<https://algo.itcharge.cn/>

<https://chat.qwen.ai/>

# 常用数据结构&API

List<Integer> res = new ArrayList<>();

int[] res\_1 = res.stream().mapToInt(Number::intValue).toArray();

在 Java 17 中，将 Deque<Character> 转换为 String

import java.util.\*;

Deque<Character> deque = new ArrayDeque<>();

deque.add('H');

deque.add('e');

deque.add('l');

deque.add('l');

deque.add('o');

✅ 优点：性能好、内存利用率高、适合大字符集。

StringBuilder sb = new StringBuilder();

for (char c : deque) {

sb.append(c);

}

String result = sb.toString();

System.out.println(result); // "Hello"

✅ 优点：代码简洁，适合函数式编程风格  
⚠️ 注意：性能略低于 StringBuilder，小数据量无感

String result = deque.stream()

.map(String::valueOf) // 将 Character 转为 String

.collect(Collectors.joining());

System.out.println(result); // "Hello"

## java.util.HashMap

基于哈希表（Hash Table） 实现的 Map 接口，提供高效的增删改查操作，平均时间复杂

度为 O(1)。

不保证元素的顺序（插入顺序、自然顺序）

**JDK 8+ 新增的便捷方法（函数式增强）**

|  |  |
| --- | --- |
| getOrDefault(Object key, V defaultValue) | 获取值，若不存在则返回默认值 |
| putIfAbsent(K key, V value) | 若键不存在才插入 |
| computeIfAbsent(K key, Function mappingFunction) | 若键不存在，则用函数计算值并放入 |
| computeIfPresent(K key, BiFunction remappingFunction) | 若键存在，则重新计算值 |
| merge(K key, V value, BiFunction<V,V,V> remappingFunction) | 合并值（如统计词频） |
| forEach(BiConsumer<K,V>) | 遍历键值对（函数式） |

int count = map.getOrDefault("orange", 0); // 如果没有 orange，返回 0

map.putIfAbsent("apple", 99); // 只有 apple 不存在时才设置

// 构建 Map<String, List<String>> 防止空指针

Map<String, List<String>> grouped = new HashMap<>();

grouped.computeIfAbsent("group1", k -> new ArrayList<>()).add("item1");

Map<String, Integer> wordCount = new HashMap<>();

String[] words = {"hello", "world", "hello", "java"};

for (String word : words) {

wordCount.merge(word, 1, Integer::sum);

}

// 结果: {hello=2, world=1, java=1}

map.forEach((key, value) -> System.out.println(key + ": " + value));

## java.util.TreeMap

基于红黑树（Red-Black Tree）实现的有序映射（SortedMap），它会根据键（key）的自然顺序或自定义比较器进行排序。

**适用场景**

* 需要按键有序存储（如排行榜、时间轴）
* 范围查询（如查找 10~50 分之间的学生）
* 查找“最接近”的键（floorKey, ceilingKey）
* 实现 LRU 缓存的有序管理（配合双向链表）

**基础增删查改**

|  |  |
| --- | --- |
| put(K key, V value) | 插入键值对，若键已存在则替换值 |
| get(Object key) | 获取指定键的值，不存在返回null |
| **remove(Object key)** | **删除指定键的映射** |
| **containsKey(Object key)** | **判断是否包含某个键** |
| **containsValue(Object value)** | **判断是否包含某个值（较慢，O(n)）** |
| size() | 返回映射数量 |
| isEmpty() | 是否为空 |

**有序性相关方法**

|  |  |
| --- | --- |
| firstKey()/firstEntry() | 返回最小键 / 最小键值对 |
| lastKey()/lastEntry() | 返回最大键 / 最大键值对 |
| lowerKey(K key)/lowerEntry(K key) | 返回小于给定键的最大键及其映射 |
| floorKey(K key)/floorEntry(K key) | 返回小于等于给定键的最大键 |
| ceilingKey(K key)/ceilingEntry(K key) | 返回大于等于给定键的最小键 |
| higherKey(K key)/higherEntry(K key) | 返回大于给定键的最小键 |
| pollFirstEntry() | 获取并移除最小键值对 |
| pollLastEntry() | 获取并移除最大键值对 |

TreeMap<Integer, String> map = new TreeMap<>();

map.put(10, "A");

map.put(20, "B");

map.put(30, "C");

System.out.println(map.firstKey()); // 10

System.out.println(map.lastKey()); // 30

System.out.println(map.lowerKey(25)); // 20

System.out.println(map.ceilingKey(25)); // 30

System.out.println(map.floorKey(20)); // 20

System.out.println(map.higherKey(20)); // 30

System.out.println(map.pollFirstEntry()); // 10=A

System.out.println(map.size()); // 2

**子映射（Submap）操作**

|  |  |
| --- | --- |
| subMap(K fromKey, K toKey) | 返回[fromKey, toKey)范围的视图 |
| headMap(K toKey) | 返回小于toKey的所有键的视图 |
| tailMap(K fromKey) | 返回大于等于fromKey的所有键的视图 |
| subMap(K from, boolean inclusive, K to, boolean inclusive) | 可指定是否包含边界 |

TreeMap<Integer, String> map = new TreeMap<>();

map.put(10, "A");

map.put(20, "B");

map.put(30, "C");

map.put(40, "D");

SortedMap<Integer, String> sub = map.subMap(15, 35); // [20, 30)

System.out.println(sub); // {20=B, 30=C}

SortedMap<Integer, String> head = map.headMap(25); // < 25

System.out.println(head); // {10=A, 20=B}

SortedMap<Integer, String> tail = map.tailMap(30); // >= 30

System.out.println(tail); // {30=C, 40=D}

**遍历方式**

遍历 Entry（推荐）

for (Map.Entry<String, Integer> entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

遍历 Key

for (String key : map.keySet()) {

System.out.println(key);

}

使用 Iterator（可删除）

Iterator<Map.Entry<Integer, String>> it = map.entrySet().iterator();

while (it.hasNext()) {

Map.Entry<Integer, String> entry = it.next();

if (entry.getValue().equals("B")) {

it.remove(); // 安全删除

}

}

## java.util.ArrayDeque

高性能双端队列，性能优于 LinkedList 和旧的 Stack 类。

不允许 null

作为双端队列使用（头尾操作）

**offer/poll/peek 方法更安全，失败时返回 null 或 false，而 add/remove/get 失败抛异常。**

|  |  |
| --- | --- |
| addFirst(e)/offerFirst(e) | 在头部添加元素 |
| addLast(e)/offerLast(e) | 在尾部添加元素 |
| removeFirst()/pollFirst() | 移除并返回头部元素 |
| removeLast()/pollLast() | 移除并返回尾部元素 |
| getFirst()/peekFirst() | 查看头部元素（不删除） |
| getLast()/peekLast() | 查看尾部元素（不删除） |

作为普通队列使用（FIFO）

|  |  |
| --- | --- |
| offer(e)/add(e) | 入队（等价于addLast） |
| poll()/remove() | 出队（等价于removeFirst） |
| peek()/element() | 查看队首 |

作为栈使用（LIFO）——推荐替代 Stack 类

|  |  |
| --- | --- |
| push(e) | 入栈（等价于addFirst） |
| pop() | 出栈（等价于removeFirst） |
| peek() | 查看栈顶 |

其他常用方法

|  |  |
| --- | --- |
| size() | 返回元素个数 |
| isEmpty() | 是否为空 |
| contains(Object o) | 是否包含某元素 |
| iterator() | 返回从头到尾的迭代器 |
| descendingIterator() | 返回从尾到头的迭代器 |

// 创建空的双端队列

ArrayDeque<Integer> deque = new ArrayDeque<>();

// 指定初始容量（可选）

ArrayDeque<String> deque2 = new ArrayDeque<>(16);

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.addLast(10); // [10]

deque.addLast(20); // [10, 20]

deque.addFirst(5); // [5, 10, 20]

System.out.println(deque.getFirst()); // 5

System.out.println(deque.getLast()); // 20

System.out.println(deque.removeFirst()); // 5 → [10, 20]

System.out.println(deque.removeLast()); // 20 → [10]

ArrayDeque<String> queue = new ArrayDeque<>();

queue.offer("A"); // A

queue.offer("B"); // A B

queue.offer("C"); // A B C

while (!queue.isEmpty()) {

System.out.print(queue.poll() + " "); // A B C

}

ArrayDeque<Integer> stack = new ArrayDeque<>();

stack.push(1); // 1

stack.push(2); // 2 1

stack.push(3); // 3 2 1

System.out.println(stack.peek()); // 3

System.out.println(stack.pop()); // 3

System.out.println(stack.pop()); // 2

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.add(1); // 1

deque.add(2); // 1 2

deque.add(3); // 1 2 3

// 正向遍历

for (Integer n : deque) {

System.out.print(n + " "); // 1 2 3

}

// 反向遍历（适合栈遍历）

Iterator<Integer> it = deque.descendingIterator();

while (it.hasNext()) {

System.out.print(it.next() + " "); // 3 2 1

}

**用 ArrayDeque 实现回文检查**

import java.util.ArrayDeque;

import java.util.Deque;

public class PalindromeChecker {

public static boolean isPalindrome(String str) {

Deque<Character> deque = new ArrayDeque<>();

String cleaned = str.toLowerCase().replaceAll("[^a-z0-9]", "");

for (char c : cleaned.toCharArray()) {

deque.addLast(c);

}

while (deque.size() > 1) {

if (!deque.removeFirst().equals(deque.removeLast())) {

return false;

}

}

return true;

}

public static void main(String[] args) {

System.out.println(isPalindrome("A man a plan a canal Panama")); // true

System.out.println(isPalindrome("race a car")); // false

}

}

## java.util.Arrays

1. 排序前记得 sort()，否则 binarySearch() 结果错误。
2. 修改 asList() 返回的 List 会抛异常，如需修改请用 new ArrayList<>(Arrays.asList(arr))。
3. 多维数组比较用 deepEquals()，打印用 deepToString()。
4. 大数据量处理优先使用 Arrays.stream() 配合 Lambda。

用途：将一维数组转换为可读的字符串格式。

int[] arr = {3, 1, 4, 1, 5};

System.out.println(Arrays.toString(arr)); // 输出: [3, 1, 4, 1, 5]

int[] arr = {3, 1, 4, 1, 5};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr)); // [1, 1, 3, 4, 5]

Arrays.deepToString(multiArray)

用途：打印多维数组（如二维数组）。

int[][] matrix = {{1, 2}, {3, 4}, {5, 6}};

System.out.println(Arrays.deepToString(matrix));

// [[1, 2], [3, 4], [5, 6]]

用途：对数组进行排序（升序），使用优化的快速排序/归并排序算法。

// 字符串数组排序（字典序）

String[] strs = {"Java", "C++", "Python"};

Arrays.sort(strs);

System.out.println(Arrays.toString(strs)); // [C++, Java, Python]

**自定义排序（配合 Comparator）**

Integer[] nums = {3, 1, 4, 1, 5};

Arrays.sort(nums, Collections.reverseOrder()); // 降序

System.out.println(Arrays.toString(nums)); // [5, 4, 3, 1, 1]

Arrays.binarySearch(array, key)

用途：在已排序的数组中使用二分查找，返回目标元素的索引，找不到返回负值。

int[] arr = {1, 3, 4, 5, 7, 9};

int index = Arrays.binarySearch(arr, 5);

System.out.println("5 的索引: " + index); // 3

int notFound = Arrays.binarySearch(arr, 6);

System.out.println("6 的索引: " + notFound); // -5（插入点为 -(-5)-1=4）

Arrays.copyOf(original, newLength)

用途：复制数组，可以指定新长度（支持扩容或截断）。

int[] arr = {1, 2, 3};

int[] copy = Arrays.copyOf(arr, 5); // 扩容到5，补0

System.out.println(Arrays.toString(copy)); // [1, 2, 3, 0, 0]

int[] shorter = Arrays.copyOf(arr, 2); // 截断

System.out.println(Arrays.toString(shorter)); // [1, 2]

Arrays.copyOfRange(original, from, to)

用途：复制数组的指定范围 [from, to)（左闭右开）。

int[] arr = {1, 2, 3, 4, 5};

int[] range = Arrays.copyOfRange(arr, 1, 4);

System.out.println(Arrays.toString(range)); // [2, 3, 4]

Arrays.equals(array1, array2)

用途：判断两个数组是否“相等”（长度相同且对应元素相等）。

int[] a = {1, 2, 3};

int[] b = {1, 2, 3};

System.out.println(Arrays.equals(a, b)); // true

int[] c = {1, 2};

System.out.println(Arrays.equals(a, c)); // false

Arrays.fill(array, value)

用途：用指定值填充整个数组或部分范围。

int[] arr = new int[5];

Arrays.fill(arr, 7);

System.out.println(Arrays.toString(arr)); // [7, 7, 7, 7, 7]

// 填充部分

Arrays.fill(arr, 1, 4, 9); // 索引 [1,4) 填为 9

System.out.println(Arrays.toString(arr)); // [7, 9, 9, 9, 7]

Arrays.asList(array)

用途：将数组转换为 List，但注意：返回的是固定大小的 List，不能增删元素。

String[] strs = {"Java", "Python", "Go"};

List<String> list = Arrays.asList(strs);

System.out.println(list); // [Java, Python, Go]

// ❌ list.add("Rust"); // 抛出 UnsupportedOperationException

Arrays.stream(array)**（Java 8+）**

用途：将数组转换为 Stream，便于使用函数式编程。

int[] numbers = {1, 2, 3, 4, 5};

int sum = Arrays.stream(numbers).sum();

System.out.println("总和: " + sum); // 15

// 过滤并打印偶数

Arrays.stream(numbers)

.filter(n -> n % 2 == 0)

.forEach(System.out::println);

## java.util.TreeSet

特点：

1. 有序（默认自动升序）
2. 去重集合
3. 不允许null
4. 插入、删除、查找时间复杂度：O(log n)

基本操作

|  |  |
| --- | --- |
| add(E e) | 添加元素 |
| remove(Object o) | 删除元素 |
| contains(Object o) | 判断是否包含元素 |
| size() | 返回元素个数 |
| isEmpty() | 是否为空 |

Set<Integer> set\_1 = new TreeSet<>();  
set\_1.add(3);  
set\_1.add(1);  
set\_1.add(2);  
System.*out*.println(set\_1); // [1, 2, 3]  
  
Set<Integer> set\_2 = new TreeSet<>(new Comparator<Integer>() {  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 return o2 - o1;  
 }  
});

// 或者

// Set<Integer> set\_2 = new TreeSet<>((o1, o2) -> o2 - o1);

// TreeSet<Integer> set\_2 = new TreeSet<>((a, b) -> b.compareTo(a));  
set\_2.add(1);  
set\_2.add(3);  
set\_2.add(2);  
System.*out*.println(set\_2); // [3, 2, 1]

System.out.println("集合: " + set); // [5, 10, 20]

System.out.println("大小: " + set.size()); // 3

System.out.println("包含10? " + set.contains(10)); // true

set.remove(20);

排序相关方法

|  |  |
| --- | --- |
| first() | 返回最小元素 |
| last() | 返回最大元素 |
| headSet(E toElement) | 返回小于toElement的子集（不包含 toElement） |
| tailSet(E fromElement) | 返回大于等于fromElement的子集 |
| subSet(E from, E to) | 返回[from, to)范围内的子集 |

TreeSet<Integer> set = new TreeSet<>();

set.addAll(Arrays.asList(5, 10, 15, 20, 25));

System.out.println("最小值: " + set.first()); // 5

System.out.println("最大值: " + set.last()); // 25

System.out.println("小于15: " + set.headSet(15)); // [5, 10]

System.out.println("大于等于15: " + set.tailSet(15)); // [15, 20, 25]

System.out.println("10到20之间: " + set.subSet(10, 20)); // [10, 15]

**实用方法**

|  |  |
| --- | --- |
| lower(E e) | 返回小于e的最大元素，不存在则返回null |
| floor(E e) | 返回小于等于e的最大元素 |
| higher(E e) | 返回大于e的最小元素 |
| ceiling(E e) | 返回大于等于e的最小元素 |
| pollFirst() | 获取并移除最小元素 |
| pollLast() | 获取并移除最大元素 |

TreeSet<Integer> set = new TreeSet<>();

set.addAll(Arrays.asList(5, 10, 15, 20));

System.out.println("lower(12): " + set.lower(12)); // 10

System.out.println("floor(10): " + set.floor(10)); // 10

System.out.println("higher(12): " + set.higher(12)); // 15

System.out.println("ceiling(10): " + set.ceiling(10)); // 10

System.out.println("弹出最小: " + set.pollFirst()); // 5

System.out.println("当前集合: " + set); // [10, 15, 20]

System.out.println("弹出最大: " + set.pollLast()); // 20

System.out.println("当前集合: " + set); // [10, 15]

遍历方式

TreeSet<Integer> set = new TreeSet<>(Arrays.asList(3, 1, 4, 1, 5));

// 方式1：增强 for 循环

for (Integer num : set) {

System.out.print(num + " "); // 1 3 4 5

}

// 方式2：迭代器（升序）

Iterator<Integer> it = set.iterator();

while (it.hasNext()) {

System.out.print(it.next() + " ");

}

// 方式3：降序遍历

Iterator<Integer> descIt = set.descendingIterator();

while (descIt.hasNext()) {

System.out.print(descIt.next() + " "); // 5 4 3 1

}

# 二分查找

## 模板

int binarySearch(int[] nums, int target) {  
 int left = 0;  
 int right = nums.length - 1; // 注意  
 while(left <= right) {  
 int mid = left + (right - left) / 2;  
 if(nums[mid] == target)  
 return mid;  
 else if (nums[mid] < target)  
 left = mid + 1; // 注意  
 else if (nums[mid] > target)  
 right = mid - 1; // 注意  
 }  
 return -1;  
}  
  
int left\_bound(int[] nums, int target) {  
 int left = 0, right = nums.length - 1;  
 // 搜索区间为 [left, right]  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 if (nums[mid] < target) {  
 // 搜索区间变为 [mid+1, right]  
 left = mid + 1;  
 } else if (nums[mid] > target) {  
 // 搜索区间变为 [left, mid-1]  
 right = mid - 1;  
 } else if (nums[mid] == target) {  
 // 收缩右侧边界  
 right = mid - 1;  
 }  
 }  
 // 检查出界情况  
 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) {  
 // 这⾥改成收缩左侧边界即可  
 left = mid + 1;  
 }  
 }  
 // 这⾥改为检查 right 越界的情况，⻅下图  
 if (right < 0 || nums[right] != target)  
 return -1;  
 return right;  
}

## 题目

### 704. 二分查找

<https://leetcode.cn/problems/binary-search/description/?envType=problem-list-v2&envId=6uaxYMyj>

给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target  ，写一个函数搜索 nums 中的 target，如果 target 存在返回下标，否则返回 -1。

你必须编写一个具有 O(log n) 时间复杂度的算法。

**示例 1:**

**输入:** nums = [-1,0,3,5,9,12], target = 9

**输出:** 4

**解释:** 9 出现在 nums 中并且下标为 4

**示例 2:**

**输入:** nums = [-1,0,3,5,9,12], target = 2

**输出:** -1

**解释:** 2 不存在 nums 中因此返回 -1

class Solution {

    public int 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 {

                return mid;

            }

        }

        return -1;

    }

}

### 374. 猜数字大小

<https://leetcode.cn/problems/guess-number-higher-or-lower/description/?envType=problem-list-v2&envId=6uaxYMyj>

我们正在玩猜数字游戏。猜数字游戏的规则如下：

我会从 1 到 n 随机选择一个数字。 请你猜选出的是哪个数字。

如果你猜错了，我会告诉你，我选出的数字比你猜测的数字大了还是小了。

你可以通过调用一个预先定义好的接口 int guess(int num) 来获取猜测结果，返回值一共有三种可能的情况：

* -1：你猜的数字比我选出的数字大 （即 num > pick）。
* 1：你猜的数字比我选出的数字小 （即 num < pick）。
* 0：你猜的数字与我选出的数字相等。（即 num == pick）。

返回我选出的数字。

**示例 1：**

**输入：**n = 10, pick = 6

**输出：**6

**示例 2：**

**输入：**n = 1, pick = 1

**输出：**1

**示例 3：**

**输入：**n = 2, pick = 1

**输出：**1

public class Solution extends GuessGame {

    public int guessNumber(int n) {

        int left = 0, right = n;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (guess(mid) == 1) {

                left = mid + 1;

            } else if (guess(mid) == -1) {

                right = mid - 1;

            } else {

                return mid;

            }

        }

        return -1;

    }

}

### 34. 在排序数组中查找元素的第一个和最后一个位置

<https://leetcode.cn/problems/find-first-and-last-position-of-element-in-sorted-array/description/?envType=problem-list-v2&envId=6uaxYMyj>

给你一个按照非递减顺序排列的整数数组 nums，和一个目标值 target。请你找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 target，返回 [-1, -1]。

你必须设计并实现时间复杂度为 O(log n) 的算法解决此问题。

**示例 1：**

**输入：**nums = [5,7,7,8,8,10], target = 8

**输出：**[3,4]

**示例 2：**

**输入：**nums = [5,7,7,8,8,10], target = 6

**输出：**[-1,-1]

**示例 3：**

**输入：**nums = [], target = 0

**输出：**[-1,-1]

class Solution {

    public int[] searchRange(int[] nums, int target) {

        int left = getleft(nums, target), right = getright(nums, target);

        return new int[]{left, right};

    }

    private int getleft(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 {

                right = mid-1;

            }

        }

        if (left >= nums.length || nums[left] != target) {

            return -1;

        }

        return left;

    }

    public int getright(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 {

                left = mid+1;

            }

        }

        if (right < 0 || nums[right] != target) {

            return -1;

        }

        return right;

    }

}

### 35. 搜索插入位置

<https://leetcode.cn/problems/search-insert-position/description/?envType=problem-list-v2&envId=6uaxYMyj>

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 O(log n) 的算法。

**示例 1:**

**输入:** nums = [1,3,5,6], target = 5

**输出:** 2

**示例 2:**

**输入:** nums = [1,3,5,6], target = 2

**输出:** 1

**示例 3:**

**输入:** nums = [1,3,5,6], target = 7

**输出:** 4

class Solution {

    public int searchInsert(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 {

                return mid;

            }

        }

        return left;

    }

}

# 双指针

## 模板

## 题目

### 27.移除元素

<https://leetcode.cn/problems/remove-element/description/?envType=problem-list-v2&envId=6uaxYMyj>

给你一个数组 nums和一个值 val，你需要 [**原地**](https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95) 移除所有数值等于 val的元素。元素的顺序可能发生改变。然后返回 nums 中与 val 不同的元素的数量。

假设 nums 中不等于 val 的元素数量为 k，要通过此题，您需要执行以下操作：

* 更改 nums 数组，使 nums 的前 k 个元素包含不等于 val 的元素。nums 的其余元素和 nums 的大小并不重要。
* 返回 k。

**用户评测：**

评测机将使用以下代码测试您的解决方案：

int[] nums = [...]; // 输入数组

int val = ...; // 要移除的值

int[] expectedNums = [...]; // 长度正确的预期答案。

// 它以不等于 val 的值排序。

int k = removeElement(nums, val); // 调用你的实现

assert k == expectedNums.length;

sort(nums, 0, k); // 排序 nums 的前 k 个元素

for (int i = 0; i < actualLength; i++) {

assert nums[i] == expectedNums[i];

}

如果所有的断言都通过，你的解决方案将会 **通过**。

**示例 1：**

**输入：**nums = [3,2,2,3], val = 3

**输出：**2, nums = [2,2,\_,\_]

**解释：**你的函数函数应该返回 k = 2, 并且 nums中的前两个元素均为 2。

你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。

**示例 2：**

**输入：**nums = [0,1,2,2,3,0,4,2], val = 2

**输出：**5, nums = [0,1,4,0,3,\_,\_,\_]

**解释：**你的函数应该返回 k = 5，并且 nums 中的前五个元素为 0,0,1,3,4。

注意这五个元素可以任意顺序返回。

你在返回的 k 个元素之外留下了什么并不重要（因此它们并不计入评测）。

class Solution {

    public int removeElement(int[] nums, int val) {

        int slow = 0, fast = 0;

        while (fast < nums.length) {

            if (nums[fast] != val) {

                int temp = nums[fast];

                nums[fast] = nums[slow];

                nums[slow] = temp;

                slow++;

            }

            fast++;

        }

        return slow;

    }

}

### 167. 两数之和 II - 输入有序数组

<https://leetcode.cn/problems/two-sum-ii-input-array-is-sorted/description/?envType=problem-list-v2&envId=6uaxYMyj>

给你一个下标从 **1** 开始的整数数组 numbers ，该数组已按**非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 target 的两个数。如果设这两个数分别是 numbers[index1] 和 numbers[index2] ，则 1 <= index1 < index2 <= numbers.length 。

以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1和index2。

你可以假设每个输入 **只对应唯一的答案** ，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

**示例 1：**

**输入：**numbers = [***2***,***7***,11,15], target = 9

**输出：**[1,2]

**解释：**2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。

**示例 2：**

**输入：**numbers = [***2***,3,***4***], target = 6

**输出：**[1,3]

**解释：**2 与 4 之和等于目标数 6 。因此 index1 = 1, index2 = 3 。返回 [1, 3] 。

**示例 3：**

**输入：**numbers = [***-1***,***0***], target = -1

**输出：**[1,2]

**解释：**-1 与 0 之和等于目标数 -1 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。

class Solution {

public int[] twoSum(int[] numbers, int target) {

int left = 0;

int right = numbers.length - 1;

while (left < right) {

if (numbers[left] + numbers[right] > target) {

right--;

} else if (numbers[left] + numbers[right] < target) {

left++;

} else {

break;

}

}

return new int[]{left + 1, right + 1};

}

}

### 153. 寻找旋转排序数组中的最小值

<https://leetcode.cn/problems/find-minimum-in-rotated-sorted-array/description/?envType=problem-list-v2&envId=6uaxYMyj>

思路：

<https://algo.itcharge.cn/solutions/0100-0199/find-minimum-in-rotated-sorted-array/>

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 **旋转** 后，得到输入数组。例如，原数组 nums = [0,1,2,4,5,6,7] 在变化后可能得到：

* 若旋转 4 次，则可以得到 [4,5,6,7,0,1,2]
* 若旋转 7 次，则可以得到 [0,1,2,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] **旋转一次** 的结果为数组 [a[n-1], a[0], a[1], a[2], ..., a[n-2]] 。

给你一个元素值 **互不相同** 的数组 nums ，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素** 。

你必须设计一个时间复杂度为 O(log n) 的算法解决此问题。

**示例 1：**

**输入：**nums = [3,4,5,1,2]

**输出：**1

**解释：**原数组为 [1,2,3,4,5] ，旋转 3 次得到输入数组。

**示例 2：**

**输入：**nums = [4,5,6,7,0,1,2]

**输出：**0

**解释：**原数组为 [0,1,2,4,5,6,7] ，旋转 4 次得到输入数组。

**示例 3：**

**输入：**nums = [11,13,15,17]

**输出：**11

**解释：**原数组为 [11,13,15,17] ，旋转 4 次得到输入数组。

class Solution {

public int findMin(int[] nums) {

int left=0,right=nums.length-1;

while (left < right) {

int mid = left +(right-left)/2;

if (nums[mid] > nums[right]) {

left=mid+1;

} else {

right=mid;

}

}

return nums[left];

}

}

### 154. 寻找旋转排序数组中的最小值II

<https://leetcode.cn/problems/find-minimum-in-rotated-sorted-array-ii/description/?envType=problem-list-v2&envId=6uaxYMyj>

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 **旋转** 后，得到输入数组。例如，原数组 nums = [0,1,4,4,5,6,7] 在变化后可能得到：

* 若旋转 4 次，则可以得到 [4,5,6,7,0,1,4]
* 若旋转 7 次，则可以得到 [0,1,4,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] **旋转一次** 的结果为数组 [a[n-1], a[0], a[1], a[2], ..., a[n-2]] 。

给你一个可能存在 **重复** 元素值的数组 nums ，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素** 。

你必须尽可能减少整个过程的操作步骤。

**示例 1：**

**输入：**nums = [1,3,5]

**输出：**1

**示例 2：**

**输入：**nums = [2,2,2,0,1]

**输出：**0

class Solution {

public int findMin(int[] nums) {

int left=0,right=nums.length-1;

while (left < right) {

int mid = left +(right-left)/2;

if (nums[mid] < nums[right]) {

right = mid;

} else if (nums[mid] > nums[right]) {

left = mid+1;

} else {

right--;

}

}

return nums[left];

}

}

### 33.搜索旋转排序数组

<https://leetcode.cn/problems/search-in-rotated-sorted-array/description/?envType=problem-list-v2&envId=6uaxYMyj>

题解：

<https://algo.itcharge.cn/solutions/0001-0099/search-in-rotated-sorted-array/#%E6%80%9D%E8%B7%AF-1-%E4%BA%8C%E5%88%86%E6%9F%A5%E6%89%BE>

整数数组 nums 按升序排列，数组中的值 **互不相同** 。

在传递给函数之前，nums 在预先未知的某个下标 k（0 <= k < nums.length）上进行了 **旋转**，使数组变为 [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]（下标 **从 0 开始** 计数）。例如， [0,1,2,4,5,6,7] 向左旋转 3 次后可能变为 [4,5,6,7,0,1,2] 。

给你 **旋转后** 的数组 nums 和一个整数 target ，如果 nums 中存在这个目标值 target ，则返回它的下标，否则返回 -1 。

你必须设计一个时间复杂度为 O(log n) 的算法解决此问题。

**示例 1：**

**输入：**nums = [4,5,6,7,0,1,2], target = 0

**输出：**4

**示例 2：**

**输入：**nums = [4,5,6,7,0,1,2], target = 3

**输出：**-1

**示例 3：**

**输入：**nums = [1], target = 0

**输出：**-1

class Solution {

public int search(int[] nums, int target) {

int n = nums.length;

int left = 0, right = n-1;

while (left <= right) {

int mid= left + (right-left)/2;

if (nums[mid] == target) {

return mid;

}

if (nums[0] <= nums[mid]) {

if (nums[0] <= target && target < nums[mid]) {

right=mid-1;

} else {

left=mid+1;

}

} else {

if (nums[mid] < target && target <= nums[n-1]) {

left=mid+1;

} else {

right=mid-1;

}

}

}

return -1;

}

}

### 81.搜索旋转排序数组II

<https://leetcode.cn/problems/search-in-rotated-sorted-array-ii/description/?envType=problem-list-v2&envId=6uaxYMyj>

题解：

<https://algo.itcharge.cn/solutions/0001-0099/search-in-rotated-sorted-array-ii/>

已知存在一个按非降序排列的整数数组 nums ，数组中的值不必互不相同。

在传递给函数之前，nums 在预先未知的某个下标 k（0 <= k < nums.length）上进行了 **旋转**，使数组变为 [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]（下标 **从 0 开始** 计数）。例如， [0,1,2,4,4,4,5,6,6,7] 在下标 5 处经旋转后可能变为 [4,5,6,6,7,0,1,2,4,4] 。

给你 **旋转后** 的数组 nums 和一个整数 target ，请你编写一个函数来判断给定的目标值是否存在于数组中。如果 nums 中存在这个目标值 target ，则返回 true ，否则返回 false 。

你必须尽可能减少整个操作步骤。

**示例 1：**

**输入：**nums = [2,5,6,0,0,1,2], target = 0

**输出：**true

**示例 2：**

**输入：**nums = [2,5,6,0,0,1,2], target = 3

**输出：**false

**提示：**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* 题目数据保证 nums 在预先未知的某个下标上进行了旋转
* -104 <= target <= 104

class Solution {

public boolean search(int[] nums, int target) {

int n = nums.length;

int left=0,right=n-1;

while (left<=right) {

int mid = left+(right-left)/2;

if (nums[mid] == target){

return true;

}

if (nums[mid] == nums[left] && nums[mid] == nums[right]) {

left++;

right--;

} else if (nums[left] <= nums[mid]) {

if (nums[left] <= target && target < nums[mid]) {

right=mid-1;

} else {

left=mid+1;

}

} else {

if (nums[mid] < target && target <= nums[n-1]) {

left=mid+1;

} else {

right=mid-1;

}

}

}

return false;

}

}

### 278.第一个错误版本

<https://leetcode.cn/problems/first-bad-version/description/?envType=problem-list-v2&envId=6uaxYMyj>

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。

假设你有 n 个版本 [1, 2, ..., n]，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 bool isBadVersion(version) 接口来判断版本号 version 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

**示例 1：**

**输入：**n = 5, bad = 4

**输出：**4

**解释：**

调用 isBadVersion(3) -> false

调用 isBadVersion(5) -> true

调用 isBadVersion(4) -> true

所以，4 是第一个错误的版本。

**示例 2：**

**输入：**n = 1, bad = 1

**输出：**1

**提示：**

* 1 <= bad <= n <= 231 - 1

/\* The isBadVersion API is defined in the parent class VersionControl.

boolean isBadVersion(int version); \*/

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int left=1,right=n;

while (left < right) {

int mid = left+(right-left)/2;

if (isBadVersion(mid)) {

right=mid;

} else {

left=mid+1;

}

}

return left;

}

}

### 162.寻找峰值

<https://leetcode.cn/problems/find-peak-element/submissions/644830779/?envType=problem-list-v2&envId=6uaxYMyj>

峰值元素是指其值严格大于左右相邻值的元素。

给你一个整数数组 nums，找到峰值元素并返回其索引。数组可能包含多个峰值，在这种情况下，返回 **任何一个峰值** 所在位置即可。

你可以假设 nums[-1] = nums[n] = -∞ 。

你必须实现时间复杂度为 O(log n)的算法来解决此问题。

**示例 1：**

**输入：**nums = [1,2,3,1]

**输出：**2

**解释：**3 是峰值元素，你的函数应该返回其索引 2。

**示例 2：**

**输入：**nums = [1,2,1,3,5,6,4]

**输出：**1 或 5

**解释：**你的函数可以返回索引 1，其峰值元素为 2；

  或者返回索引 5， 其峰值元素为 6。

class Solution {

public int findPeakElement(int[] nums) {

int left=0, right=nums.length-1;

while (left<right) {

int mid=left+(right-left)/2;

if (nums[mid] > nums[mid+1]) {

right=mid;

} else {

left=mid+1;

}

}

return left;

}

}

# 滑动窗口

## 固定长度滑动窗口

### 模板

left = 0

right = 0

while right < len(nums):

window.append(nums[right])

# 超过窗口大小时，缩小窗口，维护窗口中始终为 window\_size 的长度

if right - left + 1 >= window\_size:

# ... 维护答案

window.popleft()

left += 1

# 向右侧增大窗口

right += 1

### 题目

#### 1343. 大小为 K 且平均值大于等于阈值的子数组数目

<https://leetcode.cn/problems/number-of-sub-arrays-of-size-k-and-average-greater-than-or-equal-to-threshold/description/>

给你一个整数数组 arr 和两个整数 k 和 threshold 。

请你返回长度为 k 且平均值大于等于 threshold 的子数组数目。

**示例 1：**

**输入：**arr = [2,2,2,2,5,5,5,8], k = 3, threshold = 4

**输出：**3

**解释：**子数组 [2,5,5],[5,5,5] 和 [5,5,8] 的平均值分别为 4，5 和 6 。其他长度为 3 的子数组的平均值都小于 4 （threshold 的值)。

**示例 2：**

**输入：**arr = [11,13,17,23,29,31,7,5,2,3], k = 3, threshold = 5

**输出：**6

**解释：**前 6 个长度为 3 的子数组平均值都大于 5 。注意平均值不是整数。

class Solution {

public int numOfSubarrays(int[] arr, int k, int threshold) {

int left=0,right=0,window\_size=0;

int res = 0;

while (right < arr.length) {

window\_size += arr[right];

if (right-left+1 >= k) {

if (window\_size >= k\*threshold) {

res++;

}

window\_size -= arr[left];

left++;

}

right++;

}

return res;

}

}

## 不定长度滑动窗口

### 模板

left = 0

right = 0

while right < len(nums):

window.append(nums[right])

while 窗口需要缩小:

# ... 可维护答案

window.popleft()

left += 1

# 向右侧增大窗口

right += 1

### 题目

#### 3. 无重复字符的最长子串

<https://leetcode.cn/problems/longest-substring-without-repeating-characters/description/>

给定一个字符串 s ，请你找出其中不含有重复字符的 **最长 子串**的长度。

**示例 1:**

**输入:** s = "abcabcbb"

**输出:** 3

**解释:** 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

**示例 2:**

**输入:** s = "bbbbb"

**输出:** 1

**解释:** 因为无重复字符的最长子串是 "b"，所以其长度为 1。

**示例 3:**

**输入:** s = "pwwkew"

**输出:** 3

**解释:** 因为无重复字符的最长子串是 "wke"，所以其长度为 3。

  请注意，你的答案必须是 **子串** 的长度，"pwke" 是一个*子序列，*不是子串。

class Solution {

public int lengthOfLongestSubstring(String s) {

int left=0,right=0;

int res = 0;

Map<Character,Integer> map=new HashMap<>();

while (right<s.length()) {

char c = s.charAt(right);

right++;

map.put(c, map.getOrDefault(c, 0)+1);

while (map.get(c) > 1) {

char temp = s.charAt(left);

map.put(temp, map.get(temp)-1);

left++;

}

res = Math.max(res, right - left);

}

return res;

}

}

# 深度优先遍历（DFS）

# 广度优先遍历（BFS）

## 题目

### 279.完全平方数

<https://leetcode.cn/problems/perfect-squares/description/?envType=problem-list-v2&envId=6uaxYMyj>

题解：

<https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#%E8%A7%A3%E9%A2%98%E6%80%9D%E8%B7%AF>

1. 定义 visited为标记访问节点的 set 集合变量，避免重复计算。定义 queue为存放节点的队列。使用 count表示为树的最小深度，也就是和为 n 的完全平方数的最小数量。
2. 首先，我们将 n 标记为已访问，即 visited.add(n)。并将其加入队列 queue中，即 queue.append(n)。
3. 令 count加 11，表示最小深度加 11。然后依次将队列中的节点值取出。
4. 对于取出的节点值 value，遍历可能出现的平方数（即遍历 [1,value+1] 中的数）。
5. 每次从当前节点值减去一个平方数，并将减完的数加入队列。
   1. 如果此时的数等于 0，则满足题意，返回当前树的最小深度。
   2. 如果此时的数不等于 0，则将其加入队列，继续查找。

class Solution {

public int numSquares(int n) {

Queue<Integer> queue = new LinkedList<>();

Set<Integer> visited = new HashSet<>();

queue.offer(n);

visited.add(n);

int res = 0;

while (!queue.isEmpty()) {

int len = queue.size();

res++;

for (int i = 0; i < len; i++) {

int val = queue.poll();

for (int j = 1; j <= (int)Math.sqrt(val)+1; j++) {

int x = val - j\*j;

if (x == 0) {

return res;

}

if (!visited.contains(x)) {

queue.offer(x);

visited.add(x);

}

}

}

}

return res;

}

}

# 回溯算法

穷举式的搜索算法。

**Backtrace**

**走不通就回退**

一棵决策树

res = [] # 存放所欲符合条件结果的集合

path = [] # 存放当前符合条件的结果

def backtracking(nums): # nums 为选择元素列表

if 遇到边界条件: # 说明找到了一组符合条件的结果

res.append(path[:]) # 将当前符合条件的结果放入集合中

return

for i in range(len(nums)): # 枚举可选元素列表

path.append(nums[i]) # 选择元素

backtracking(nums) # 递归搜索

path.pop() # 撤销选择

backtracking(nums)

## 子集

参考：

<https://www.cnblogs.com/labuladong/p/15953806.html>

**输入：**nums = [1,2,3]

**输出：**[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

class Solution {

List<List<Integer>> result = new ArrayList<>();

LinkedList<Integer> path = new LinkedList<>();

public List<List<Integer>> subsets(int[] nums) {

backtrace(nums, 0);

return result;

}

private void backtrace(int[] nums, int startIndex) {

result.add(new ArrayList<>(path));

for (int i=startIndex;i<nums.length;i++) {

path.add(nums[i]);

backtrace(nums, i+1);

path.removeLast();

}

}

}

N皇后

class Solution {

List<List<String>> result = new ArrayList<>();

/\* 输入棋盘边长 n，返回所有合法的放置 \*/

public List<List<String>> solveNQueens(int n) {

// '.' 表示空，'Q' 表示皇后，初始化空棋盘

List<String> board = new ArrayList<>();

for (int i=0;i<n;i++) {

StringBuilder builder = new StringBuilder();

for (int j=0;j<n;j++) {

builder.append('.');

}

board.add(builder.toString());

}

backtrace(board, 0);

return result;

}

// 路径：board 中小于 row 的那些行都已经成功放置了皇后

// 选择列表：第 row 行的所有列都是放置皇后的选择

// 结束条件：row 超过 board 的最后一行

private void backtrace(List<String> board, int row) {

// 触发结束条件

if (row == board.size()) {

result.add(new ArrayList<>(board));

return;

}

int n = board.get(row).length();

for (int col=0;col<n;col++) {

// 排除不合法选择

if (!isValid(board, row, col)) {

continue;

}

// 做选择

StringBuilder builder = new StringBuilder(board.get(row));

builder.setCharAt(col, 'Q');

board.set(row, builder.toString());

// 进入下一行决策

backtrace(board, row+1);

// 撤销选择

builder.setCharAt(col, '.');

board.set(row, builder.toString());

}

}

/\* 是否可以在 board[row][col] 放置皇后？ \*/

/\* 路径：board 中小于 row 的那些行都已经成功放置了皇后 \*/

private boolean isValid(List<String> board, int row, int col) {

int n = board.size();

/\* 检查列是否有皇后互相冲突 \*/

for (int i=0;i<n;i++) {

if (board.get(i).charAt(col) == 'Q') {

return false;

}

}

/\* 检查右上方是否有皇后互相冲突 \*/

for (int i=row-1,j=col+1;i>=0 && j < n;i--,j++) {

if (board.get(i).charAt(j) == 'Q') {

return false;

}

}

/\* 检查左上方是否有皇后互相冲突 \*/

for (int i=row-1,j=col-1;i>=0 && j >=0;i--,j--) {

if (board.get(i).charAt(j) == 'Q') {

return false;

}

}

return true;

}

}

## 全排列

给定一个不含重复数字的数组 nums ，返回其 *所有可能的全排列* 。

**输入：**nums = [1,2,3]

**输出：**[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

import java.util.LinkedList;

import java.util.List;

class Solution {

List<List<Integer>> result = new LinkedList<>();

public List<List<Integer>> permute(int[] nums) {

LinkedList<Integer> paths = new LinkedList<>();

boolean[] visited = new boolean[nums.length];

backtrace(nums, paths, visited);

return result;

}

private void backtrace(int[] nums, LinkedList<Integer> paths, boolean[] visited) {

if (paths.size() == nums.length) {

result.add(new LinkedList<>(paths));

return;

}

for (int i = 0; i < nums.length; i++) {

if (visited[i]) {

continue;

}

paths.add(nums[i]);

visited[i] = true;

backtrace(nums, paths, visited);

paths.removeLast();

visited[i] = false;

}

}

}

给定一个可包含重复数字的序列 nums ，***按任意顺序*** 返回所有不重复的全排列。

**输入：**nums = [1,1,2]

**输出：**

[[1,1,2],

[1,2,1],

[2,1,1]]

class Solution {

List<List<Integer>> result = new ArrayList<>();

public List<List<Integer>> permuteUnique(int[] nums) {

boolean[] used = new boolean[nums.length];

LinkedList<Integer> path = new LinkedList<>();

Arrays.sort(nums);

backtrace(nums, path, used);

return result;

}

private void backtrace(int[] nums, LinkedList<Integer> path, boolean[] used) {

if (path.size() == nums.length) {

result.add(new ArrayList<>(path));

return;

}

for (int i=0;i<nums.length;i++) {

if (used[i]) {

continue;

}

// 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置

if (i > 0 && nums[i-1]==nums[i] && !used[i-1]) {

// 如果前面的相邻相等元素没有用过，则跳过

continue;

}

path.add(nums[i]);

used[i]=true;

backtrace(nums, path, used);

path.removeLast();

used[i]=false;

}

}

}

# 贪心算法

## 题目

### 162.寻找峰值

<https://leetcode.cn/problems/find-peak-element/description/?envType=problem-list-v2&envId=6uaxYMyj>

class Solution {

public int findPeakElement(int[] nums) {

int res =0;

for (int i=1;i<nums.length;i++) {

if (nums[i] > nums[res]) {

res = i;

}

}

return res;

}

}

# 动态规划

## 题目

### 279.完全平方数

<https://leetcode.cn/problems/perfect-squares/submissions/?envType=problem-list-v2&envId=6uaxYMyj>

思路：

<https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#%E6%80%9D%E8%B7%AF-2-%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92>

我们可以将这道题转换为「完全背包问题」中恰好装满背包的方案数问题。

1. 将 k=1,4,9,16,...看做是 k种物品，每种物品都可以无限次使用。
2. 将 n看做是背包的装载上限。
3. 这道题就变成了，从 k 种物品中选择一些物品，装入装载上限为 n的背包中，恰好装满背包最少需要多少件物品。

[**1. 划分阶段**](https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#_1-%E5%88%92%E5%88%86%E9%98%B6%E6%AE%B5)

按照当前背包的载重上限进行阶段划分。

[**2. 定义状态**](https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#_2-%E5%AE%9A%E4%B9%89%E7%8A%B6%E6%80%81)

定义状态 dp[w] 表示为：从完全平方数中挑选一些数，使其和恰好凑成 w，最少需要多少个完全平方数。

[**3. 状态转移方程**](https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#_3-%E7%8A%B6%E6%80%81%E8%BD%AC%E7%A7%BB%E6%96%B9%E7%A8%8B)

dp[w]=min{dp[w], dp[w−num]+1}

[**4. 初始条件**](https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#_4-%E5%88%9D%E5%A7%8B%E6%9D%A1%E4%BB%B6)

* 恰好凑成和为 0，最少需要 0 个完全平方数。
* 默认情况下，在不使用完全平方数时，都不能恰好凑成和为 w ，此时将状态值设置为一个极大值（比如 n+1），表示无法凑成。

[**5. 最终结果**](https://algo.itcharge.cn/solutions/0200-0299/perfect-squares/#_5-%E6%9C%80%E7%BB%88%E7%BB%93%E6%9E%9C)

根据我们之前定义的状态，dp[w]表示为：将物品装入装载上限为 w的背包中，恰好装满背包，最少需要多少件物品。 所以最终结果为 dp[n]。

1. 如果 dp[n]≠n+1，则说明：dp[n] 为装入装载上限为 n 的背包，恰好装满背包，最少需要的物品数量，则返回 dp[n]。
2. 如果 dp[n]=n+1，则说明：无法恰好装满背包，则返回 −1。因为 n 肯定能由 n 个 11 组成，所以这种情况并不会出现。

class Solution {

public int numSquares(int n) {

int[] dp = new int[n + 1];

dp[0] = 0;

for (int i = 1; i <= n; i++) {

dp[i] = i;//最坏的情况都是由1的平方组成

for (int j = 1; j \* j <= i; j++) {

//动态规划公式

dp[i] = Math.min(dp[i], dp[i - j \* j] + 1);

}

}

return dp[n];

}

}

**输入：**nums = [-2,1,-3,4,-1,2,1,-5,4]

**输出：**6

**解释：**连续子数组 [4,-1,2,1] 的和最大，为 6 。

用 f(i) 代表以第 i 个数结尾的「连续子数组的最大和」，那么很显然答案就是：

Max {f(i)} 0≤i≤n−1

因此，只需要求出每个位置的 f(i)，然后返回 f 数组中的最大值即可。

如何求 f(i) 呢？

可以考虑 nums[i] 单独成为一段还是加入 f(i−1) 对应的那一段，这取决于 nums[i] 和 f(i−1)+nums[i] 的大小，于是动态规划转移方程：

f(i)=max{f(i−1)+nums[i],nums[i]}

class Solution {

public int maxSubArray(int[] nums) {

int[] dp = new int[nums.length];

dp[0] = nums[0];

int res = dp[0];

for (int i=1;i<nums.length;i++) {

dp[i] += Math.max(dp[i-1]+nums[i], nums[i]);

res = Math.max(dp[i], res);

}

return res;

}

}

# 树

## 二叉树

### 遍历

# 合并区间

public static void main(String[] args) {  
 Map<Integer, Integer> map = new TreeMap<>();  
 map.put(0, 2);  
 map.put(4, 5);  
 map.put(3, 6);  
 map.put(8, 10);  
 System.out.println(map); // {0=2, 3=6, 4=5, 8=10}  
 System.out.println(combin(map)); // {0=6, 8=10}  
}  
static Map<Integer, Integer> combin(Map<Integer, Integer> map) {  
 Map<Integer, Integer> new\_map = new TreeMap<>();  
 if (map.size() == 0) {  
 return new\_map;  
 }  
 int begin = -1, end = -1;  
 for (Map.Entry<Integer, Integer> entry : map.entrySet()) {  
 int from = entry.getKey();  
 int to = entry.getValue();  
 if (begin == -1) {  
 // 第一个区间  
 begin = from;  
 end = to;  
 continue;  
 }  
 if (from <= end+1) {  
 int new\_end = Math.max(to, end);  
 new\_map.put(begin, new\_end);  
 end = new\_end;  
 continue;  
 }  
 new\_map.put(begin, end);  
 begin = from;  
 end = to;  
 }  
 new\_map.put(begin, end);  
 return new\_map;  
}

# todo

冯唐的散文《一万次的春和景明》