Each of you comes here today a hopeful. Wanting in on the game. A month ago you were in med school being taught by doctors. Today, you are the doctors. The seven years you spend here as a surgical resident will be the best and worst of your life. You will be pushed to the breaking point. Look around you. Say hello to your competition. Eight of you will switch to an easier specialty. Five of you will crack under the pressure and two of your will be asked to leave. This is your starting line. This is your arena. How well you play? That's up to you.

——实习医生里面的台词，共勉之

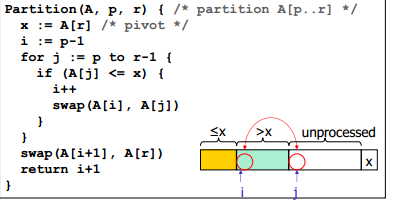
算法第二次测验整理

算法第一次测验的整理是到归并排序，以下是第六章开始。

QuickSort(快速排序):

主要思想：先选定一个数（伪代码中取出数组的最后一个数）作为支点pivot，将所排的序列分成两个集合set，比pivot小的放在pivot前面位置，比pivot大的放在其后面位置，因此可以得出pivot在数组中的确切位置——这个寻找pivot在数组中位置的功能由一个partition函数实现。现在，我们已经把一个排序数组的任务划分成两个排序数组的小任务——排序在pivot之前的数组，和在pivot之后的数组。因此，我们可以进行递归调用来实现快速排序。

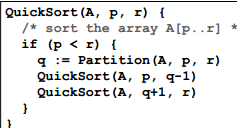
Partition的伪代码（和国内教的代码不一样，有兴趣可以去看一下国内的代码）：



解释：将数组的最后一位作为pivot，从数组的第一位开始进行处理。A[j]指向的是处理完的数组序列。如果是小于pivot的数，就放在A[p]..A[i]这部分中，大于pivot的数就放在A[i+1]..A[j]之中。当处理完A[p]..A[r-1]之间的数时，i+1位置就是pivot在数组中的正确位置，所以又swap(A[i+1],A[r])的操作。

时间复杂度：O(n)

快排的伪代码：



时间复杂度：

最好情况下，每次选取pivot的时候，所得的两个子数组A[p..q-1]和A[q+1..r]是相同的大小。

T(n)=2T(n/2)+O(n) ---(主定理master theorem解)-🡪 O(nlogn)

最差情况下，当pivot是最小或者最大值的时候，我们划分的子数组有一个为空，因此快排退化为选择排序。

T(n)=T(n-1)+O(n) -🡪O(n^2)

为了减少排序的时间复杂度，我们要尽量避免选择的pivot是最值。方法如下：

1. Randomized algorithms: uses some random inputs in the algorithm(因为我们选取的pivot位置是固定的，所以使用随机输入可以保证选取的pivot不是最值)
2. 随机选择数组中的元素作为pivot（不固定pivot的位置）

Java中现成的调用：Java Arrays.sort()

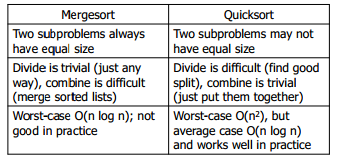
MergeSort vs. QuickSort

相同点：

1. 都使用了分而治之（divide and conquer）的思想

2. Divide/combine work O(n) time: 递推公式里面的O(n)的由来

不同点：

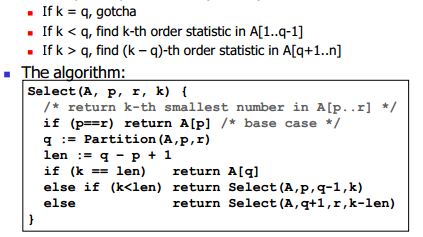


Mergesort主要是在combine上面，Merge()函数将两个有序数列整合起来（又申请了一个c数组辅助）。Quicksort重点在divide，partitio()函数找到pivot的位置，再划分数组。

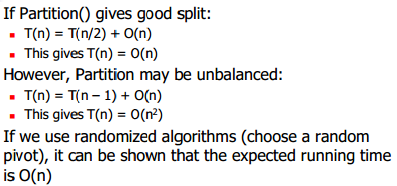
Order Statistics

Find the k-th order statistic(在数组中找到第K大的数)

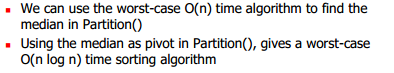
思路：参考快速排序，当我们选取一个数作为pivot的时候，我们可以通过调用partition()函数得知此数在数组之中的位置。假设我们得到pivot的位置是q，如果k<q，我们查找所有比pivot小的子数组（此时因为partition函数，这些数已经全部放置在pivot之前，继续递归调用前半部分数组即可）。如果k>q，我们查找所有比pivot大的子数组（同样因为partition函数，这些数已经全部放置在pivot之后，查找这部分数组中第k-q大小的数即可）。如果k=q，直接返回数值。



时间复杂度（和快排类似）：



改进的快速排序：



将中位数median设置为pivot。老师PPT上的median是指floor(n/2)，和数学上的定义还是不太一样。具体的实现可以参考我们的作业题。算法复杂度是O(nlogn)

Integer Multiplication假设1bit位操作时间复杂度是O(1)

Add two n-bit numbers: O(n) //用最简单的方法

Grade school Multiplication: O(n^2)

使用分而治之（divide and conquer）思想:

多项式展开（2个n bit位的数相乘）：

四个子问题的规模是n/2， 2^n对于计算机来说就是左移操作，所以问题规模主要取决于a,b,c,d T(n)=4T(n/2)+O(n) -🡪 T(n)=O(n^2)

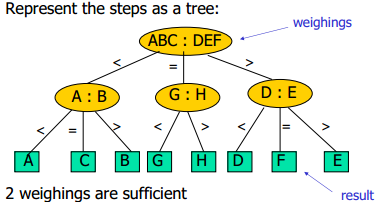
Karatsuba Algorithm改进上部分的算法：

上图中因式分解的部分ad+bc可以拆解成(a+b)(c+d)-ac-bd，我们可以看到ac,bd已经在上图中算出了，所以我们只要再增加一个(a+b)(c+d)的操作，减少了一个乘法操作（加法的时间复杂度比乘法的小）。

因此时间复杂度：T(n)=3T(n/2)+O(n) -🡪O(n^log2 3)=O(n^1.59)

Chapter 7 Lower Bounds

8个硬币，其中一个是假币，而且假币比较轻，决策树（decision tree）注意决策树的格式如下



叶子个数（没有分支的节点）number of leaves=no. of different outcome 不同输出的个数

枝条个数 no. of branches=no. of outcomes per decision

高度Height=number of decisions(step)决策的个数

Lower bound的定义：

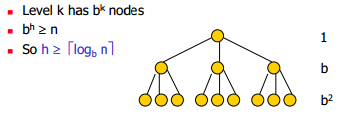




Decision Tree Model

决策树可以用来帮助计算lower bound: the lower bound for worst-case computational complexity is proportional to the largest depth among the decision trees for all possible inputs for a given computational problem.

e.g. 如果有一棵树，一个结点有b个分支，树高是h， 树叶个数是n

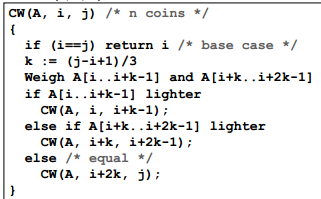


解释：当树高固定的时候，当且仅当树是满N叉树（我们假设所有的节点最大可能的分枝数一致）的时候，树的叶子最多。所以，在我们得知决策的最终输出个数（树的叶子节点个数）之后，求决策树至少要多少高（进行多少次决策才能获得输出）的时候，我们可以考虑最优情况，树是满N叉树，这样得出的树高是最小的。因此，，n是实际输出个数(从决策树上得出的输出结果要比实际得出的输出结果大)，b^h是满N叉树的叶子数。

应用：

我们再回到找假币的那个问题，一共有8个输出，也就是最后的叶子节点是8个。决策树的分支度是3， lower bound==2

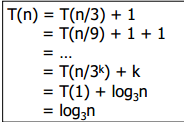
找假币的伪代码：



解释：将硬币分成三部分i..i+k-1,i+k..i+2k-1,i+2k..j，比较了两部分重量之后，就能知道假币在哪个部分之中。并对得到的有假币的部分再次进行递归调用。

时间复杂度：

Tips：这里每一层的常数项是O(1)，相较于快排的O(n)，因为partition()函数中有一层for循环，而这里没有for循环，只有几个赋值操作，所以每次递归调用除了加上三分之一上层规模的时间复杂度之外再加上O(1)



Comparison-based model: only allowed operation is comparison of two elements这是用于数组排序的模型，只允许两个数之间进行比较。在这个模型下，排序最优的时间复杂度是O(nlogn)， lower bound 是Ω（nlogn）

e.g.我们考虑三个数的排序，总共有6种情况的输出（决策树的叶子节点数）：



根据排列组合的原理，我们知道n个数之间大小关系有n!种

因为数只能在两个数之间进行一次比较，所以2^h>n!



证明：



1.证明upper bound:



2.证明lower bound:



Chapter 8 Greedy Algorithms贪心算法

Acitivity Selection活动选择

输入：活动的集合，有活动开始时间和结束时间

目标：尽量参加更多的活动

分析题目的方法：an abstract formalization of problem is useful

我们可以假设每个区间i的开始时间是s(i)，结束时间是f(i)

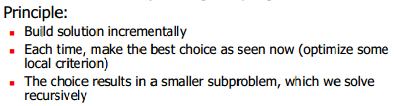
活动有冲突：



1.尝试最短区间优先shortest Interval First来尽可能选择活动

理想化的想法：因为短的区间造成的冲突更少 –>事实上这个算法是错误的

贪心算法的规则：

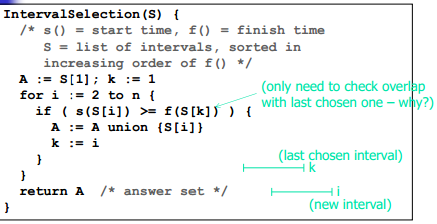


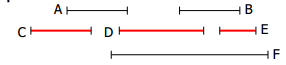
我们采用递推的思想，将一个问题分解成各个子问题，并且每个子问题都选取最优解，因此最后得到的结果就是全局的最优解

2.尝试最早结束优先Earliest Finishing First

选择最早结束的活动

伪代码：





代码解释：

我们看上图的例子，因为S已经按照结束时间升序排好了，所以S={C,A,D,B,E,F}

我们按序扫描下去，选取不冲突的活动即可

时间复杂度：

1. 排序的时间复杂度是O(nlogn)
2. 处理区间的时间复杂度（有一个for循环）：O(n)

所以说总体的时间复杂度是：O(nlogn)

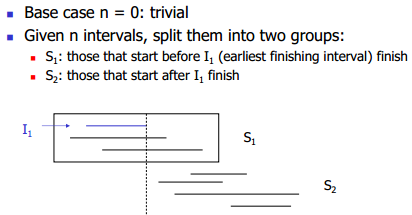
贪心算法的正确性：

e.g.



贪心思想是每次选择当前可选择价值对应最大值的硬币。但是这并不是正确算法。个人分析如下。如果我们根据子问题的最优解得出问题的最优解，那么我们划分出来的子问题并不会改变筛选条件。子问题只是一个小规模的问题，但是并不会改变问题的实质。例如上个例子，我们选择50p，和24p最优解的时候，这个并不能作为74p的子问题。因为他们不满足相同的筛选条件。50p和24p不是遵循同一规则的价值（50P可以接受50p，而24p的最大面额硬币只能从20p开始算起）

证明贪心选择活动的正确性：



因为S1是在最早活动结束之前的区间，所以在S1之中最多只存在一个活动。因此，我们可以知道最大活动个数的最大值是区间S2之中的活动数加上1（S1区间最多只能选择一个活动）。



因为我们知道最早结束的活动没有影响S2之中的活动，所以，选择最早结束时间的活动，能够使得在S2之中的活动数获得最优解。并且选择最早结束的活动也是S1区间之中的最优解。所以，将选择尽可能多的活动这个问题转变成两个子问题，并且在这两个子问题之中都获得了最优解，因此我们可以得知最后获得的解也是最优解。



When is Greedy Optimal?什么时候获得贪心算法的最优解

根据上面的例子讲解，我们可以知道：

全局的最优解可以由局部的最优解得到 Globally optimal solution can be obtained by making locally optimal choices ---🡪贪心算法的解题步骤正确划分子问题，获得局部最优解，最终获得全局最优解

The Knapsack Problem 这是基于贪心算法的背包问题，还有一个是和动态规划有关的背包问题，注意区分

输入：

物品的集合，每个重量Wi, 价值Vi

背包的容量W

输出：

找到一组使得背包中价值最大，并且总重量基本上等于W的物品集合

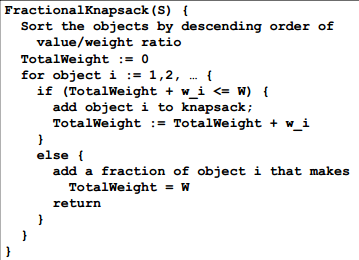
e.g.

有三个物品：



部分背包问题Fractional Knapsack:相较于0/1背包问题，此背包问题中的物品可以部分存在于背包中，例如我可以只放50%的某件物品在其中

因为物品可以拿部分，所以我们每次选择value/weight最大的的物品



首先要对value/weight进行降序排序

Chapter 9 Elementary Graph Algorithms——图论的一部分

graph G = (V, E) consists of:

V: a set of vertices 节点的集合

E: a set of edges joining the vertices 边的集合

图的种类：

有向图和无向图Undirected graphs & directed graphs

节点的度数 degree:节点的边数number of adjacent edges

路径Path: : a sequence of edges between two vertices

环路Cycles: A path with same starting and finishing vertex

连通图Connected graph: a (undirected) graph where any two vertices reachable by a path

树Tree: a connected graph with no cycles

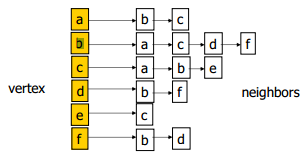
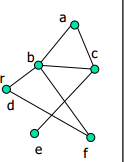
连通图满足以下条件：

n – 1<= m <= n(n – 1)/2n是节点数，m是边数。因为是连通图，所以每个节点至少连接一个节点，最多连接n-1个节点

节点的度数之和一定是边长的两倍。因为一条边是一个节点的入度，是另外一个节点的出度。

图在计算机中的存储方法：

1.邻接表Adjacency list

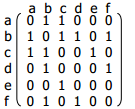


先写出所有的节点，然后列出每个节点的邻节点作为一条记录。

空间复杂度是O(n+m) n个节点加上m个相邻节点

如果要检查两个节点是否相邻，则时间复杂度是O(deg)。搜索一个节点的所有邻节点，查找另一个节点是否在其中。

2.邻接矩阵



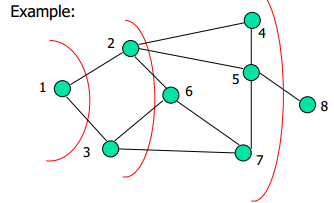
空间复杂度是O(n^2)

检查两个节点是否相邻的时间复杂度是O(n)，横向扫描一行即可得出结果。

以上两种表示方法均可以表示有向图和带权图。

Graph Traversal图的遍历：广度优先搜索BFS，深度优先搜索DFS

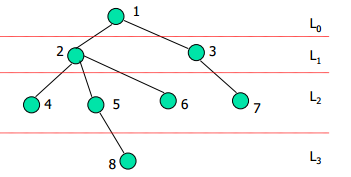
Breadth First Search (BFS)



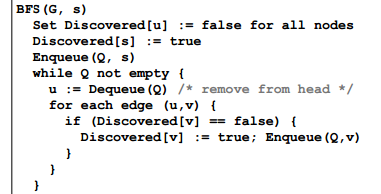
解释：我们可以将每一层的节点放入一个队列Queue（First in First out）实现算法。

我们先选择一个初始节点，查找它的相邻节点。如图，我们将1节点设置为初始节点，作为队列的第一个元素。将1从队列中取出来（现在队列为空），接下来我们可以查找到1的相邻节点2,3，并将这两个节点放入队列。然后取出队列的第一个元素，是节点2，我们查找节点2的相邻节点4,5,6，放入队列之中。不会将1放入队列之中，是因为有一个标记数组，如果我们查找过这个元素，就标记它为true。我们将所有2未遍历过的相邻节点放入队列之中，再取出队列的第一个元素，就是3。我们遍历节点3的所有没有遍历过的节点7，把7放入队列之中。现在队列是4,5,6,7，我们已经查找到了第三层。然后取出队列的第一个元素，4，查找它没有被遍历过的相邻节点，以此类推，直到队列为空，查找结束。BFS是一层一层查找节点。

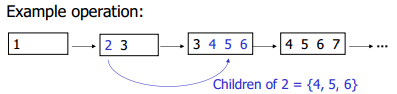
用一棵树更清晰地表现上述的过程：



BFS的伪代码：



解释：s是初始节点，先将s放入队列之中，进入循环，循环退出聊天是Q不空（如果Q空的话，表示下一层没有相邻的节点，也就是所有的节点遍历结束）。每一层的节点挨个出队列，并搜索出队列的每个节点的相邻节点，放入队列之中。

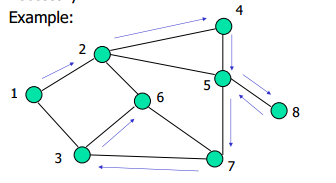


时间复杂度：

因为每个节点都要进一次队列，出一次队列，所以循环至少要执行n遍。

如果使用邻接表作为存储图的方式，因为要检查遍历每个节点的相邻节点，所以时间复杂度是。

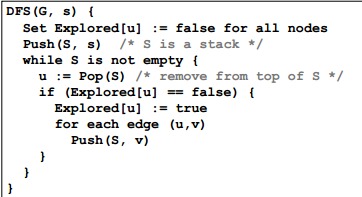


深度优先搜索Depth First Search (DFS) 

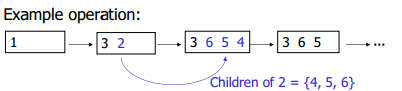
思路：DFS顾名思义，尽可能向深处搜索，直到没有下一个邻接节点，然后回溯（因为有这个操作，所以使用的是栈stack,先进后出）到上一个节点，尝试新的路径。

例如上图，我们将1作为初始节点，作为栈底base元素，首先随意选择相邻节点，这里为了方便，我们可以按序号大小选择节点2，压入push栈中（每次选择的元素都要压入栈中，并且标记已经遍历过了）。我们看到2的相邻节点是1,4,5,6，因为1已经选择过了（通过一个标记数组得知）我们选择4，并且将4压入栈中。4的相邻节点是2，5，同样的道理，2已经被选择过了，我们选择5，把5也压入栈中。5的相邻节点是2，4，7,8，选择7（图中是选择8，但是无所谓，选择7更好理解），然后选择3,再选择6. 我们发现选择6之后无法再进行进一步地选择了，因为6的相邻节点2,3,7都已经被选择（都在栈中）。于是我们要进行回溯，开始弹栈pop操作。现在栈顶元素是6，弹出6。现在栈顶元素是3，我们查找3的相邻节点，发现都已经遍历过了，再把3弹出，现在栈顶元素是7，查找7的相邻节点，也都遍历过了。弹出7，栈顶元素是5，发现5的相邻节点还有8没有遍历，于是将8压栈，8也找不到没有遍历的相邻节点了，弹出8，弹出5,4,2,1，直到栈空，遍历结束。

伪代码：



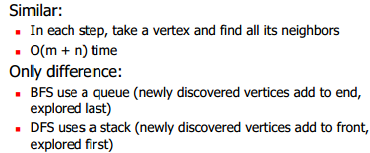
这个伪代码并没有优化，它会把一个节点的所有邻接节点压入栈中，如果已经被遍历过（Explored[u]=true）的节点，压栈之后直接弹栈，相当于没有任何操作。如果没有被遍历过的节点，则将它的邻接节点进一步做压栈操作。



时间复杂度：

每个节点都要将它的邻接节点压栈，所以压栈和弹栈操作加起来=2\*O(deg)=O(m)。标记数组的操作是O(n). 所以总共O(m+n)

DFS和BFS总结

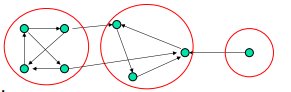


BFS是按顺序查找下去的，所以是先进先出的思想，使用的是队列queue。而DFS是尝试一条新的路径，如果走不下去了，就要往回走，进行新的尝试，所以是先进后出的思想，使用的是栈stack。

DFS和BFS都是用于连通图Connected，如果是非连通图，就要对每个独立的图都进行一次遍历。

强连通图strongly connected：任何两个节点之间都可达if any two vertices u and v are mutually reachable by some paths

strongly connected components (SCC)： 是一个子图，而且这个子图每个节点之间可达a graph are the subgraphs all of which are strongly connected



问题描述：

我们要判断一个图是否是强连通图

并且要找出它所有SCC

一个重要的性质：

如果{u,v}可达，并且{v,w}可达，那么{u,w}可达

问题解答：

我们可以选择一个初始节点s，通过BFS算法，找出s所有可达的节点，如果依旧有一些节点不在这个集合之中，说明这个图不是强连通图。

那怎么判断是强连通图呢？

答：我们根据BFS已经能找出s可达的所有点。我们选取可达点中的任意两点u和v，即{s,u},{s,v}可达，根据上面的性质，我们可以得到{u,v}可达。所以这个s可达节点集合中的任意两点可达，因此是强连通图。

Runtime: O(m + n)

查找SCC：对不在s可达节点集合中的集合再次进行BFS操作，直到所有的节点都被遍历到。执行BFS的次数就是SCC的个数。