**《算法》读书笔记**

**班级：软工181 姓名：林鑫 学号：2018212212189**

**读书日期：2020-04-15 至 2020-06-02**

**前言：这是2020年大二下学期，阅读《算法》以及各种优秀博客的读书笔记，记录下学习算法的一些心路历程，并对一些经典的算法做一些总结，希望以后能够温故而知新。**

**目录**

1. [**经典八大排序算法**](#一)
2. [**背包、队列和栈**](#二)
3. [**二叉查找树**](#三)
4. [**无向图**](#四)
5. [**有向图**](#五)
6. [**最小生成树**](#六)
7. [**最短路径**](#七)
8. **……待续**
   1. **经典八大排序算法**

**1.1 冒泡排序**

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

**1.2 算法描述**

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个；

② 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

1. 针对所有的元素重复以上的步骤，除了最后一个；
2. 重复步骤1~3，直到排序完成。

**1.3 代码实现：**



**1.4算法分析**

最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

**2.1 选择排序（Selection Sort）**

表现最稳定的排序算法之一，因为无论什么数据进去都是O(n2)的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

**2.2 算法描述**

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

1. 初始状态：无序区为R[1..n]，有序区为空；
2. 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
3. n-1趟结束，数组有序化了。

**2.3代码实现**



**2.4 算法分析**

最佳情况：T(n) = O(n2) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

**3.1 插入排序（Insertion Sort）**

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

**3.2算法描述**

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序；
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤2~5。

**3.3 代码实现**



**3.4 算法分析**

最佳情况：T(n) = O(n) 最坏情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

**4.1 希尔排序（Shell Sort）**

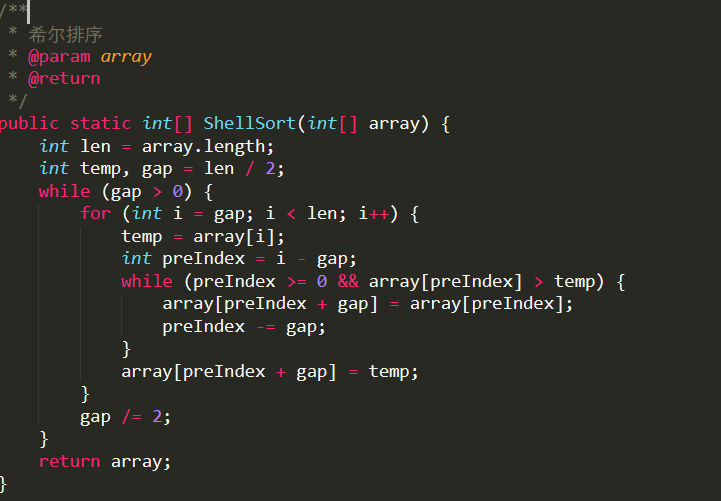
希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。希尔排序是把记录按下表的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

**4.2 算法描述**

我们来看下希尔排序的基本步骤，在此我们选择增量gap=length/2，缩小增量继续以gap = gap/2的方式，这种增量选择我们可以用一个序列来表示，{n/2,(n/2)/2...1}，称为增量序列。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。此处我们做示例使用希尔增量。先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

1. 选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；
2. 按增量序列个数k，对序列进行k 趟排序；
3. 每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

**4.3 代码实现**



**4.4 算法分析**

最佳情况：T(n) = O(nlog2 n) 最坏情况：T(n) = O(nlog2 n) 平均情况：T(n) =O(nlog2n)

**5.1 归并排序（Merge Sort）**

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

**5.2 算法描述**

把长度为n的输入序列分成两个长度为n/2的子序列；

1. 对这两个子序列分别采用归并排序；
2. 将两个排序好的子序列合并成一个最终的排序序列。

**5.3 代码实现**



**5.4 算法分析**

最佳情况：T(n) = O(n) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)

**6.1 快速排序（Quick Sort）**

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

**6.2 算法描述**

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

1. 从数列中挑出一个元素，称为 “基准”（pivot）；

②重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

1. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

**6.3 代码实现**



**6.4 算法分析**

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)

**7.1 堆排序（Heap Sort）**

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

**7.2算法描述**

① 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

② 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

③ 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

**7.3 代码实现**



**7.4 算法分析**

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)

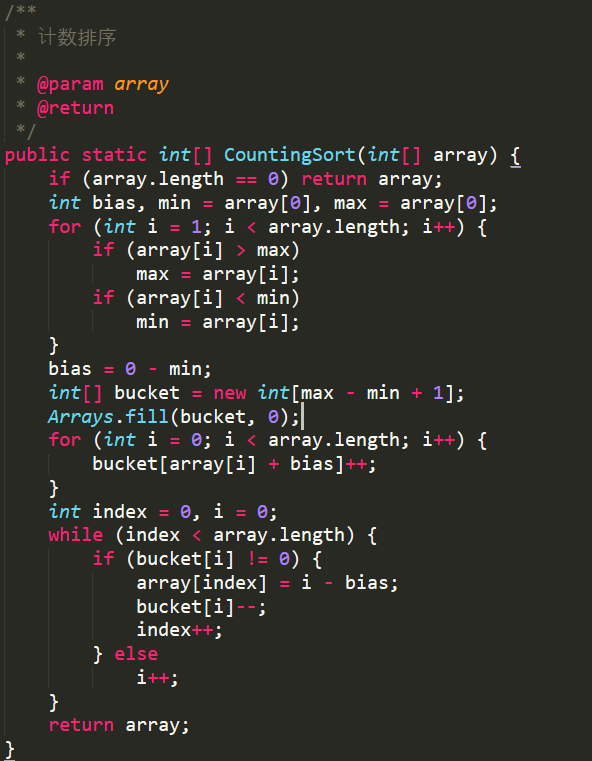
**8.1 计数排序（Counting Sort）**

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

**8.2 算法描述**

1. 找出待排序的数组中最大和最小的元素；
2. 统计数组中每个值为i的元素出现的次数，存入数组C的第i项；
3. 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；
4. 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

**8.3 代码实现**

****

**8.4 算法分析**

当输入的元素是n 个0到k之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n+k)

**对排序算法的总结：**



* 1. **队列和栈**

队列:先进先出(FIFO)

下压栈:后进先出(LIFO)

求值算法:

将操作数压入操作数栈

将运算符压入运算符栈

忽略左括号

遇到右括号时,弹出一个运算符,弹出所需数量的操作数,并将运算结果压入操作数栈

* 1. **二叉查找树**

一棵二叉查找树(BST)是一棵二叉树,其中每个结点都含有一个Comparable的键以及值,且每个结点的键都大于其左子树中的任意结点的键而小于其右子树的任意结点的键.

每个结点还会有一个结点计数器,它给出了以该结点为根的子树的结点总数.size(x)=size(x.left)+size(x.right)+1

一棵二叉查找树代表了一组键的集合,而同一个集合可以用多颗不同的二叉查找树表示.如果将一棵二叉查找树的所有键按从左到右的顺序投影到一条直线上,那么会得到一条有序的键列.

用递归的方法查找:

如果树是空的,则查找未命中;

如果被查找的键较小就选择左子树;

如果被查找的键较大就选择右子树;

插入和查找的难度差不多,如果一个结点不存在,只需要将链接指向一个含有被查找的键的新结点,并更新搜索路径上每个父结点中结点计数器的值.两者的时间复杂度都为O(lnN).

排名也是递归实现的:

如果给定的键和根结点的键相等,返回左子树的结点总数t;

如果给定的键小于根结点,返回该键在左子树中的排名;

如果给定的键大于根结点,返回t+1加上它在右子树中的排名;

删除操作通过将x替换为它的后继结点(其右子树中的最小结点)完成.

将指向即将被删除结点的链接保存为t

将x指向它的后继结点min(t.right)

将x的右链接指向删掉后继结点的原右子树

将x的左链接设为t.left

修改结点计数器的值

使用中序遍历来进行范围查找，将符合条件的键放入一个队列,跳过不可能符合条件的子树.

在一棵二叉查找树中,所有操作在最坏情况下所需的时间都和树的高度成正比.因此在某些场景下二叉查找树是不可接受的。

* 1. **无向图**

图是由一组顶点和一组能够将两个顶点相连的边组成的,一般用0至V-1表示一张含有V个顶点的图中的各个顶点,用v-w或w-v表示;连接v和w的边.边仅仅是两个顶点之间的连接的图称为无向图.

一条连接一个顶点和其自身的边称为自环.连接同一对顶点的两条边称为平行边.一般将含有平行边的图称为多重图,将没有平行边或自环的图称为简单图.

某个顶点的度数即为依附于它的边的总数.子图是由一幅图的所有边的一个子集组成的图.

路径是由边顺序连接的一系列顶点.简单路径是一条没有重复顶点的路径.

环是一条至少含有一条边且起点和终点相同的路径.简单环是一条不含有重复顶点和边的环.

如果从任意一个顶点都存在一条路径到达另一个任意顶点,那么这幅图是连通图.

图的密度是指已经连接的顶点对斩所有可能被连接的顶点对的比例,由此分出稀疏图和稠密图.

二分图是一种能够将所有结点分为两部分的图,其中图的每条边所连接的两个顶点都分别属于不同的部分.

一般采用邻接表数组来表示图.它将每个顶点的所有相邻顶点都保存在该顶点对应的元素所指向的一张链表中.它使用的空间和V+E成正比.添加一条边所需的时间为常数.遍历顶点v的所有相邻顶点所需的时间和v的度数成正比.

深度优先搜索(DFS)是搜索连通图的经典递归算法,所需的时间和顶点的度数之和成正比.使用的是栈:

在访问其中一个顶点时将它标记为已访问

递归地访问该顶点的所有没有被标记过的邻居顶点

广度优先搜索(BFS)是解决单点最短路径的经典算法,它所需的时间在最坏情况下和V+E成正比.使用的是队列,先将起点加入队列,重复以下步骤直到队列为空:

将与v相邻的所有未被标记过的顶点加入队列,删除v

取队列中的下一个顶点v并标记它

DFS和BFS的区别在于DFS总是获取最晚加入的顶点,而BFS总是获取最早加入的结点.

DFS更适合实现图的抽象数据类型,因为它能更有效地利用已有的数据结构.而union-find算法适合于只需要判断连通性的任务.

利用一个符号表保存字符和索引,一个数组保存反向索引和一张图就可以实现符号图。

* 1. **有向图**

一幅有向图是由一组顶点和一组有方向的边组成的,每条有方向的边都连接着有序的一对顶点.在有向图中,一个顶点的出度为由该顶点指出的边的总数;一个顶点的入度为指向该顶点的边的总数.用v→w表示有向图中一条由v指向w的边.

有向路径由一系列顶点组成,对于其中的每个顶点都存在一条有向边从它指向序列中的下一个顶点.有向环为一条至少含有一条边且起点和终点相同的有向路径.简单有向环是一条不含有重复的顶点和边的环.

和无向图类似,一般使用邻接表来表示有向图,用顶点v所对应的邻接链表中包含一个w顶点来表示边v→w.每条边都只会在其中出现一次.DFS和BFS同样可用于有向图.

拓扑排序:给定一幅有向图,将所有的顶点排序,使得所有的有向边均从排在前面的元素指向排在后面的元素.即有优先级限制下的调度问题.当且仅当一幅有向图是无环图时它才能进行拓扑排序.

有向无环图(DAG)是一幅不含有环的有向图.想要进行有向环检测,可以基于DFS,一旦找到一条边v→w且w已经存在于栈中,就找到了一个环.

DFS遍历一幅图以后,有以下三种排列顺序:

前序:在递归调用之前将顶点加入队列,即dfs()的调用顺序

后序:在递归调用之后将顶点加入队列,即顶点遍历完成的顺序

逆后序:在递归调用之后将顶点压入栈,即这幅图的拓扑排序(如果是有向无环图)

如果两个顶点v和w是互相可达的,则称它们为强连通的.如果一幅有向图中的任意两个顶点都是强连通的,则称这幅有向图也是强连通的.两个顶点是强连通的当且仅当它们都在一个普通的有向环中。

通常用Kosaraju算法来计算强连通分量:

在给定的一幅有向图G中,计算它的反向图的逆后序排列。

在G中进行DFS,但是要按照1得到的顺序来访问所有未被标记的顶点。

所有在同一个递归DFS调用中被访问到的顶点都在同一个强连通分量中。

**六、** **最小生成树**

加权图是一种为每条边关联一个权值或是成本的图模型.

图的生成树是它的一棵含有其所有顶点的无环连通子图.一幅加权无向图的最小生成树(MST)是它的一棵权值最小的生成树.

在计算最小生成树时,做以下约定:

只考虑连通图

边的权重不一定表示距离

边的权重可能是0或者负数

所有边的权重都各不相同

树的两个重要性质:

用一条边连接树中的任意两个顶点都会产生一个新的环

从树中删去一条边将会得到两颗独立的树

图的一种切分是将图的所有顶点分为两个非空且不重复的两个集合.横切边是一条连接两个属于不同集合的顶点的边.

切分定理:在一幅加权图中,给定任意的切分,它的横切边中的权重最小者必然属于图的最小生成树.

切分定理是解决最小生成树问题的所有算法的基础.这些算法都是一种贪心算法的特殊情况.即找到一种切分,它产生的横切边均不为黑色,将它权重最小的横切边标记为黑色,直到标记了V-1条黑色边为止.不同之处在于保存切分和判定权重最小的横切边的方式.

同样可以使用邻接表来表示加权无向图,只需要增加一个权重域。

**Prim算法**:

1. 一开始最小生成树只有一个顶点,然后会向它添加V-1条边.每次总是将下一条**连接**树顶点与非树顶点且**权重最小**的边加入树中.每一步总是为一棵树**添加一条边**.
2. 使用**优先队列**来根据权重比较所有边
3. 分为**延时实现**和**即时实现**.区别在于是否**立即删除**失效的横切边(即连接新加入的顶点和树中已有顶点之间的边).
4. **延时实现**所需**空间**与**E**成正比,所需**时间**与**ElogE**成正比.
5. **即时实现**不仅删除失效的边,而是仅保存非树顶点到树顶点的边中权重最小的那条,并在每次加入新顶点后检查是否需要更新.所需**空间**与**V**成正比,**时间**与**ElogV**成正比.

**Kruskal算法**:

1. 按照边的**权重顺序**处理它们,将边加入最小生成树中,加入的边不会与已经加入的边构成环,直到树中含有V-1条边为止.每一步总是连**接森林中的两棵树**(包括单顶点树).
2. 在实现中,使用**优先队列**来将边按照权重排序,使用**union-find**来识别会形成环的边,以及一条**队列**来保存最小生成树的所有边.
3. 所需**空间**与**E**成正比,所需**时间**与**ElogE**成正比.

**七、****最短路径**

在一幅**加权有向图**中,从顶点s到顶点t的**最短路径**是所有从s到t的路径中的**权重最小**者.**最短路径树(SPT)**包含了顶点s到所有可达的顶点的最短路径**边v->w的松弛**是指检查从顶点s到w的最短路径是否是先从s到v,然后再由v到w,即.如果是,则更新,如果不是,则称这条边**失效**了.

**顶点的松弛**是指对该顶点指出的所有边进行松弛.

**最优性条件**:当且仅当对于从v到w的任意一条边e,这些值都满足distTo[w]<=distTo[v]+e.weight()时它们是最短路径的长度.这证明了判断是否为最短路径的**全局条件**与松弛时检测的**局部条件**是**等价**的.

**通用最短路径算法**:放松图中的任意边,直到不存在有效边为止.

**Dijkstra算法**(注意只能解决**正权重**的加权有向图中的最短路径问题):

1. 首先将distTo[s]初始化为0,distTo[]中的其他元素初始化为起点s到该顶点的距离,注意如果不相邻则为正无穷.
2. 然后将distTo[]**最小的非树顶点松弛**并加入树中
3. 重复2,直到所有的顶点都在树中或者所有的非树顶点的distTo[]值均为无穷大.

按照**拓扑顺序**放松顶点,就能在和E+V成正比的时间内解决**无环加权有向图**的单点最短路径问题.同理,要解决无环加权有向图的**最长路径**问题,只需将原图中所有边的权重变为负值,再求最短路径即可.

当且仅当**加权有向图**中至少存在一条从s到v的有向路径且该路径上的任意顶点都不存在于任何**负权重环**中时,s到v的最短路径才是存在的.

**Bellman-Ford算法**:

1. 在任意含有V个顶点的**加权有向图**中给定起点s,从s无法到达任何负权重环,可以解决其中的单点最短路径问题
2. 将distTo[s]初始化为0,其他distTo[]元素初始化为无穷大,以任意顺序放松有向图的所有边,重复V-1轮
3. 算法所需的**时间**和**EV**成正比,**空间**和**V**成正比.

**八、未完待续。。。**