计算机算法基础总结

Jerry4me 人工智能与大数据技术 2018-12-26

> **作者：Jerry4me**
> **链接：https://www.jianshu.com/p/f6e35db6bc51**

> 我的Github地址：https://github.com/Jerry4me
> demo：https://github.com/Jerry4me/JRBaseAlgorithm

本文主要是通过通俗易懂的算法和自然语言，向大家介绍基础的计算机排序算法和查找算法，还有一些作为一名程序猿应该知道的名词，数据结构，算法等等. 但是仅仅止于介绍，因为本人能力不足，对一些高级的算法和数据结构理解不够通透，所以也不作太多的深入的剖析.. demo都在我的Github中能找得到。

同样的，通过最近面试实习生的机会，把一些基础都捡起来，巩固巩固，同时如果能帮助到大家，那也是极好的. 废话不多说，入正题吧。

排序算法

算法	最优复杂度	最差复杂度	平均复杂度	稳定性
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	不稳定
冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	稳定
插入排序	O(n)	$O(n^2)$	$O(n^2)$	稳定
希尔排序	O(n)	$O(n^2)$	O(n1.3)	不稳定
归并排序	O(nlog n)	O(nlog n)	O(nlog n)	稳定
快速排序	O(nlog n)	$O(n^2)$	O(nlog n)	不稳定
堆排序	O(nlog n)	O(nlog n)	O(nlog n)	不稳定
基数排序	O(d(r+n))	O(d(n+rd))	O(d(r+n))	稳定

ps：基数排序的复杂度中，r代表关键字的基数， d代表位数， n代表关键字的个数. 也就是说， 基数排序不受待排序列规模的影响。

算法复杂度：这里表中指的是算法的时间复杂度， 一般由O(1), O(n), O(logn), O(nlogn), O(n^2), ..., O(n!). 从左到右复杂度依次增大, 时间复杂度是指在多少时间内能够执行完这个算法， 常数时间内呢, 还是平方时间还是指数时间等等。

还有个概念叫空间复杂度, 这就指的是执行这个算法需要多少额外的空间。 (源数组/链表所占的空间不算)

稳定性：算法的稳定性体现在执行算法之前，若a = b, a在b的前面，若算法执行完之后a依然在b的前面, 则这个算法是稳定的， 否则这个算法不稳定

选择排序

原理：每次从无序区中找出最小的元素, 跟无序区的第一个元素交换

```
void selectSort(int array[], int count){

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

        // 最小元素的位置
        int index = i;
        // 找出最小的元素所在的位置
        for (int j = i + 1; j < count; j++) {

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

        }
        // 交换元素
        int temp = array[index];
        array[index] = array[i];
        array[i] = temp;

    }
}
```

冒泡排序

原理：每次对比相邻两项的元素的大小，不符合顺序则交换

```
void bubblingSort(int array[], int count){


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

        // 交换相邻的元素
```

```
        for (int j = 1; j < count - i; j++) {

            if (array[j] < array[j-1]){
                // 交换元素
                int temp = array[j-1];
                array[j-1] = array[j];
                array[j] = temp;
            }

        }
    }
}
```

插入排序

原理：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。

```
void insertSort(int array[], int count){

    int i, j, k;

    for (i = 1; i < count ; i++) {

        for (j = i - 1; j >= 0; j--) { // 为a[i]在a[0, i-1]上找一个合适的位置
            if(array[j] < array[i]) break;
        }

        if (j != i-1) { // 找到了一个合适的位置j

            int temp = array[i];
            // 将比array[i]大的数据全部往后移
            for(k = i - 1; k > j; k--) {
                array[k+1] = array[k];
            }
            // 将array[i]放入合适的位置
            array[k+1] = temp;

        }

    }
}
```

希尔排序

其实就是分组插入排序，也称为缩小增量排序. 比普通的插入排序拥有更高的性能.

算法思想：根据增量dk将整个序列分割成若干个子序列. 如dk = 3, 序列1, 7, 12, 5, 13, 22 就被分割成1, 5, 7, 13和12, 22, 在这几个子序列中分别进行直接插入排序, 然后依次缩减增量dk再进行排序,

直到序列中的元素基本有序时，再对全体元素进行一次直接插入排序. (直接插入排序在元素基本有序的情况下效率很高)

```
void shellSort(int array[], int count){

    for (int dk = count / 2; dk > 0; dk = dk / 2) { // dk增量

        for (int i = 0; i < dk; i++) { // 直接插入排序

            for (int j = i + dk; j < count; j += dk) {

                if (array[j] < array[j - dk]) { // 如果相邻的两个元素，后者比前者大，则不用调整

                    int temp = array[j];
                    int k = j - dk;
                    while (k >= 0 && array[k] > temp) { // 每次while循环结束后，保证把最小的插入到每

                        array[k + dk] = array[k];
                        k -= dk;

                    }
                    // 每组第一个元素为最小的元素
                    array[k + dk] = temp;

                }

            }

        }

    }

}
```

归并排序

原理：归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换),然后把各个有序的段序列并成一个有序的长序列，不断合并直到原序列全部排好序。

```
void merge(int array[], int temp[], int start, int middle, int end) {
    // 将两个有序序列array[start, middle]和array[middle+1, end]进行合并

    int i = start, m = middle, j = middle + 1, n = end, k = 0;

    while(i <= m && j <= n) { // 哪个小就先插那个，然后把temp下标和array插入位置的下标++

        if (array[i] <= array[j]) {
            temp[k++] = array[i++];
        } else {
            temp[k++] = array[j++];
        }
```

```
    }
    // 插完之后看谁没插完就继续插谁
    while(i <= m) temp[k++] = array[i++];
    while(j <= n) temp[k++] = array[j++];

    // 把temp的元素copy回array中
    for (int i = 0; i < k; i++) array[start + i] = temp[i];



}

void mSort(int array[], int temp[], int start, int end) {

    if (start < end) {
        int middle = (start + end) / 2;
        mSort(array, temp, start, middle); // 递归出来以后左边有序
        mSort(array, temp, middle + 1, end); // 右边有序
        merge(array, temp, start, middle, end); // 合并两个有序序列
    }

}

void mergeSort(int array[], int count){

    // 定义辅助数组
    int *temp = (int *)malloc(sizeof(array[0]) * count);
    // 开始进行归并排序
    mSort(array, temp, 0, count - 1);
    // 释放指针
    free(temp);

}
```

堆排序

- **二叉堆的定义**
  - **二叉堆是完全二叉树或者是近似完全二叉树。**
- **二叉堆满足二个特性：**
  - 父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值。
  - 每个结点的左子树和右子树都是一个二叉堆（都是最大堆或最小堆）。

> **大顶堆：**父结点的键值总是大于或等于任何一个子节点的键值
> **小顶堆：**父结点的键值总是小于或等于任何一个子节点的键值
> **算法思想：**堆排序 = 构造堆 + 交换堆末尾元素与根结点 + 删除末尾结点 + 构造堆 + 交换....依次循环，由于根结点必定是堆中最大(最小)的元素，所以删除出来的元素序列也必定是升序(降序)的.

```
void minHeapFixdown(int array[], int i, int count) {

    int j, temp;

    temp = array[i];
    j = 2 * i + 1;
    while(j < count) {
        if (j + 1 < count && array[j+1] < array[j]) j++; // 找出较小的子节点

        if (array[j] >= temp) break; // 如果较小的子节点比父节点大, 直接返回

        array[i] = array[j]; // 设置父节点为较小节点
        i = j; // 调整的子节点作为新一轮的父节点
        j = 2 * i + 1; // 调整的子节点的子节点
    }

    array[i] = temp;
}

void heapSort(int array[], int count) {

    for (int i = (count - 1) / 2; i >= 0; i--) {
        // 构造小顶堆
        minHeapFixdown(array, i, count);
}

    for (int i = count - 1; i >= 1; i--) {

        // 交换根结点与最末节点
        int temp = array[i];
        array[i] = array[0];
        array[0] = temp;

        // 剩余的n-1个元素再次建立小顶堆
        minHeapFixdown(array, 0, i);

    }

}
```

快速排序

算法思想：先从数列中取出一个数作为基准数 -> 将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边 -> 再对左右区间重复第二步，直到各区间只有一个数

```
int quickSortPartition(int array[], int start, int end) {

    int i = start, j = end;
    // 默认第一个元素为哨兵
    int sentry = array[i];

    while (i < j) {

        // 从右往左找第一个小于哨兵的元素
```

```
        while (i < j && array[j] >= sentry) j--;
        // 找到了
        if (i < j) {
            array[i] = array[j];
            i++;
        }

        // 从左往右找第一个大于哨兵的元素
        while(i < j && array[i] <= sentry) i++;
        // 找到了
        if (i < j) {
            array[j] = array[i];
            j--;
        }

    }
    // 把哨兵放到i == j的位置上
    array[i] = sentry;

    // 返回哨兵的位置
    return i;
}

void quickSort(int array[], int start, int end) {

    if (start < end) {

        // 找出分界点
        int index = quickSortPartition(array, start, end);
        quickSort(array, start, index - 1); // 对分界点左边进行排序
        quickSort(array, index + 1, end); // 对分界点右边进行排序
    }

}

void quickSortEntry(int array[], int count) {
    quickSort(array, 0, count - 1);
}
```

基数排序

基数排序的算法复杂度不会因为待排序列的规模而改变. 基数排序又称为桶排序. 基数排序有3个重要概念：

- r：关键字的基数, 指的是关键字k的取值范围, 十进制数的话, k=10
- d：位数
- n：关键字的个数

这里给个例子, 没有代码.

```
例如一组序列121 83 17 9 13

1. 先根据个位数排序
    121 83 13 17 9
2. 再根据十位数排序
    9 13 17 121 83
3. 再根据百位数排序
    9 13 17 83 121
4. 由于没有千位数，所以算法结束

ps : 需要注意的是，基数排序每一轮排序所采用的算法必须是稳定的排序算法，
也就是说，例如13和17的十位数均为1，但是由于个位数排序的时候13是在17的前面的，
所以十位数排序过后13也必须在17的前面.
```

查找算法

算法	最优复杂度	最差复杂度	平均复杂度
顺序查找	O(1)	O(n)	O(n)
折半查找	O(1)	O(log n)	O(log n)
哈希查找	O(1)	O(1)	O(1)

顺序查找

算法思想：顾名思义就是从数组的0坐标开始逐个查找对比.

```
int orderSearch(int array[], int num, int count) {
    int index = -1;

    for (int i = 0; i < count; i++) {
        if (array[i] == num) {
            index = i;
            break;
        }
    }
    return index;
}
```

折半查找

算法思想：在一个有序数组里，先对比数组中间的数middle与要查找的数num的大小关系

- middle == num：直接返回

- middle < num：递归查找数组右半部分
- middle > num：递归查找数组左半部分

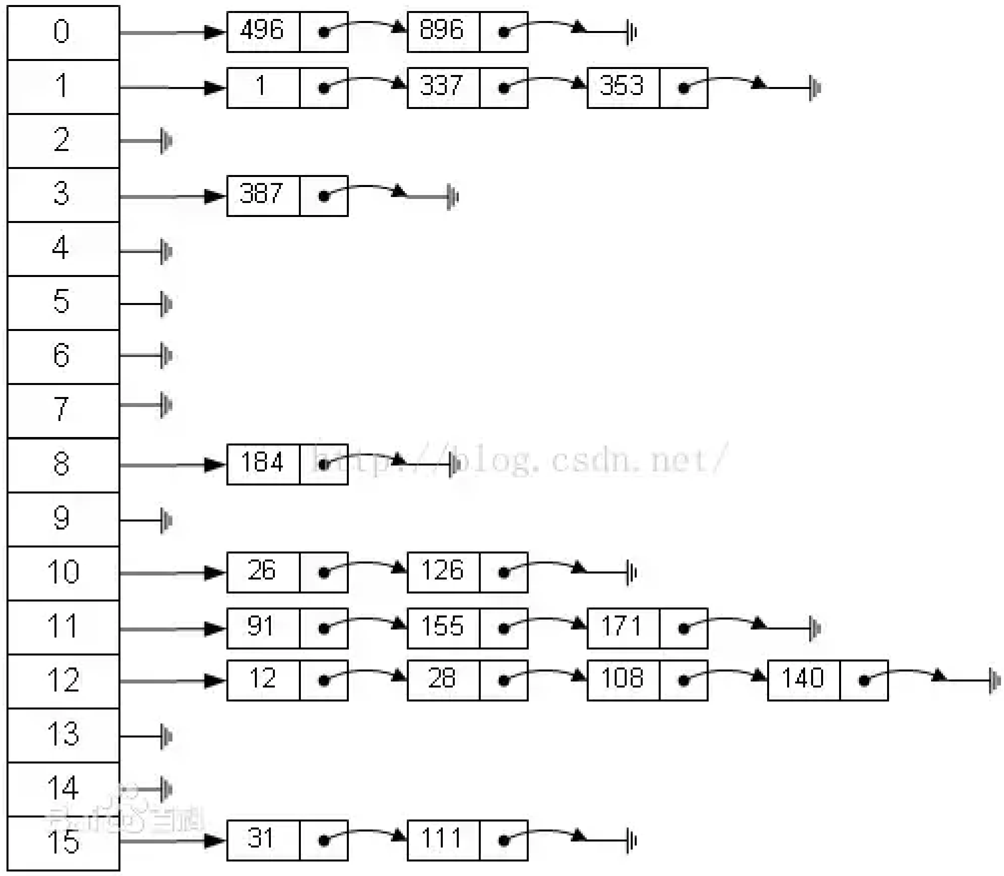
```
int binarySearch(int array[], int num, int start, int end) {

    int index = -1;

    if (start <= end) {

        int middle = (start + end) / 2; // 数组中点
        if (array[middle] == num) {
            index = middle; // 找到了
            return index;
        }
        else if (array[middle] > num) {
            return binarySearch(array, num, start, middle - 1); // 查找数组左边
        } else if (array[middle] < num) {
            return binarySearch(array, num, middle + 1, end); // 查找数组右边
        }

    }

    return index;
}
```

哈希查找

哈希查找需要一张哈希表，哈希表又称为散列法，是高效实现字典的方法. 查找速度在O(1)，这说明无论你需要查找的数据量有多大，他都能在常数时间内找到，快得有点违背常理吧？嘿嘿.

哈希表几个重要的概念：

- **负载因子：** α = n / m (n为键的个数，m为单元格个数)，负载因子越大，发生冲突的概率则越大
- **哈希函数：**
  - 哈希函数是指你把一样东西存进去之前，先对它的key进行一次函数转换，然后在通过转换出来的值作为key，把你要存的东西存在表上.
- **碰撞解决机制：**
  - 再哈希法：使用其他的哈希函数对key再次计算，直到没有发生冲突为止(计算量增加，不推荐)
  - **线性勘测法：** 通过一个公式，算出下一个地址，(存储连续的)
  - **二次探测法：** 生成的地址不是连续的，而是跳跃的.
  - 如果两样东西通过哈希函数算出来的key相同怎么办？东西怎么存？这个时候就是碰撞检测机制派上用场的时候
  - **方法一：** 开散列法，也称分离链法：即相当于在数组中每个格子是一个链表，只要发生冲突就把后来的value拼接在先来的value后面，形成一条链.



- **方法二：** 闭散列法，也称开式寻址法：

可以这么说，哈希函数设计得越好，冲突越少，哈希表的效率就越高.

需要了解的名词

- **旅行商问题**
  - 哈密顿回路：经过图中所有顶点一次仅一次的通路
- **凸包问题**
  - 凸集合：平面上一个点集合，任意两点为端点的线段都属于该集合
  - 凸包：平面上n个点，求包含这些点的最小凸多边形
  - 极点：不是线段的中点
- **曼哈顿距离**
  - dM (p1, p2) = |x1 - x2| + |y1-y2|
- 深度优先查找(DFS)：用栈实现
- 广度优先查找(BFS)：用队列实现
- 拓扑排序(无环有向图)
  - 深度优先查找 -> 记住顶点(出栈)的顺序，反过来就是一个解
  - 找源，它是一个没有输入边的顶点，然后删除它和它出发的所有边，重复操作直到没有源为止.
- **2-3树**

- 2节点：只包含1个键和2个子女
- 3节点：包含2个键和3个子女
- 高度平衡<所有叶子节点必须位于同一层>
- 可以包含两种类型的节点

- **BST树：**
  - 也称为B树
  - 二叉查找树, 随着插入和删除的操作有可能不是平衡的.
- **AVL树：**
  - 平衡二叉查找树
  - 左右子树深度只差不超过1
  - 左右子树仍为平衡二叉树
- **RBT红黑树：**
  - 一种平衡二叉树
  - 跟AVL树类似, 通过插入和删除操作时通过特定操作保持二叉查找树的平衡, 从而有较高的查找性能.
  - 相当于弱平衡的AVL数(牺牲了一定次数的旋转操作), 若查找 > 插入/删除, 则选择AVL树; 若差不多则选择红黑树
- **哈夫曼树**
  - 自由前缀变长编码
  - 叶子之间的加权路径长度达到最小
  - 哈夫曼编码

几种图论的算法

- **Warshall算法**
  - 选取一个顶点作为桥梁, 考察所有顶点是否可以通过该桥梁到达其他的顶点
  - 求有向图的传递闭包
  - 算法思想
- **Floyed算法**
  - 选取一个顶点作为桥梁, 考察所有顶点是否可以通过该桥梁到达其他的顶点, 如果能, (如a到c, b为桥梁)再比较Dab + Dbc < Dac ? 如果成立, 则更新最短距离
  - 求每个顶点到各个顶点的最短路径
  - 算法思想
- **Prim算法**
  - 首先找出S = {你选取的一个顶点}, 然后添加另一顶点(该顶点∈(V-S)且它们两顶点之间的边的权重最小, 直到S = V.
  - 求无向带权连通图的最小生成树(每次按节点递增)

- 算法思想

- **Kruskal算法**
  - 第一次选出权重最小的边加入，之后每次选择权重最小的边加入并不构成环.
  - 求无向带权连通图的最小生成树(每次按边递增)
  - 算法思想
- **Dijkstra算法**
  - 找一个源(起点)，之后求出离起点最近的点的距离；然后第二近，以此类推(允许通过其他点为中间点)，设置顶点集合S，只要源到该顶点的距离已知就把该顶点加入到S中. 直到S包含了V中所有顶点.
  - 求有向带权图中一个"源"到所有其他各顶点的最短路径
  - 算法思想
