# 算法和算法分析【归纳】

注：详细的可见图形用户界面, 应用程序, Word

描述已自动生成

### 【x】 基本概念

#### 算法和数据结构

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

#### 算法的定义

参1：《数据结构-严蔚敏v2》



参2：<https://baike.baidu.com/item/%E7%AE%97%E6%B3%95/209025>

参3：<https://zh.wikipedia.org/wiki/%E7%AE%97%E6%B3%95>

参4：crash course

#### 算法的特性

文本

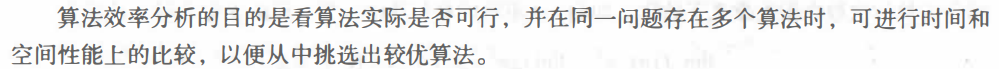
描述已自动生成

#### 评价算法优劣的基本标准

文本, 信件

描述已自动生成

#### 算法效率分析是什么



#### 衡量算法效率的方法【主要2种】

图片包含 文本

描述已自动生成

#### 问题规模

**问题规模**是算法求解问题输入量的多少，是问题大小的本质表示，一般用整数n表示。问题规模n对不同的问题含义不同，例如，在排序运算中n为参加排序的记录数，在矩阵运算中n为矩阵的阶数，在多项式运算中n为多项式的项数，在集合运算中n为集合中元素的个数，在树的有关运算中n为树的结点个数，在图的有关运算中n为图的顶点数或边数。

#### 算法和语句的执行时间

一个**算法的执行时间**大致上等于其所有语句执行时间的总和，而**语句的执行时间**则为该条语句的重复执行次数和执行一次所需时间的乘积。

#### 语句频度

一条语句的重复执行次数称作**语句的频度（Frequency Count）**

#### 基本语句/基本运算

最深层循环内的语句

#### 大O符号（Big O notation）的意义

<https://en.wikipedia.org/wiki/Big_O_notation>

<https://zh.wikipedia.org/zh-hans/%E5%A4%A7O%E7%AC%A6%E5%8F%B7>

又称为**渐进符号，**表数量级（Order of Magnitude）

### 【1】 时复

#### 《1》 是什么

一般情况下，算法中基本语句 重复执行的次数（即频度）是问题规模*n*的某个函数。算法的**渐近时间复杂度**（简称 **时间复杂度**）记为



#### 《2》 最好，最坏和平均 时复

**最坏时间复杂度** 是指在在最坏情况下，算法的时间复杂度，即算法计算量可能达到的最大值。

**平均时间复杂度** 是指所有可能输入实例在等概率出现的情况下，算法的期望运行时间。

**最好时间复杂度** 是指在最好情况下，算法的时间复杂度，即算法计算量可能达到的最小值。

一般总是考虑在最坏情况下的时间复杂度，以保证算法的运行时间不会比它更长。

#### 《x》 分析方法归纳

##### 非递归

###### 1》 元素

1. （嵌套）循环
2. 单个

时复为最内层循环语句（们）的时复

1. 多个并列

参考：享学-2021最新-数据结构VIP刷题指南.pdf

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

即max(O（loop1），。。。，O（loopn）)

1. API提供的排序（Arrays.sort, Collections.sort）

时复为排序的时复

* 1. Arrays.sort:

<https://leetcode-cn.com/problems/merge-intervals/solution/he-bing-qu-jian-by-leetcode-solution/>

###### 2》 组合

1. 若算法中不存在1）中元素/执行时间与问题规模无关，则时复为O（1）（可理解为整个算法为只执行一次的循环）
2. 否则，时复为 循环时复+api排序时复（一项没有就省略）

##### 递归

###### 方法

A） 通用

1. 通过递归状态树等方式算出函数节点总数的数量级O（f1（n））【f1（n）为函数节点总数】
2. 对递归函数按1）中方法求时复O（f2（n））【即每层递归的时复】
3. 得最终时复O（f1（n））\*O（f2（n））

B） 特例：回溯

1. 算出所有可能的结果f1（n）
2. 算出得出每个结果的时间f2（n）
3. 算出f（n）= f1（n）\*f2（n）
4. 时复T（n）=O（f（n））=



###### 例子

1. Fib

文本

描述已自动生成

图示

描述已自动生成

1. 反转链表

<https://leetcode-cn.com/problems/reverse-linked-list/solution/fan-zhuan-lian-biao-by-leetcode-solution-d1k2/>

1. 省份数量

<https://leetcode-cn.com/problems/number-of-provinces/solution/sheng-fen-shu-liang-by-leetcode-solution-eyk0/> dfs解法

##### 非递归+递归

若算法既有非递归，又有递归，则总时复=非递归时复+递归时复

#### 《x》 常见复杂度记忆

1. O（logn）

有序数列的二分查找

1. O（n）
2. 二叉树遍历：前序，中序，后序
3. 图的遍历
4. 搜索算法：DFS,BFS
5. 有序二维矩阵的二分查找
6. O（nlogn）

归并排序：merge sort

1. O（k^n）
2. Fibonacci递归解法：O（2^n）

文本

中度可信度描述已自动生成

#### 《x》 递归分析注意点

图形用户界面, 应用程序, Teams

描述已自动生成

递归时复是O（n）或O（k^n），永不为O^2

递归看每层调几次递归，蓝色的标志

描述已自动生成，调了k次就是O（k^n），调1次就是O（n）

上题最坏要调两次函数（

）=》O（2^n）

用这演示

图片包含 图形用户界面

描述已自动生成

### 【2】 空复

#### 是什么

<https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/9664257>

<https://zhuanlan.zhihu.com/p/50479555>

算法的 **渐近空间复杂度（**简称**空间复杂度）**是对该算法在运行过程中临时占用存储空间【一般为内存】大小的量度，它是问题规模*n*的函数。记为



其中g（n）为辅助空间关于n的函数。

#### 如何分析

##### 通用

<https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/9664257>

<https://zhuanlan.zhihu.com/p/50479555>

分析一个算法的空复，即看运行过程中算法内除输入和程序之外 使用到的所有量（常量，变量）新增的空间（又叫额外空间/辅助空间）。

##### 面试

和面试官说清楚，返回结果不算空复。

#### 原地工作/原地算法

<https://baike.baidu.com/item/%E5%8E%9F%E5%9C%B0%E7%AE%97%E6%B3%95>

原地算法包含使用O(1)[空间复杂度](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)的所有算法，即是一种使用固定数量/常量/常数大小【与输入大小/问题规模无关】的额外/辅助空间进行的。

#### 最好，最坏和平均 空复

https://leetcode-cn.com/problems/er-cha-shu-de-shen-du-lcof/solution/mian-shi-ti-55-i-er-cha-shu-de-shen-du-xian-xu-bia/

#### 分析方法归纳

##### 1） 非递归

###### 1》 方法

1. 元素

A》 数组

1. 单数组

空复=O（数组的长度）；

1. 多数组

空复=O（数组1长度）+。。。O（数组n长度）

=max（O（数组1长度）。。。O（数组n长度））

B》 集合框架

x》 API排序（Collections.sort, Arrays.sort）

A】 Arrays.sort

<https://leetcode-cn.com/problems/merge-intervals/solution/he-bing-qu-jian-by-leetcode-solution/>

1. 组合

若 无A）中元素/辅助空间大小与问题规模无关，空复=O（1）

否则，空复= A）中元素空复相加

###### 2》 例子

1. 遍历

单循环：

<https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/solution/mian-shi-ti-21-diao-zheng-shu-zu-shun-xu-shi-qi-4/>

https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/solution/jian-zhi-offer-21-diao-zheng-shu-zu-shun-77jc/

##### 2） 递归

###### 1》 方法

A）若在递归函数外无数组，空复=O（递归函数深度）\*按

1）中方法求得的递归函数空复

B） 若在递归函数外有数组，空复=max（数组空复，A）中空复）

###### 2》 例子

1. 爬楼梯

<https://leetcode-cn.com/problems/climbing-stairs/>

见：<https://u.geekbang.org/lesson/116?article=246041>

20:11

1. 省份数量

<https://leetcode-cn.com/problems/number-of-provinces/solution/sheng-fen-shu-liang-by-leetcode-solution-eyk0/> dfs解法

1. 二叉树的最大深度

<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/solution/er-cha-shu-de-zui-da-shen-du-by-leetcode-solution/> dfs解法

1. 二叉树的直径

<https://leetcode-cn.com/problems/diameter-of-binary-tree/solution/er-cha-shu-de-zhi-jing-by-leetcode-solution/>

##### 3） 非递归+递归

#### 《x》 常见复杂度记忆

1. 对树，递归深度为树的高度height

<https://leetcode-cn.com/problems/diameter-of-binary-tree/solution/er-cha-shu-de-zhi-jing-by-leetcode-solution/>

### 【x】 时复和空复关系

对于一个算法，其时间复杂度和空间复杂度往往是相互影响的，当追求一个较好的时间复杂度时，可能会导致占用较多的存储空间，即可能会使空间复杂度的性能变差，反之亦然。不过，通常情况下，鉴于运算空间较为充足，人们都以算法的时间复杂度作为算法优劣的衡量指标。

### 【3】 O的运算规则

* O（f（n））=O（xx），即化为标准数量级



即：按多项式将f（n）拆成单项，然后扫一眼看对应数量级最大的（即系数化为1）x，得出结果O（x）

推论：若f（n）为常数，O（f（n））=O（1）

* O（f1（n））+O（f2（n））=O（f1（n）+f2（n））=max（O（f1（n）），O（f2（n）））
* O（f1（n））\*O（f2（n））=O（f1（n）\*f2（n））

### 【x】 O的运算技巧

《1》 O（f（n））有时不需精确算出f（n），能看出数量级即可

### 【x】 复杂度数量级

#### 《1》 概述

常见的复杂度按数量级递增排列依次为：

常数/量阶<对数阶<线性阶< 线性对数阶<平方阶<立方阶<指数阶<阶乘阶

注1：所有的“阶”可叫“级别”

不同数量级的时间复杂度性状如图1.7所示。

图示

描述已自动生成

一般情况下，随着*n*的增大，*T*（*n*）的增长较慢的算法为较优的算法。显然，时间复杂度为指数阶的算法效率极低，当*n*值稍大时就无法应用。相比之下，应该尽可能选择用多项式阶的算法，而避免使用指数阶的算法。

#### 《2》 说明

##### 1） 多项式级别复杂度

<https://zh.wikipedia.org/wiki/%E5%A4%9A%E9%A0%85%E5%BC%8F%E6%99%82%E9%96%93>

<https://blog.csdn.net/yanghangjun/article/details/7298798>

文本, 信件

描述已自动生成

##### 2） 指数级别复杂度

##### 3） O（logn）底数是多少

<https://blog.csdn.net/jdbc/article/details/42173751>

文本

描述已自动生成



### 【4】 复杂度分析实例（待放到【1】和【2】中）

#### 树

##### 近遍历操作

###### 复杂度

时复：O（n）

空复：O（n）：递归深度

n是树节点数

###### 例题

二叉树的最近公共祖先-递归法

<https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-hou-xu/>

#### 回溯

##### 求共k种结果

###### 复杂度

时复：O（（结果个数k） \* （得到每个结果的递归深度））

空复：得到每个结果的递归深度

###### 例题

组合：

<https://leetcode-cn.com/problems/combinations/solution/zu-he-by-leetcode-solution/>

### 【5】 常用性能优化方案

#### 《1》耗时/执行用时

1. 位运算代替模，乘除法；
2. 数组代替hash table；
3. 数组代替List：access操作；

#### 《2》内存占用/消耗

#### 《3》时复

1. 升维

<https://u.geekbang.org/lesson/116?article=246042> 22:00

#### 《4》空复

# 排序

## 各算法

### 堆排序【二叉堆】

#### 《1》 是什么

见：《王道2021数据结构》

图片包含 文本

描述已自动生成

#### 《x》 实现

##### 参考：

###### 1） 大根堆

写法1：《王道数据结构2018》P297（index从1开始）

同参考：<https://www.cnblogs.com/Camilo/p/3904899.html>的删除操作

堆：

|  |
| --- |
| public class BigRootHeap<T extends Comparable<T>> {  void BuildMaxHeap(T A[], int len){  for(int i=len/2;i>0;i--){  AdjustDown(A,i,len); //从i=[n/2]~1,反复调整堆  }  }   void AdjustDown(T A[],int k,int len){  //函数AdjustDown将元素k向下进行调整  A[0]=A[k]; //A[0]暂存  for(int i=2\*k;i<=len;i\*=2){ //沿key较大的子节点向下筛选  if(i<len&&A[i].compareTo(A[i+1])<0){  i++; //取key较大的子节点下标  }  if(A[0].compareTo(A[i])>=0){  break; //筛选结束  }else{  A[k]=A[i]; //将A[i]调整到双亲节点上  k=i; //修改k值，以便继续向下筛选  }  }  A[k]=A[0];  }   void HeapSort(T A[],int len){  BuildMaxHeap(A,len); //初始建堆  for(int i=len;i>1;i--){ //n-1趟的交换和建堆  Swap(i,1,A); //输出堆顶元素（和堆底元素交换）  AdjustDown(A,1,i-1); //整理，把剩余的i-1个元素整理成堆  }  }   void Swap(int a,int b,T A[]){  T tmp=A[a];  A[a]=A[b];  A[b]=tmp;  }   void AdjustUp(T A[],int k){  //参数k为向上调整的节点，也为堆的元素个数  A[0]=A[k];  int i=k/2; //若节点值大于双亲节点，则将双亲节点向下调，并继续向上比较  while(i>0 && A[i].compareTo(A[0])<0){ //循环跳出条件  A[k]=A[i]; //双亲节点下调  k=i;  i=k/2; //继续向上比较  }  A[k]=A[0]; //复制到最终位置  }   void HeapDelete(T A[],int len){  //使用堆尾元素直接覆盖堆顶元素  A[1]=A[len];  //从堆顶到堆尾(此时堆中只有len-1个元素)进行堆调整。  AdjustDown(A,1,len-1);  } } |

测试：

|  |
| --- |
| public class Main {  private static final Integer[] *heap*={0,53,17,78,9,45,65,87,32};   public static void main(String[] args){  *Big*();   }   static void Big(){  BigRootHeap<Integer> bh=new BigRootHeap<>();   //建堆  int len=*heap*.length-1;  bh.BuildMaxHeap(*heap*,len);  System.*out*.print("建大根堆: ");  for(int i=1;i<=len;i++){  System.*out*.print(*heap*[i]+",");  }  System.*out*.println();   //删除  bh.HeapDelete(*heap*,len--);  System.*out*.print("删除堆顶元素后: ");  for(int i=1;i<=len;i++){  System.*out*.print(*heap*[i]+",");  }  System.*out*.println();   //堆排序  bh.HeapSort(*heap*,len);  System.*out*.print("堆排序结果: ");  for(int i=1;i<=len;i++){  System.*out*.print(*heap*[i]+",");  }  System.*out*.println();  } } |

运行结果：

手机屏幕的截图

描述已自动生成

写法2：覃超给的（index从0开始）

<https://shimo.im/docs/M2xfacKvwzAykhz6/read>

|  |
| --- |
| // Java  static void heapify(int[] array, int length, int i) {      int left = 2 \* i + 1, right = 2 \* i + 2；      int largest = i;      if (left < length && array[left] > array[largest]) {          largest = left;      }      if (right < length && array[right] > array[largest]) {          largest = right;      }      if (largest != i) {          int temp = array[i]; array[i] = array[largest]; array[largest] = temp;          heapify(array, length, largest);      }  }  public static void heapSort(int[] array) {      if (array.length == 0) return;      int length = array.length;      for (int i = length / 2-1; i >= 0; i--)          heapify(array, length, i);      for (int i = length - 1; i >= 0; i--) {          int temp = array[0]; array[0] = array[i]; array[i] = temp;          heapify(array, i, 0);      }  } |

###### 2） 小根堆

<https://www.cnblogs.com/Camilo/p/3904899.html>

##### 自实现：

###### 大根堆：升序

|  |
| --- |
| //大根堆,index从0开始 public class BigRoot {   private int[] heap;  private int heapSize;   public BigRoot(int[] heap) {  this.heap = Arrays.*copyOf*(heap, heap.length); //深拷贝  heapSize = heap.length;  }   //建堆  void buildHeap() {  for (int i = heapSize / 2 - 1; i >= 0; i--) {  heapifyDown(i, heapSize);  }  }   //堆排序  void heapSort() {  buildHeap(); //先建堆  for (int i = heapSize - 1; i > 0; i--) {  swap(heap, i, 0); //输出堆顶元素【和堆底元素交换】  heapifyDown(0, i); //将剩余的i个元素调整为堆  }  }   private void swap(int[] heap, int a, int b) {  int tmp = heap[a];  heap[a] = heap[b];  heap[b] = tmp;  }   //将元素heap[k]在大小为len的堆中向下调整【迭代写法】  void heapifyDown(int k, int len) {  int tmp = heap[k];  for (int i = 2 \* k + 1; i < len; i = i \* 2 + 1) { //沿较大的子节点向下调整  if (i + 1 < len && heap[i] < heap[i + 1]) {  i++; //取较大的子节点下标  }  if (tmp >= heap[i]) {  break; //调整结束  } else {  heap[k] = heap[i]; //将heap[i]调整到父节点上  k = i; //修改k值，以便继续向下调整  }  }  heap[k] = tmp;  }   //打印堆  void printHeap() {  for (int i = 0; i < heapSize; i++) {  if (i != heapSize - 1) {  System.*out*.print(heap[i] + ",");  } else {  System.*out*.print(heap[i]);  }  }  System.*out*.println();  } } |

###### 小根堆：降序

|  |
| --- |
| //小根堆,index从0开始 public class SmallRoot {   private int[] heap;  private int heapSize;   public SmallRoot(int[] heap) {  this.heap = Arrays.*copyOf*(heap, heap.length); //深拷贝  heapSize = heap.length;  }   //建堆  void buildHeap() {  for (int i = heapSize / 2 - 1; i >= 0; i--) {  heapifyDown(i, heapSize);  }  }   //堆排序  void heapSort() {  buildHeap(); //先建堆  for (int i = heapSize - 1; i > 0; i--) {  swap(heap, i, 0); //输出堆顶元素  heapifyDown(0, i); //将剩余的i个元素调整为堆  }  }   private void swap(int[] heap, int a, int b) {  int tmp = heap[a];  heap[a] = heap[b];  heap[b] = tmp;  }   //将元素heap[k]在大小为len的堆中向下调整【迭代写法】  void heapifyDown(int k, int len) {  int tmp = heap[k];  for (int i = 2 \* k + 1; i < len; i = i \* 2 + 1) { //沿较大的子节点向下调整  if (i + 1 < len && heap[i] > heap[i + 1]) {  i++; //取较小的子节点下标  }  if (tmp <= heap[i]) {  break; //调整结束  } else {  heap[k] = heap[i]; //将heap[i]调整到父节点上  k = i; //修改k值，以便继续向下调整  }  }  heap[k] = tmp;  }   //打印堆  void printHeap() {  for (int i = 0; i < heapSize; i++) {  if (i != heapSize - 1) {  System.*out*.print(heap[i] + ",");  } else {  System.*out*.print(heap[i]);  }  }  System.*out*.println();  } } |

###### Client：

|  |
| --- |
| public class Client {   private static final int[] *heap* = {53, 17, 78, 9, 45, 65, 87, 32};   public static void main(String[] args) {  *bigRoot*(); //大根堆  *smallRoot*(); //小根堆  }   private static void bigRoot() {  BigRoot bigRoot = new BigRoot(*heap*);   System.*out*.println("大根堆-排序前：");  bigRoot.printHeap();  //堆排序  bigRoot.heapSort();  System.*out*.println("大根堆-排序后：");  bigRoot.printHeap();  }   private static void smallRoot() {  SmallRoot smallRoot = new SmallRoot(*heap*);   System.*out*.println("小根堆-排序前：");  smallRoot.printHeap();  //堆排序  smallRoot.heapSort();  System.*out*.println("小根堆-排序后：");  smallRoot.printHeap();  } } |

运行结果：

文本

描述已自动生成

###### 一些总结：

A》 看了下大小根堆在堆排序中的实现差别点为下面选中的2处

文本

描述已自动生成

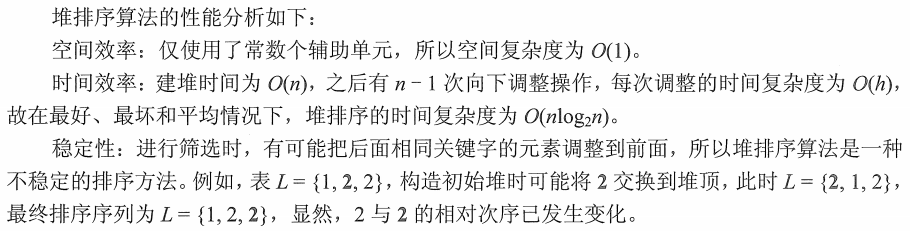
文本

描述已自动生成

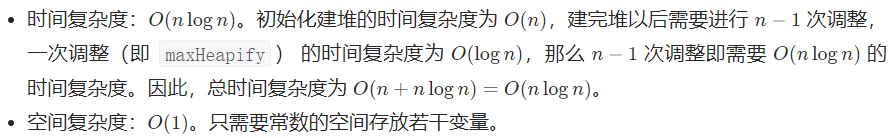
B》 大根堆排序结果为升序，小根堆排序结果为降序

#### 《x》 性能分析

参1：《王道2021数据结构》8.4.2



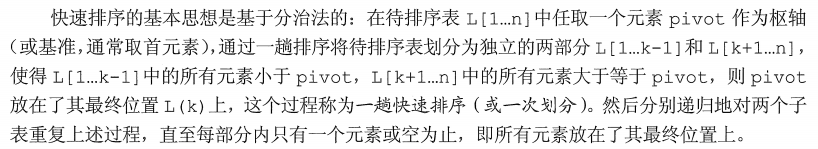
参2：<https://leetcode.cn/problems/sort-an-array/solution/pai-xu-shu-zu-by-leetcode-solution/>



### 快速排序

#### 是什么

参考：王道数据结构2021



#### 性质

参考：王道数据结构2021



#### 实现

##### 1） 递归

注：划分操作【即下面的partition方法】有很多写法，但作用相同

###### 写法1：王道数据结构2021+严蔚敏数据结构第二版

A） 原始

快排类：

|  |
| --- |
| public class QuickSort<ElemType extends Comparable<ElemType>> {   void quickSort(ElemType arr[], int low, int high) {  if (low < high) { //记住要加这个if判断  int pivotPos = partition(arr, low, high); //partition()是划分操作，将表arr[low...high]划分为满足条件的两个子表  quickSort(arr, low, pivotPos - 1); //依次对两个子表进行递归排序  quickSort(arr, pivotPos + 1, high);  }  }   int partition(ElemType arr[], int low, int high) {  ElemType pivot = arr[low]; //将当前表中第一个元素设为枢轴，对表进行划分  while (low < high) {  while (low < high && arr[high].compareTo(pivot) >= 0) {  --high;  }  arr[low] = arr[high]; //将比枢轴小的元素移到左侧  while (low < high && arr[low].compareTo(pivot) <= 0) {  ++low;  }  arr[high] = arr[low]; //将比枢轴大的元素移到右侧  }  arr[low] = pivot; //枢轴元素存放到最终位置  return low;  } } |

Client：

|  |
| --- |
| public class Client {   static final Integer[] *arr* = {3, 2, 3, 1, 2, 4, 5, 5, 6};   public static void main(String[] args) {  new QuickSort<Integer>().quickSort(*arr*, 0, *arr*.length - 1);  System.*out*.println("快排结果：" + Arrays.*asList*(*arr*));  } } |



注意：

第一点：

在partition方法中，

文本

描述已自动生成

这里的2个while循环，可以是以下三种中的任一：

* arr[high].compareTo(pivot)>=0 + arr[low].compareTo(pivot)<=0
* arr[high].compareTo(pivot)>0 + arr[low].compareTo(pivot)<=0
* arr[high].compareTo(pivot)>=0 + arr[low].compareTo(pivot)<0

但不能是

“arr[high].compareTo(pivot)>0 + arr[low].compareTo(pivot)<0”，这种会运行超时。

第二点：

必须是先右到左找、再左到右找，即先

文本

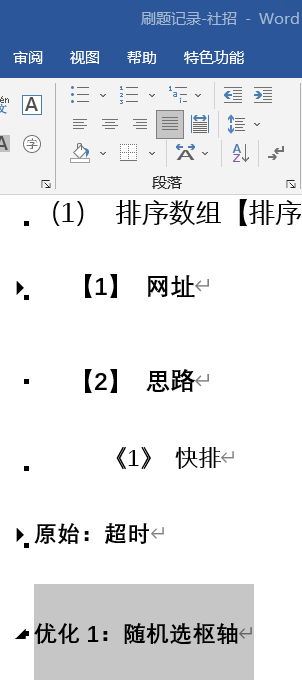
描述已自动生成、再

文本

描述已自动生成。

不能是先左到右找、再右到左找，这样会结果错误。

B） 优化1：随机选枢轴

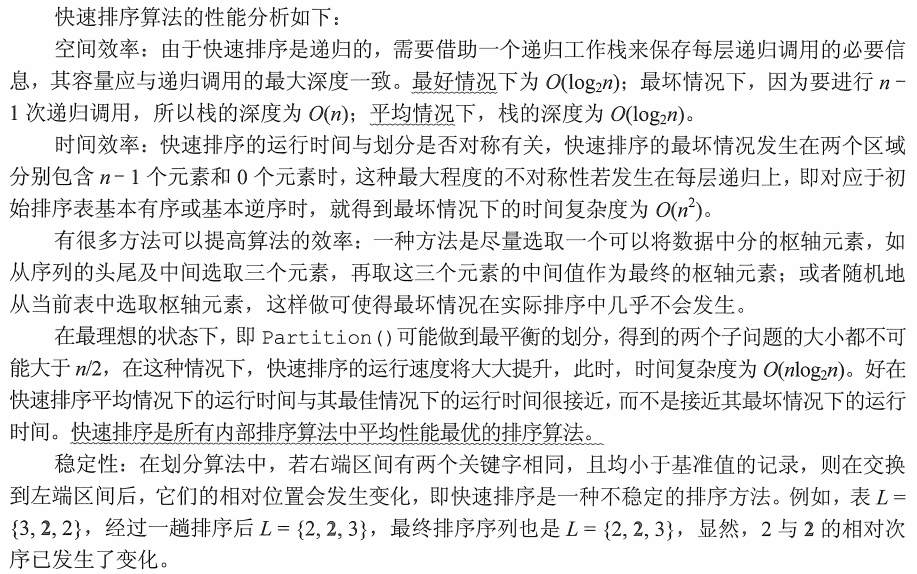
见

###### 写法2：覃超

<https://shimo.im/docs/TX9bDbSC7C0CR5XO/read>

#### 性能分析

参考：《王道数据结构2021》8.3.2



#### 《5》 衍生：快速选择<快选>

1. 和快排对比

快选不需要快排中的这个条件判断。

1. 相关题

<https://leetcode.cn/problems/kth-largest-element-in-an-array/>

### 【3】 (直接)插入排序

#### 《1》 是什么

<https://www.cnblogs.com/onepixel/p/7674659.html>

#### 《x》 实现

##### 不带哨兵：

<https://www.cnblogs.com/onepixel/p/7674659.html>

|  |
| --- |
| public class DirectInsertionSort {   //升序  void ascendingOrder(int[] arr) {  int tmp = 0;  int i, j;  for (i = 1; i < arr.length; i++) {  tmp = arr[i];  for (j = i - 1; j >= 0 && arr[j] > tmp; j--) {  arr[j + 1] = arr[j];  }  arr[j + 1] = tmp;  }  }   public static void main(String[] args) {  int[] arr = {49, 38, 65, 97, 76, 13, 27, 49};  //升序  new DirectInsertionSort().ascendingOrder(arr);  System.*out*.println(Arrays.*toString*(arr));  } } |



##### 带哨兵：暂不写

见

### 【4】 归并排序

#### 《1》 是什么

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

#### 《x》 实现【2路归并】

##### 递归

参考：

1. 《王道2021数据结构》
2. <https://shimo.im/docs/sDXxjjiKf3gLVVAU/read>

写法1：合并两个有序子数组用逻辑与

|  |
| --- |
| public class MergeSort {   //递归  void mergeSort(int[] arr, int left, int right) {  //写法1  /\*if (left < right) {  int mid = (left + right) >> 1; //等价于(left + right) / 2  //必须按arr[left,mid]和arr[mid+1,right]划分数组，否则会死递归  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }\*/   //写法2  if (left == right) {  return;  }  int mid = left + (right - left) / 2;  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }   //合并2个有序子数组  void merge(int[] arr, int left, int mid, int right) {  //1.临时数组  int tmp[] = new int[right - left + 1];  //2.合并  int i = left, j = mid + 1, k = 0;  //2-1.两个子表均未检测完  /\*for (; i <= mid && j <= r; k++) { //写法1  if (arr[i] < arr[j]) {  tmp[k] = arr[i++];  } else {  tmp[k] = arr[j++];  }  }\*/  while (i <= mid && j <= right) { //写法2  tmp[k++] = arr[i] < arr[j] ? arr[i++] : arr[j++];  }  //2-2.一个子表已检测完【下面2个while只有1个会执行】  while (i <= mid) {  tmp[k++] = arr[i++];  }  while (j <= right) {  tmp[k++] = arr[j++];  }  //3.赋给原数组  System.*arraycopy*(tmp, 0, arr, left, right - left + 1); //写法1  /\*for (int p = 0; p < tmp.length; p++) { //写法2  arr[left + p] = tmp[p];  }\*/  }   public static void main(String[] args) {  int[] arr = {49, 38, 65, 97, 76, 13, 27};  //二路归并+升序  new MergeSort().mergeSort(arr, 0, arr.length - 1);  System.*out*.println(Arrays.*toString*(arr));  } } |

写法2：合并两个有序子数组用逻辑或

|  |
| --- |
| class MergeSort {   //二路归并  void mergeSort(int[] arr, int left, int right) {  if (left < right) {  int mid = left + (right - left) / 2;  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }  }   //合并2个有序数组  private void merge(int[] arr, int left, int mid, int right) {  int[] tmp = new int[right - left + 1]; //临时数组  int i = left, j = mid + 1;  int k = 0;  while (i <= mid || j <= right) {  if (i > mid) {  tmp[k++] = arr[j++];  } else if (j > right) {  tmp[k++] = arr[i++];  } else {  tmp[k++] = arr[i] < arr[j] ? arr[i++] : arr[j++];  }  }  //tmp写回arr  System.*arraycopy*(tmp, 0, arr, left, right - left + 1);  }   public static void main(String[] args) {  int[] arr = {49, 38, 65, 97, 76, 13, 27};  //二路归并+升序  new MergeSort().mergeSort(arr, 0, arr.length - 1);  System.*out*.println(Arrays.*toString*(arr));  } } |

结果：



### 【5】 冒泡排序

#### 《1》 是什么

见：《王道2021数据结构》

#### 《x》 实现

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

|  |
| --- |
| public class BubbleSort {   void bubbleSort(int[] arr) {  for (int i = 0; i < arr.length - 1; i++) {  boolean isChanged = false;  for (int j = arr.length - 1; j > i; j--) {  if (arr[j - 1] > arr[j]) {  swap(arr, j - 1, j);  isChanged = true;  }  }  if (!isChanged) {  return;  }  }  }   void bubbleSort1(int[] arr) {  for (int i = 0; i < arr.length - 1; i++) {  boolean isChanged = false;  for (int j = 0; j < arr.length - 1 - i; j++) {  if (arr[j] > arr[j + 1]) {  swap(arr, j, j + 1);  isChanged = true;  }  }  if (!isChanged) {  return;  }  }  }   private void swap(int[] arr, int i, int j) {  int tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp;  }    public static void main(String[] args) {  //升序+内层循环从后往前  int[] arr = {49, 38, 65, 97, 76, 13, 27, 49};  new BubbleSort().bubbleSort(arr);  System.*out*.println(Arrays.*toString*(arr));  //升序+内存循环从前往后  int[] arr1 = {49, 38, 65, 97, 76, 13, 27, 49};  new BubbleSort().bubbleSort1(arr1);  System.*out*.println(Arrays.*toString*(arr1));  } } |



### 【6】 (简单)选择排序

#### 《1》 是什么

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

#### 《2》 实现

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

|  |
| --- |
| public class SelectionSort {   void selectionSort(int[] arr) {  for (int i = 0; i < arr.length - 1; i++) { //arr.length - 1趟即可  int minIdx = i;  for (int j = i + 1; j < arr.length; j++) {  if (arr[j] < arr[minIdx]) {  minIdx = j;  }  }  if (minIdx != i) {  swap(arr, minIdx, i);  }  }  }   private void swap(int[] arr, int i, int j) {  int tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp;  }   public static void main(String[] args) {  int[] arr = {49, 38, 65, 97, 76, 13, 27, 49};  //升序  new SelectionSort().selectionSort(arr);  System.*out*.println(Arrays.*toString*(arr));  } } |



### 【7】 希尔排序

#### 《1》 是什么

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

#### 《2》 实现

参考：

1. 《王道2021数据结构》
2. <https://www.cnblogs.com/onepixel/p/7674659.html>

|  |
| --- |
| public class ShellSort {   void shellSort(int arr[]) {  int len = arr.length;  for (int dk = len / 2; dk >= 1; dk /= 2) { //步长/增量变化  for (int i = dk; i < len; i++) { //对下标增量为dk的各子表进行直接插入排序  int current = arr[i];  int j = i;  while (j - dk >= 0 && current < arr[j - dk]) {  arr[j] = arr[j - dk];  j -= dk;  }  arr[j] = current;  }  }  }   public static void main(String[] args) {  int[] arr = {49, 38, 65, 97, 76, 13, 27, 49, 55, 4};  //升序  new ShellSort().shellSort(arr);  System.*out*.println(Arrays.*toString*(arr));  } } |



## 性能比较

需要结合下面2个参考看，因为对于每种排序算法其中一个参考可能会错。

### 参1：覃超课及给的资料

<https://u.geekbang.org/lesson/116?article=262138>

<https://www.cnblogs.com/onepixel/p/7674659.html>



上图快排的空复是错的，应为O（logn）：

<https://www.zhihu.com/question/51802358>

### 参2：王道2021数据结构

表格

描述已自动生成

## （3） 语言库

### 【1】 java

《1》 Arrays.sort()

<https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/solution/zui-xiao-de-kge-shu-by-leetcode-solution/>

时：*O*(*n*log*n*)

空：*O*(log*n*)

其中 *n* 是数组 arr 的长度。

# 动态规划：dynamic programming<dp>

参考：

1. 覃超课
2. 研究生凌应标课
3. <https://en.wikipedia.org/wiki/Dynamic_programming>
4. 最优化方法

<https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E5%8C%96%E6%96%B9%E6%B3%95/4019480>

1. 已有整理表格

   描述已自动生成
2. 递归公式和递推公式

<https://baike.baidu.com/item/%E9%80%92%E5%BD%92%E5%85%AC%E5%BC%8F/6644031>

<https://baike.baidu.com/item/%E9%80%92%E6%8E%A8%E5%85%AC%E5%BC%8F/3799173>

## 是什么

### 参1:<https://en.wikipedia.org/wiki/Dynamic_programming>

**Dynamic programming** is both a [mathematical optimization](https://en.wikipedia.org/wiki/Mathematical_optimization) method：数学最优化方法 and a computer programming method.

### 参2：覃超课

动态规划也叫动态递推。

一般DP是要你求一个最优解/最值【也可以不是，如fib、爬楼梯、路径计数问题】。

## 前提条件

注：以下两条件有至少其中一个就要想到用dp，很多题并非两条件均有

1. 重复/重叠子问题

递归算法求解问题时，每次产生的**子问题**【即代码相同，但问题规模更小，参考递归过程】并不总是新问题，有些子问题被反复计算多次。这种性质称为**子问题的重复性质**。

1. 最优子结构

要解决问题的最优解包含其子问题的最优解，这种性质称为**最优子结构性质**。

同一个问题可以有多种方式刻划它的最优子结构，有些表示方法的求解速度更快。

## 使用步骤

1. 定义状态空间/dp数组/dp状态表【即定义子问题】
2. 一维
3. 二维
   1. 横坐标是str1，纵坐标是str2
   2. 横纵坐标都是str
4. 用dp数组写出dp方程/状态转移方程/递推公式【即描述子问题之间的联系】
5. 初始化dp数组
6. 根据dp方程求解

根据dp方程自底向上地迭代计算dp数组，最终问题的答案有2种情况：

* 迭代完的dp[dp.length-1]即为解
* 迭代完的dp[dp.length-1]不为解，需对dp数组进一步处理得到解

1. 看能否优化空复

## 和递归的关系

本质上要解决的就是一个递归问题【即能用dp解决的问题必然能用递归解决】，但和普通的递归存在不同：

* 递归是自顶向下的；动规是以自底向上的方式递推地从子问题的（最优）解逐步构造出整个问题的（最优）解，同时对每个子问题只解一次，而后将其解保存在一个表格【也叫备忘录】中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
* “能用dp解决的问题必然能用递归解决”的具体意义

递归问题分为两类：

* 能用递归公式表示

这里的**递归公式**也叫**递推公式**，dp能解的问题就属于这类。一般这类问题有3种解法：傻递归、记忆化递归/带备忘录的递归、dp。

* 不能用递归公式表示

即不存在**递推公式**能表示这类问题。例如：汉诺塔问题、八皇后问题、树的遍历。

* 对于求解存在最优子结构的最优化问题，dp每步需淘汰次优解，只保留在该步中最优或较优的一些状态，来推导出最后的全局最优。

## 示例

### 非最优化问题

#### 《1》 斐波那契数列

<https://leetcode.cn/problems/fibonacci-number/>

#### 《2》 爬楼梯

<https://leetcode.cn/problems/climbing-stairs/>

#### 《3》 路径计数

<https://leetcode.cn/problems/unique-paths/>

<https://leetcode.cn/problems/unique-paths-ii/>

### 最优化问题

#### 《1》 矩阵连乘问题

<https://blog.csdn.net/qq_19782019/article/details/94356886>

#### 《2》 最长回文子串

<https://leetcode.cn/problems/longest-palindromic-substring/>

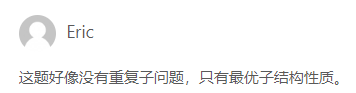
有重复子问题，有最优子结构

#### 《3》 最大子序和

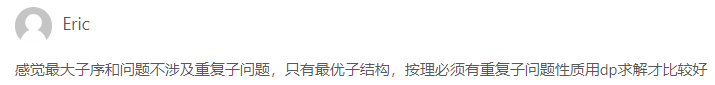
<https://leetcode.cn/problems/maximum-subarray/>

无重复子问题，有最优子结构

<https://leetcode.cn/problems/maximum-subarray/solution/by-bestqiang-q7lq/>



<https://leetcode.cn/problems/maximum-subarray/solution/dong-tai-gui-hua-fen-zhi-fa-python-dai-ma-java-dai/>



#### 《4》 买卖股票的最佳时机

<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>

无重复子问题，有最优子结构