比 pivot 大的值
            if (i < j) {
                arr[j] = arr[i];
                j--;
            }
        }
        arr[i] = pivot; // 找回 arr[i]
        // -- partition 和递归代码的分割线 -- //
        quickSort(arr, left, i - 1);
        quickSort(arr, i + 1, right);
    }
}

调用: quickSort(arr, 0, arr.length - 1);
```

### 求滑动窗口中的最大值: 单调队列、双端队列

```
// 单调队列, 要始终维持队列递增或递减的状态。
// 递增(减)队列的队头是最小(大)值。
int[] maxSlidingWindow(int[] arr, int sz) {
    int[] ans = new int[arr.length - sz + 1];
    Deque<Integer> deque = new LinkedList<>();
    // r 表示滑动窗口右边界
    for (int r = 0; r < arr.length; r++) {
        // 移除队尾比当前值小的元素的索引
        while (!deque.isEmpty()
            && arr[r] >= arr[deque.peekLast()])
            deque.removeLast();
        deque.addLast(r); // 存储元素下标
        int l = r - sz + 1; // 窗口左边界
        if (deque.peekFirst() < l) // 超出左边界
            deque.removeFirst();
        if (r + 1 >= sz) // 若已经形成窗口
            ans[l] = arr[deque.peekFirst()];
    }
    return ans;
}
```

### 递归的基本思路

核心思想: 适用小问题的算法也适用于大问题

1. 找出基线条件 (递归终止条件)
2. 不断将问题分解, 直到符合基线条件

涉及数组时基线条件通常是数组为空或只含一个元素

```
// 求和: 1 + ... + 50, // traverse(dp, 0)
int traverse(int[] dp, int i) {
    // 如果依赖未来的知识, 需要知道最后状态的值
    if (i == dp.length)
        return 50;
    return i + traverse(dp, i + 1);
}
```

### 动态规划：带记忆功能的（非）递归

先暴力递归，再使用优化技巧（消除重复计算）。

1. 直接写出初始状态  $dp[0]$ ,  $dp[1]$  的答案;
  2. 当前状态  $dp[i]$  的通常是题目要求的结果;
  3.  $dp[i]$  与  $dp[i-2:i+2]$  有何关联?
  4.  $dp[i,j]$  与  $dp[i-2:i+2,j-2:j+2]$  有何关联?
- 注:  $dp[i]$  是状态数组, 表示第  $i$  个状态, 也就是数组中的第  $i$  个元素。通常, 整个数组作为**额外的不变参数**一直传递, 用于判断是否访问越界。

// 1. 暴力递归求斐波那契数列: 0 1 1 2 3 ...

```
int fib(int n) { // 求最后状态 dp[n] 的值
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

// 2. 将递归转为动态规划（消除重复计算）

```
int fib(int n) {
    int[] dp = new int[3];
    dp[0] = 0; dp[1] = 1; // 初始记忆
    dp[2] = dp[0] + dp[1]; // dp[2] 是最新记忆
    for (int i = 2; i <= n; i++) {
        dp[0] = dp[1]; dp[1] = dp[2]; // 更新记忆
        dp[2] = dp[0] + dp[1]; // 依赖前两个记忆
    }
    return dp[2];
}
```

### 全排列问题：回溯、N 叉树遍历、穷举

```
// 比如：模拟从黑箱子中取球的过程（有放回）
// 回溯不同于动态规划，动态规划有公式可循
// 用 arr 表示原始数组，用 used 剪枝优化
// 用 i == arr.length 判断递归是否终止
List<List<Integer>> ans = new ArrayList<>();
List<Integer> path = new ArrayList<>();
void dfs(int[] arr, boolean[] used, int i) {
    if (i == arr.length) {
        // 注意，深拷贝
        ans.add(new ArrayList<>(path));
        return;
    }
    // 每次都向 path 的第 j 个位置推送不同数字
    for (int j = 0; j < arr.length; j++) {
        if (!used[j]) {
            path.add(arr[j]);
            used[j] = true;
            dfs(arr, used, i + 1);
            used[j] = false; // 撤销原操作
            path.remove(path.size() - 1);
        }
    }
}
```

### 岛问题：求连通区域的个数

```
// "感染" 每个可连通的单元，由 1 变成 2
void infect(int[][] arr, int i, int j,
            int N, int M) {
    if (i < 0 || i >= N || j < 0 || j >= M
        || arr[i][j] != 1)
        return;
    arr[i][j] = 2; // 感染
    infect(arr, i+1, j, N, M);
    infect(arr, i-1, j, N, M);
    infect(arr, i, j+1, N, M);
    infect(arr, i, j-1, N, M);
}

int count(int[][] arr) {
    if (arr == null || arr[0] == null)
        return 0;
    int N = arr.length;
    int M = arr[0].length;
    int ans = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            if (arr[i][j] == 1) {
                ans++;
                infect(arr, i, j, N, M);
            }
}
```

### KMP 算法: 求 next 数组

```
// next 数组记录最长相等的前后缀长度
void getNext(int[] next, String pat) {
    next[0] = 0;
    int j = 0; // 失配后的回退点
    // 循环从 1 开始, 不是 0
    for (int i = 1; i < pat.length(); i++) {
        char chi = pat.charAt(i);
        char chj = pat.charAt(j);
        while (j > 0 && chi != chj)
            j = next[j - 1]; // 回退
        if (chi == chj)
            j++;
        next[i] = j;
    }
}

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

### 中序遍历: 迭代实现

```
// 需要借助栈和指针来实现
void inorder(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode cur = root;
    while (cur != null || !stack.isEmpty()) {
        if (cur != null) { // 遍历左子节点, 入栈
            stack.push(cur);
            cur = cur.left;
        }
        else { // 遍历完左子节点, 出栈, 保存结果
            cur = stack.peek();
            stack.pop();
            // ans.add(cur.val);
            cur = cur.right;
        }
    }
}
```

### 前序遍历: 迭代实现

```
void preorder(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    if (root == null)
        return;
    stack.push(root);
    while (!stack.isEmpty()) {
        TreeNode cur = stack.peek();
        stack.pop();
        // ans.add(cur.val);
        if (cur.right != null) // 先右后左
            stack.push(cur.right);
        if (cur.left != null)
            stack.push(cur.left);
    }
}
```

### 优先级队列: 默认是小根堆

```
// 默认的初始化方法
PriorityQueue<Integer> pq = new PriorityQueue<>();

// 自定义排序规则
PriorityQueue<Integer> pq = new PriorityQueue<>() {
    new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o1 - o2; // (升序) 谁小谁优先
        }
    }
};
```

### 后序遍历: 迭代实现

```
// 和前序遍历类似的代码, 也需要借助栈来实现
// 遍历顺序不一样, 且多了一个 reverse 环节
void postorder(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode cur = root;
    if (root == null)
        return;
    stack.push(root);
    while (!stack.isEmpty()) {
        TreeNode cur = stack.peek();
        stack.pop();
        // ans.add(cur.val);
        if (cur.left != null) // 先左后右
            stack.push(cur.left);
        if (cur.right != null)
            stack.push(cur.right);
    }
    reverse(ans);
}
```

## 后序遍历：迭代实现

```
reverseList(ListNode head) {  
    if (head == null || head.next == null)  
        return head;  
  
    // 三指针翻转链表  
    ListNode pre = null;  
    ListNode cur = head;  
    while (cur != null) {  
        ListNode nxt = cur.next; // 临时保存  
        cur.next = pre;  
        // 更新节点  
        pre = cur;  
        cur = nxt;  
    }  
    return pre;  
}
```