# 应用中常见算法

## 动态规划算法：

与贪婪算法和分治算法的区别

动态规划的重要思想：状态 状态转移方程 阶段

经典动态01背包问题

<http://blog.csdn.net/luoweifu/article/details/18509317>

<http://blog.csdn.net/mu399/article/details/7722810>

<http://www.cnblogs.com/steven_oyj/archive/2010/05/22/1741374.html>！！！！！！！！！！！

写算法注意：

1．如果是用二维数组，那么需要多大的空间

2．初始值设置

3.循环的开始值和结束值

4.决策点的设定

做一个题：

一个数组由2n个整数组成，把这个数组分成两半，各有n个整数，求一个分法，使这两个子数组和的差最小

## 第五章 优先队列

## 第六章 树

## 第七章 排序方法

### 7.1 希尔排序：

复杂度n3/2次方， 不稳定。

最坏的时间复杂度O（N^2）,平均时间复杂度：

思想：将一个最初的杂乱的数组，一步一步使其**基本有序**，最后做一个增量为1的插入排序。

关键：增量的选取直接决定排序的复杂度，经验告诉我们可以采取n/2+1的方式选取。

几种常见的增量序列：2^k-1，4^i-3\*2^i+1。

### 7.2 快速排序：

新的递归。

Java标准库中也实现了这种排序。

时间复杂度：平均时间Nlog（N）。最坏时间：O(N^2)

改进：结合堆排序和快速排序，可以使最坏时间降为Nlog（N）

难点：

1.枢纽值的选取：比较好的方案：a 三数中值分割法

2.等于枢纽值的数值是否要停？？最好都停下来。

预处理：对A[left],A[right],A[center]排序，然后让A[left]在序列最左边A[right在最右边，A[center]在right-1的位置。之后用i指向left+1，j指向right-2，

判断A[i]是否大于A[center]

i停止；判断A[j]是否小于A[center]

i++

i<j

A[i]交换A[j]

j--

i>j时，A[i]与A[center]交换

### 7.3堆排序

时间复杂度是：N\*LOG（N）：进行N次deleteMin操作，每次deltemin操作消耗时间是LOG（N）。

小技巧：删除时可以把被删除的数存在原数组的最后一位。最好采用DeleteMax来构建堆，这样可以顺序输出队列中的数据。

总的平均比较次数为：2NLog（N）-O(N)

### 7.4简单排序

简单排序的平均时间都是O（N^2）,因为任何交换相邻元素的排序法的时间都不会优于O（N^2）

#### **7.4.1选择排序**

(Selection Sort)的基本思想是：每一趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子文件的最后，直到全部记录排序完毕。

#### 7.4.2冒泡排序

### 7.4.3插入排序

前0到p-1都是有序的，第P个元素依次和p-1，p-2等元素比较，如果小于某个元素，该元素后移一位。直到比较到了0位置，或是大于某个元素。

### 7.6归并排序

时间复制度：NLog（N）

缺点：需要在空间上做出牺牲。需要存储空间增加一倍。

标准java数组库中的排序是用归并排序实现的。因为对java语言来说，比较两个对象往往可能很费时间，移动对象不是很费时间（因为是传递引用）。所以对归并排序，比较次数较少，尽管需要额外附加一些移动。但是这仅仅针对Java，如果是C++则不是这样的。C++比较时间比较少，移动费时比较多。

算法思想：递归+合并 通常是针对数组结构的存储

1.将一个数组选出right center end下标。

2.递归调用左边部分数组和右边部分数组

3.合并左边部分和右边部分。

合并时：比较左右数组的元素，并移动下标，当下标大于数组的容量。复制剩下的数组到合并后的数组。

### 7.7桶排序

### 7.8 外部排序

## 第八章 图论算法

### 8.1基本定义：

什么是图：顶点和边的集合。

有向图和无向图、边的权重

路径：一系列顶点序列的集合 路径的长：路径是所含有边的条数

**环**：比如（v,v）。一个点为起点，并以自己为终点的一条路径，中间没有经过其他点。

**圈**：起点和终点是同一个点，而且路径的长度大于1。

简单路径： 简单圈： 有向无圈图：DAG

**连通图**：到每个顶点都存在一条路径 强连通图（有向的连通图） 弱连通图（不强连通，但是去掉方向后的图示连通的）

入度和出度（针对有向图）

**完全图**：每一对顶点，都存在一条边则有边数为：n\*（n-1）/2。

负值圈：指有负权值的圈。

单源最短路径问题和多源最短路径问题

### 8.2图的存储：

可以根据图是稠密的还是稀疏的，如果一个图的边很多，比如完全图，就算是**稠密**的，那么我们可以通过**二位数组来存储**。如果一个图边不多，这个时候选用二维数组来存储就会浪费空间，这个时候用**邻接表**来存储。

### 8.3拓扑排序：

拓扑排序是指对一个有向无圈图。如果有一条（Vi ，Vj）那么最后的排序Vi就肯定出现在Vj前面。

算法步骤：

**用数组存储图**：时间消化是V\*V

1. 遍历数组找出入度为0的点且还未被编号，将他们的某个表示顺序的字段（编号）设置为0。
2. 将与1中相邻的点的入度减去1。
3. 回到第1步。

**用邻接表存储图**：V+E。需要有一个队列来当前存储入度为0点。

1. 遍历数组找出点入度为0的点，依次入队
2. 将队列中的数组出队，并编号
3. 将2中出队的点的邻接入度减1，并判断是否应该入队。

### 8.4最短路径算法

**无权最短路径算法**：可以采用类似于广度优先搜索来实现，可以用队列的数据结构来存储节点。

**Dijkstra算法**：不能用于有负权的图。

稠密时：可以采用数组

稀疏时：可以采用优先队列来存储距离。

优先队列也有几种实现方式：可以用二叉堆或是斐波那契堆

一个点w的更新也有两种方式：DecreaseKey 和插入优先队列中。

负边值的图：

无圈图：

关键路径分析法：

## 树和堆

（啊哈！算法）

**树的基本知识：**

**定义**：无回路的连通图

**基本性质**：N个点，有N-1条边

二叉树：

完全二叉树：

满二叉树：

堆：可以用来实现一个优先队列

优先队列：

存储方式：

堆的建立：两种方式：插入法和判断所有子节点形成的树都是最小堆方法。

堆的根节点删除和节点插入：

**堆排序：**

**堆的应用：求第K大的数、第K小的数、**

并查集：

树的应用，在一个森林中，根据已知信息合并小树成大树，再判断一共有几棵树。

两个原则：判断已知树的父节点（擒贼先擒王原则），向左合并原则。

应用：在一推个体中，某些个体两两联系，从而形成一个个社交网，在判断有多少个社交网的时候可以用并查集。

其他：

多元素中的主元素问题：

1. 求某个序列中超过50%的某个数。（两个不一样的数在序列中删除后，最多那个数依然超过50%）。

算法：1、如果a[i]与ME相同，++count，否则转2

2、--count，如果count==0，转3,否则转1

3、令新的主元素ME=a[i+1]，count = 1，转1(因为前面一部分已经的主元素与非主元素个数相同，重新开始。为什么呢？因为如果前面局部的主元素真的是全局主元素，那么数组减少的个数为2\*i个，i个主元素，i个非主元素，剩下的数组中，主元素的个数依然超过剩下数组长度的一半，继续如上处理。)

int mainElement(int a[],int n)

{

if(n > 0)

{

int ME = a[0],count = 1;

int i = 1;

while(i < n)

{

if(a[i] == ME)

{

++count;

++i;

}

else

{

--count;

if(0 == count)

{

ME = a[++i];

count = 1;

}

++i；

}

}

return ME;

}

}

## 深度优先算法

1. 以某个节点分析
2. 可以采用队列存储要分析的节点。

## 广度优先算法

1. 以某个节点分析
2. 采用递归算法利用1.中的函数。
3. 注意边界条件，和条件分支。
4. 注意什么时候需要标注经过的点，什么时候不需要。
5. 特别要注意什么时候不再需要递归，返回调用函数。
6. 更适合用所有边的权值都一样的情况

## FloodFill漫水填充法（染色法）

可以用基于广度或是深度的方法来实现填充。

应用：图像分割、物体识别、图中求独立的子图个数

## 存储图的方法：

二维数组邻接矩阵表示法

邻接表

## 两点最小路径

### 深度优先算法：

### 广度优先算法：

### Floyd算法：

核心思想：通过中间点是两点间的距离变短。

核心代码：

### Bellman-Ford算法：

核心思想：经过一轮遍历所有边的情况下，可以确定一部分点到特定源点的距离（第一轮遍历确定离源点距离为1的点，第二轮遍历确定距离源点距离为2的点。。。。。。。）

经过n-1轮遍历可以完成距离源点最远点的遍历。

核心代码：

For（i=1;i<n;i++）

For(i=1;i<=m;i++)

If(dis[v[i]]>u[i]+w[i]) dis[v[i]]= u[i]+w[i];

应用：有负权的情况

改进的Bellman-Ford算法：

1. 其实n-1轮松弛是最大极限，往往在前几轮就结束了，所以可以增加一个变量来记录源点到个点距离的数组Dis[]没有变化时就break
2. 其实每轮遍历的时候只用检查Dis[]中距离发生变化的点，只有这些点的邻点才需要更新距离。这样需要遍历的边就少了

算法代码：一个Dis[]数组记录当前个点到源点的距离，一个队列que[]head标记当前点，que里面放当前点的邻点。一个标记数组book[]，标记新点是否已经入队。

Dijkstra算法：