<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.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;  
}

# 双指针

## 模板

# 滑动窗口

## 固定长度滑动窗口

## 不定长度滑动窗口

# 深度优先遍历（DFS）

# 广度优先遍历（BFS）

# 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

# 动态规划

**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》