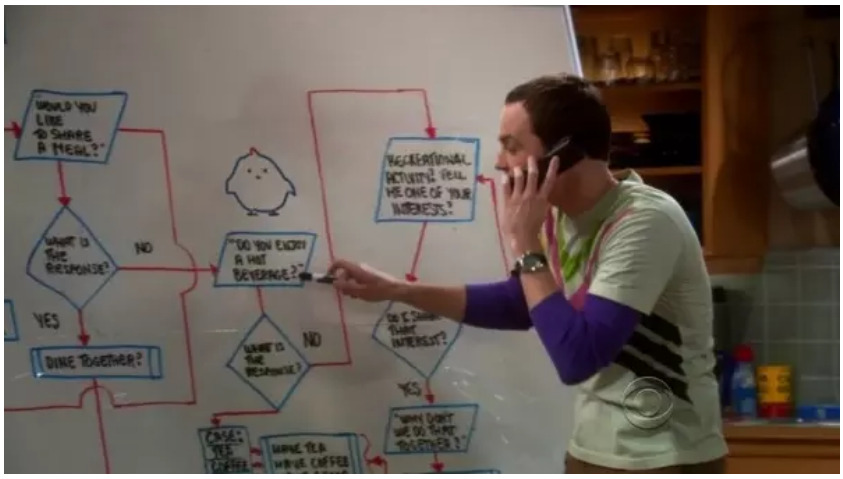
常见数据结构与算法整理总结（下）

尘语凡心
2016.10.03 16:28:46 字数 4,379

> 这篇文章是常见数据结构与算法整理总结的下篇，上一篇主要是对常见的数据结构进行集中总结，这篇主要是总结一些常见的算法相关内容，文章中如有错误，欢迎指出。

一、概述

以前看到这样一句话，语言只是工具，算法才是程序设计的灵魂。的确，算法在计算机科学中的地位真的很重要，在很多大公司的笔试面试中，算法掌握程度的考察都占据了很大一部分。不管是为了面试还是自身编程能力的提升，花时间去研究常见的算法还是很有必要的。下面是自己对于算法这部分的学习总结。



> 算法简介

算法是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。对于同一个问题的解决，可能会存在着不同的算法，为了衡量一个算法的优劣，提出了空间复杂度与时间复杂度这两个概念。

> 时间复杂度

一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数f(n)，算法的时间度量记为 ** T(n) = O(f(n)) **，它表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称时间复杂度。这里需要重点理解这个增长率。

```
举个例子，看下面3个代码：

1、{++x;}

```

```
2、for(i = 1; i <= n; i++) { ++x; }

3、for(j = 1; j <= n; j++)
        for(j = 1; j <= n; j++)
             { ++x; }

上述含有 ++x 操作的语句的频度分别为1 、n 、n^2，

假设问题的规模扩大了n倍，3个代码的增长率分别是1 、n 、n^2

它们的时间复杂度分别为O(1)、O(n )、O(n^2)
```

> 空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度，记做S(n)=O(f(n))。一个算法的优劣主要从算法的执行时间和所需要占用的存储空间两个方面衡量。

二、查找算法

查找和排序是最基础也是最重要的两类算法，熟练地掌握这两类算法，并能对这些算法的性能进行分析很重要，这两类算法中主要包括二分查找、快速排序、归并排序等等。

> 顺序查找

顺序查找又称线性查找。它的过程为：从查找表的最后一个元素开始逐个与给定关键字比较，若某个记录的关键字和给定值比较相等，则查找成功，否则，若直至第一个记录，其关键字和给定值比较都不等，则表明表中没有所查记录查找不成功，它的缺点是效率低下。

> 二分查找

- 简介

二分查找又称折半查找，对于有序表来说，它的优点是比较次数少，查找速度快，平均性能好。

二分查找的基本思想是将n个元素分成大致相等的两部分，取a[n/2]与x做比较，如果x=a[n/2]，则找到x，算法中止；如果x<a[n/2]，则只要在数组a的左半部分继续搜索x，如果x>a[n/2]，则只要在数组a的右半部搜索x。

二分查找的时间复杂度为O(logn)

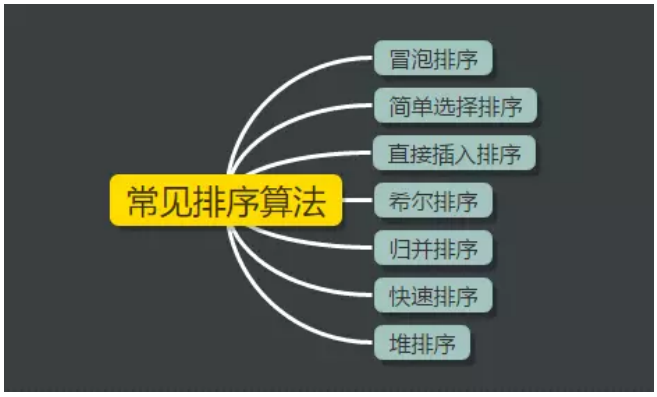
- 实现

```
//给定有序查找表array 二分查找给定的值data
//查找成功返回下标 查找失败返回-1

static int funBinSearch(int[] array, int data) {

    int low = 0;
    int high = array.length - 1;

    while (low <= high) {

        int mid = (low + high) / 2;

        if (data == array[mid]) {
            return mid;
        } else if (data < array[mid]) {
            high = mid - 1;
        } else {
            low = mid + 1;
```

```
        }
    }
    return -1;
}
```

三、排序算法

排序是计算机程序设计中的一种重要操作，它的功能是将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。下面主要对一些常见的排序算法做介绍，并分析它们的时空复杂度。



常见排序算法

常见排序算法性能比较：

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

图片来自网络

上面这张表中有稳定性这一项，排序的稳定性是指如果在排序的序列中，存在前后相同的两个元素的话，排序前和排序后他们的相对位置不发生变化。

下面从冒泡排序开始逐一介绍。

> 冒泡排序

- 简介

冒泡排序的基本思想是：设排序序列的记录个数为n，进行n-1次遍历，每次遍历从开始位置依次往后比较前后相邻元素，这样较大的元素往后移，n-1次遍历结束后，序列有序。

例如，对序列(3,2,1,5)进行排序的过程是：共进行3次遍历，第1次遍历时先比较3和2，交换，继续比较3和1,交换，再比较3和5，不交换，这样第1次遍历结束，最大值5在最后的位置，得

到序列(2,1,3,5)。第2次遍历时先比较2和1，交换，继续比较2和3，不交换，第2次遍历结束时次大值3在倒数第2的位置，得到序列(1,2,3,5)，第3次遍历时，先比较1和2，不交换，得到最终有序序列(1,2,3,5)。

需要注意的是，如果在某次遍历中没有发生交换，那么就不必进行下次遍历，因为序列已经有序。

- 实现

```
// 冒泡排序 注意 flag 的作用
static void funBubbleSort(int[] array) {

    boolean flag = true;

    for (int i = 0; i < array.length - 1 && flag; i++) {

        flag = false;

        for (int j = 0; j < array.length - 1 - i; j++) {

            if (array[j] > array[j + 1]) {

                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;

                flag = true;
            }
        }
    }

    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```

- 分析

最佳情况下冒泡排序只需一次遍历就能确定数组已经排好序，不需要进行下一次遍历，所以最佳情况下，时间复杂度为** O(n) **。

最坏情况下冒泡排序需要n-1次遍历，第一次遍历需要比较n-1次，第二次遍历需要n-2次，...，最后一次需要比较1次，最差情况下时间复杂度为** O(n^2) **。

> 简单选择排序

- 简介

简单选择排序的思想是：设排序序列的记录个数为n，进行n-1次选择，每次在n-i+1(i = 1,2,...,n-1)个记录中选择关键字最小的记录作为有效序列中的第i个记录。

例如，排序序列(3,2,1,5)的过程是，进行3次选择，第1次选择在4个记录中选择最小的值为1，放在第1个位置，得到序列(1,3,2,5)，第2次选择从位置1开始的3个元素中选择最小的值2放在第2个位置，得到有序序列(1,2,3,5)，第3次选择因为最小的值3已经在第3个位置不需要操作，最后得到有序序列（1,2,3,5）。

- 实现

```
static void funSelectionSort(int[] array) {

    for (int i = 0; i < array.length - 1; i++) {

        int mink = i;

            // 每次从未排序数组中找到最小值的坐标
```

```
        for (int j = i + 1; j < array.length; j++) {

            if (array[j] < array[mink]) {
                mink = j;
            }
        }

        // 将最小值放在最前面
        if (mink != i) {
            int temp = array[mink];
            array[mink] = array[i];
            array[i] = temp;
        }
    }

    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
```

- 分析

简单选择排序过程中需要进行的比较次数与初始状态下待排序的记录序列的排列情况** 无关。当i=1时，需进行n-1次比较；当i=2时，需进行n-2次比较；依次类推，共需要进行的比较次数是(n-1)+(n-2)+…+2+1=n(n-1)/2，即进行比较操作的时间复杂度为 O(n^2) ，进行移动操作的时间复杂度为 O(n) 。总的时间复杂度为 O(n^2) **。

最好情况下，即待排序记录初始状态就已经是正序排列了，则不需要移动记录。最坏情况下，即待排序记录初始状态是按第一条记录最大，之后的记录从小到大顺序排列，则需要移动记录的次数最多为3（n-1）。

简单选择排序是不稳定排序。

> 直接插入排序

- 简介

直接插入的思想是：是将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增1的有序表。

例如，排序序列(3,2,1,5)的过程是，初始时有序序列为(3)，然后从位置1开始，先访问到2，将2插入到3前面，得到有序序列(2,3)，之后访问1,找到合适的插入位置后得到有序序列(1,2,3)，最后访问5，得到最终有序序列(1,2,3,5).

- 实现

```
static void funDInsertSort(int[] array) {

    int j;

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

        int temp = array[i];

        j = i - 1;

        while (j > -1 && temp < array[j]) {

            array[j + 1] = array[j];

            j--;
        }

        array[j + 1] = temp;

    }

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

```
        System.out.print(array[i] + " ");
    }
}
```

- 分析

最好情况下，当待排序序列中记录已经有序时，则需要n-1次比较，不需要移动，时间复杂度为** O(n) 。最差情况下，当待排序序列中所有记录正好逆序时，则比较次数和移动次数都达到最大值，时间复杂度为 O(n^2) 。平均情况下，时间复杂度为 O(n^2) **。

> 希尔排序

希尔排序又称"缩小增量排序"，它是基于直接插入排序的以下两点性质而提出的一种改进：(1) 直接插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率。(2) 直接插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。点击查看更多关于希尔排序的内容

> 归并排序

- 简介

归并排序是分治法的一个典型应用，它的主要思想是：将待排序序列分为两部分，对每部分递归地应用归并排序，在两部分都排好序后进行合并。

例如，排序序列(3,2,8,6,7,9,1,5)的过程是，先将序列分为两部分，(3,2,8,6)和(7,9,1,5)，然后对两部分分别应用归并排序，第1部分(3,2,8,6)，第2部分(7,9,1,5)，对两个部分分别进行归并排序，第1部分继续分为(3,2)和(8,6)，(3,2)继续分为(3)和(2)，(8,6)继续分为(8)和(6)，之后进行合并得到(2,3)，(6,8)，再合并得到(2,3,6,8)，第2部分进行归并排序得到(1,5,7,9)，最后合并两部分得到(1,2,3,5,6,7,8,9)。

- 实现

```
    //归并排序
    static void funMergeSort(int[] array) {

        if (array.length > 1) {

            int length1 = array.length / 2;
            int[] array1 = new int[length1];
            System.arraycopy(array, 0, array1, 0, length1);
            funMergeSort(array1);

            int length2 = array.length - length1;
            int[] array2 = new int[length2];
            System.arraycopy(array, length1, array2, 0, length2);
            funMergeSort(array2);

            int[] datas = merge(array1, array2);
            System.arraycopy(datas, 0, array, 0, array.length);
        }

    }

    //合并两个数组
    static int[] merge(int[] list1, int[] list2) {

        int[] list3 = new int[list1.length + list2.length];

        int count1 = 0;
        int count2 = 0;
        int count3 = 0;

        while (count1 < list1.length && count2 < list2.length) {

```

```
            if (list1[count1] < list2[count2]) {
                list3[count3++] = list1[count1++];
            } else {
                list3[count3++] = list2[count2++];
            }
        }

        while (count1 < list1.length) {
            list3[count3++] = list1[count1++];
        }

        while (count2 < list2.length) {
            list3[count3++] = list2[count2++];
        }

        return list3;
    }
```

- 分析

归并排序的时间复杂度为O(nlogn)，它是一种稳定的排序，java.util.Arrays类中的sort方法就是使用归并排序的变体来实现的。

> 快速排序

- 简介

快速排序的主要思想是：在待排序的序列中选择一个称为主元的元素，将数组分为两部分，使得第一部分中的所有元素都小于或等于主元，而第二部分中的所有元素都大于主元，然后对两部分递归地应用快速排序算法。

- 实现

```
// 快速排序
static void funQuickSort(int[] mdata, int start, int end) {
    if (end > start) {
        int pivotIndex = quickSortPartition(mdata, start, end);
        funQuickSort(mdata, start, pivotIndex - 1);
        funQuickSort(mdata, pivotIndex + 1, end);
    }
}

// 快速排序前的划分
static int quickSortPartition(int[] list, int first, int last) {

    int pivot = list[first];
    int low = first + 1;
    int high = last;

    while (high > low) {

        while (low <= high && list[low] <= pivot) {
            low++;
        }

        while (low <= high && list[high] > pivot) {
            high--;
        }

        if (high > low) {
            int temp = list[high];
            list[high] = list[low];
            list[low] = temp;
        }
    }

    while (high > first && list[high] >= pivot) {
        high--;
    }

    if (pivot > list[high]) {
        list[first] = list[high];
        list[high] = pivot;
```

```
        return high;
    } else {
        return first;
    }
}
```

- 分析

在快速排序算法中，比较关键的一个部分是主元的选择。在最差情况下，划分由n个元素构成的数组需要进行n次比较和n次移动，因此划分需要的时间是O(n)。在最差情况下，每次主元会将数组划分为一个大的子数组和一个空数组，这个大的子数组的规模是在上次划分的子数组的规模上减1，这样在最差情况下算法需要(n-1)+(n-2)+...+1= ** O(n^2) **时间。

最佳情况下，每次主元将数组划分为规模大致相等的两部分，时间复杂度为** O(nlogn) **。

> 堆排序

- 简介

在介绍堆排序之前首先需要了解堆的定义，n个关键字序列K1，K2，...，Kn称为堆，当且仅当该序列满足如下性质（简称为堆性质）：(1) ki <= k(2i）且 ki <= k(2i+1) (1 ≤ i≤ n/2），当然，这是小根堆，大根堆则换成> =号。

如果将上面满足堆性质的序列看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有的非终端节点的值均不大于（或不小于）其左右孩子节点的值。

堆排序的主要思想是：给定一个待排序序列，首先经过一次调整，将序列构建成一个大顶堆，此时第一个元素是最大的元素，将其和序列的最后一个元素交换，然后对前n-1个元素调整为大顶堆，再将其第一个元素和末尾元素交换，这样最后即可得到有序序列。

- 实现

```
//堆排序
public class TestHeapSort {

    public static void main(String[] args) {
        int arr[] = { 5, 6, 1, 0, 2, 9 };
        heapsort(arr, 6);
        System.out.println(Arrays.toString(arr));
    }

    static void heapsort(int arr[], int n) {

        // 先建大顶堆
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapAdjust(arr, i, n);
        }

        for (int i = 0; i < n - 1; i++) {
            swap(arr, 0, n - i - 1);
            heapAdjust(arr, 0, n - i - 1);
        }
    }

    // 交换两个数
    static void swap(int arr[], int low, int high) {
        int temp = arr[low];
        arr[low] = arr[high];
        arr[high] = temp;
    }

    // 调整堆
    static void heapAdjust(int arr[], int index, int n) {

        int temp = arr[index];

        int child = 0;

```

```
        while (index * 2 + 1 < n) {

            child = index * 2 + 1;

            // child为左右孩子中较大的那个
            if (child != n - 1 && arr[child] < arr[child + 1]) {
                child++;
            }
            // 如果指定节点大于较大的孩子 不需要调整
            if (temp > arr[child]) {
                break;
            } else {
                // 否则继续往下判断孩子的孩子 直到找到合适的位置
                arr[index] = arr[child];
                index = child;
            }
        }

        arr[index] = temp;
    }
}

```

- 分析

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。堆排序时间复杂度也为O(nlogn)，空间复杂度为O(1)。它是不稳定的排序方法。与快排和归并排序相比，堆排序在最差情况下的时间复杂度优于快排，空间效率高于归并排序。

四、其它算法

在上面的篇幅中，主要是对查找和常见的几种排序算法作了介绍，这些内容都是基础的但是必须掌握的内容，尤其是二分查找、快排、堆排、归并排序这几个更是面试高频考察点。（这里不禁想起百度一面的时候让我写二分查找和堆排序，二分查找还行，然而堆排序当时一脸懵逼...）下面主要是介绍一些常见的其它算法。

> 递归

- 简介

在平常解决一些编程或者做一些算法题的时候，经常会用到递归。程序调用自身的编程技巧称为递归。它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。上面介绍的快速排序和归并排序都用到了递归的思想。

- 经典例子

斐波那契数列，又称黄金分割数列、因数学家列昂纳多·斐波那契以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……在数学上，斐波纳契数列以如下被以递归的方法定义：F（0）=0，F（1）=1，F（n）=F(n-1)+F(n-2)（n≥2，n∈N*）。

```
//斐波那契数列 递归实现
static long funFib(long index) {

    if (index == 0) {
        return 0;
    } else if (index == 1) {
        return 1;
    } else {
        return funFib(index - 1) + funFib(index - 2);
    }
}
```

上面代码是斐波那契数列的递归实现，然而我们不难得到它的时间复杂度是O(2^n)，递归有时候可以很方便地解决一些问题，但是它也会带来一些效率上的问题。下面的代码是求斐波那契数列的另一种方式，效率比递归方法的效率高。

```
static long funFib2(long index) {

    long f0 = 0;
    long f1 = 1;
    long f2 = 1;

    if (index == 0) {
        return f0;
    } else if (index == 1) {
        return f1;
    } else if (index == 2) {
        return f2;
    }

    for (int i = 3; i <= index; i++) {
        f0 = f1;
        f1 = f2;
        f2 = f0 + f1;
    }

    return f2;
}
```

> 分治算法

分治算法的思想是将待解决的问题分解为几个规模较小但类似于原问题的子问题，递归地求解这些子问题，然后合并这些子问题的解来建立最终的解。分治算法中关键地一步其实就是递归地求解子问题。关于分治算法的一个典型例子就是上面介绍的归并排序。查看更多关于分治算法的内容

> 动态规划

动态规划与分治方法相似，都是通过组合子问题的解来求解待解决的问题。但是，分治算法将问题划分为互不相交的子问题，递归地求解子问题，再将它们的解组合起来，而动态规划应用于子问题重叠的情况，即不同的子问题具有公共的子子问题。动态规划方法通常用来求解最优化问题。查看更多关于动态规划的内容

动态规划典型的一个例子是最长公共子序列问题。

常见的算法还有很多，比如贪心算法，回溯算法等等，这里都不再详细介绍，想要熟练掌握，还是要靠刷题，刷题，刷题，然后总结。

五、常见算法题

下面是一些常见的算法题汇总。

> 不使用临时变量交换两个数

```
static void funSwapTwo(int a, int b) {

    a = a ^ b;
    b = b ^ a;
    a = a ^ b;

    System.out.println(a + " " + b);
}
```

> 判断一个数是否为素数

```
static boolean funIsPrime(int m) {

    boolean flag = true;

    if (m == 1) {
        flag = false;
    } else {

        for (int i = 2; i <= Math.sqrt(m); i++) {
            if (m % i == 0) {
                flag = false;
                break;
            }
        }
    }

    return flag;
}
```

> 其它算法题

1、15道使用频率极高的基础算法题
2、二叉树相关算法题
3、链表相关算法题
4、字符串相关算法问题

六、总结

以上就是自己对常见的算法相关内容的总结，算法虐我千百遍，我待算法如初恋，革命尚未成功，同志仍需刷题，加油。