INT202 期末笔记

Author :王泓澍

-打不过卷王就加入

Lecture1 复杂度

Algorithm analysis: Running time(time-complexity) of the algorithm. Space usage.

Interested in dependency of the running time or memory requirement on **size** of the input.

Pseudo-code: Provides more structured description of algorithm and allow high-level analysis.

Primitive operations: Assign value, method call, +-.., compare, indexing array, follow object reference, method return. Count the number of operations to analyze the running time.

Random Access Machine: only compute the number of primitive operations.`

Average VS Worst-Case Complexity: estimate average is hard, more interested in worst-case.

Recursive Algorithms: calling itself to solve sub-problems of a smaller size. Require a **base case** that can be solved without using recursion.

Recurrence equation: T(n) denotes the running time of algorithm on input of size n. Can use T(n-1 or T(n-2)... to characterize T(n).

Recuresive Algorithms are simpler but smaller subproblems might be solved repeatedly . Improve by non-recursive method. 用迭代(for loop) 取代递归。

我觉得这没啥好看的

Lecture2 渐进分析

- T(n)= 3, if n=1
- = T(n-1)+7, for n>=2

input每增加 1，time增长7，想要描述时间复杂度，将表达式转成closed form 封闭形式，也就是n而非T(n)的形式， 如此例T(n)=7(n-1)+3 = 7n-4.

Asymptotic notation 渐进分析：**estimates** number of primitive operations, and let us **compare** the running time of two algorithms.

Growth rate: is not affected by constant factors or lower-order terms.

Big-O: given functions f(n) and g(n), f(n) is O(g(n)) if there are constants c and n0 such that: f(n)≤c*g(n) for all n≥n0.

借助大O符号我们可以说，随着n趋向无穷大，在渐进意义上，n的一个函数“小于或等于”另一个函数。 2n+10 is O(n).

大O忽略低阶项和常数因子， f(n) is g(n) means that the growth rate of f(n) is no more than the growth rate of g(n).

大Ω和大θ， 大Ω： f(n) is asymptotically greater than or equal to g(n).

大θ：f(n) is O(g(n)) and is also Ω(g(n)), is asymptotically equal to g(n).

Space Complexity: concern with how the space needs grow, in big-Oh terms, as the size N of the input problem grows.

我觉得这也没啥好看的，会算就得了

Lecture3 栈，队列，列表

Stack栈: 插入对象和删除对象按照后进后出（last in first out Lifo)的原则，we only have direct accsess to the **last element** that was inserted.

栈具有两个基本方法：push(o)在栈顶插入对象o, pop()删除栈顶对象，并返回该对象，栈顶为空时操作出错。

栈简单实现：用一个具有N个元素的数组S实现，元素存储在S[0]~s[t]中，其中t为整数, 表示S中栈顶元素的下标。初始化t为-1，并用t判断栈是否为空。

栈的应用： 方法调用，arithmetic expressions using postfix notation(reverse polish notion).

Revese polish notation: 如xy+, 当遇到运算符时， 使用在运算符之前的两个operands。

Queue队列：先进先出（first in first out FIFO)原则， 对象可以被加在队尾，但只有队头的元素可以被移除。

队列的两个基本方法：enqueue(o) 将对象o插入队尾， dequeue()删除队头并返回，队列为空时报错。

队列简单实现：用具有N个元素的数组Q实现，用两个变量f(head)和r(tail)记录deque操作要删除的下一个候选项（第一个元素）的下标以及下一个数组可用单元的下标，如果数组为空，f=r。为防止数组越界，可以使用循环数组。

队列和multiprogramming: 用round-robin协议分配CPU时间，用队列来存储各个线程。

List表：元素存储在一个特殊节点node对象中，并有指针指向它前后的节点。

singly-linked list单链表： 存储一个next link指向表中下一个结点的引用（最后一个元素的next为null).

doubly-linked list双向链表：额外存储一个prev link指向表中前一个节点的引用。还可以增加特殊节点头节点和尾节点作为观察哨（sentinel).

插入操作 insertAfter(p,e): 创造node V, v.element=e, v.prev=p, v.next = p.next, p.next.prev=v, p.next = v， 共计四次指针更新

删除操作 remove(p): temp=p.element, (p.prev).next=p.next, (p.next).prev=p.prev, p.prev=null, p.next=null, return temp

Lecture4 树

树有一个special node r , 叫做root。 除了root, 树中每个元素都有一个父结点parent元素，0个或多个子结点child元素。如果u是v的父结点，那么称v是u的子child结点，具有同一父节点的两个节点称为兄弟（sibling). 如果一个节点没有子节点，则称为外部结点external， 如果一个结点有一个或多个子结点，则为内部节点internal。 外部节点也称为叶结点leaf。 结点的祖先ancestor要么为结点自身，要么为该结点的父结点的祖先。相反，如果结点u是结点v的一个祖先，则称结点v是结点u的后代descendent。

如果对于每个结点的子结点存在一个定义的线性顺序，则称树是有序的ordered。

二叉树binary tree是一棵有序树rooted ordered tree， 其中每个结点至多有两个子结点。

二叉树是真二叉树(proper)如果每个内部节点有正好两个子结点。对于每个内部结点，可以把每个子结点标注为左子节点left child或者右子结点right child。根在内部节点v的左子结点、右子结点上的子树称为v的左/右子树left/right subtree.

树的深度depth: v的深度就是v的祖先数（不含v自身），树T的根的深度为0. 可用递归求：如果v是根，则v深度为0。否则v的深度为1+父节点的深度。

树的高度（height) 等于T的一个外部结点的最大深度，树T的高度就是树T中根的最大高度。 递归的求： 如果v是一个外部结点，则v的高度为0. 否则，v的高度为1+v的（任何一个）子结点的最大高度。

☆树的遍历（Traversal):对树中所有结点的一种拜访。有基本的前序中序后序遍历。

前序遍历preorder traversal: 先访问T的根（自身）， 然后递归遍历以其子结点为根的子树。前序遍历产生树结点的一个线性序列，父结点总是在子结点之前(文件目录结构)。

后序遍历postorder traversal: 先访问以根的子结点为根的子树， 然后再访问根。

中序遍历二叉树inorder traversal: 如果v是一个内部结点，递归遍历v的左子树，然后访问结点v，再递归遍历v的右子树。

Lecture5 二分查找和二叉查找树

表示树的数据结构： 用数组表示二叉树，根在A[1], 左结点在A[2], 右结点在A[3]。 如果v是结点u的左子结点，那么p(v)=2p(u)， 如果v是结点u的右子结点，那么p(v) = 2p(u)+1.

Ordered dictionary : maintains an order relation for its elements, where the items are ordered by their keys(**non-decreasing order** of keys).

用向量或数组来存储key有序的数据会允许更快的查找，用external comparator for the keys(额外的索引来标识key)。

Binary search二分查找: search is done on a **decreasing range** of the elements. Looking for k in S, the current range of S we consider is defined as a pair of ranks:low and high. Initially, low=1 and high = n. 把key与s中的中间索引的key比较，根据比较的大小缩小搜索范围（确定key落在比左侧更小的区域或者右侧更大的区域）mid = (low+high)/2再向下取整， 有三个可能的case:

- k=key(mid), search is completely successful.
- k<key(mid), search continued with high = mid-1.

- k>key(mid), search continued with low = mid+1.

☆Complexity of Binary Search: 每个递归调用都可以在常数时间内完成，所以总的running time is proportinal to the number of recursive calls performed. The number of candidate items still to be searched in the array A is given by the value high-low+1. The number of remaining candidates is reduced by at least one half with each recursive call.

$$(mid-1)-low+1=\lfloor(low+high)/2\rfloor-low<=\frac{high-low+1}{2}$$

因为（L+H)/2-L = (H-L)/2

复杂度为T(n) = b如果n<2, 否则T(n/2)+b, b是个常数, this recurrence equation shows that the number of candidate items remaining after each recuresive call is at most n/2^i.

☆要分析这个算法的时间复杂度其实只需要知道最多有多少个recursive call能使得算法达到basecase也就是n<2，因为每一个recursive call都只执行常数时间。 假设有m个recursive call, 输入为n时，

$$n/2^m<1，m>logn.\text{因此}m=\lfloor logn\rfloor+1。$$

Binary search tree二叉查找树：每个internal node内部结点v存储字典D中的一个数据项（k,e），存储在v的左子树中的结点上的关键字小于或等于k，存储在v的右子树中的结点上的关键字大于或等于k。

二叉查找树的search:从根节点开始，和当前的node比较要查找的key，比当前结点的key小就找左子树，大就找右子树，如果到达leaf,则查找失败。查找的运行时间与树T的高度成正比，有n个结点的树高度最小为O(logn), 最大为O(n).

Insertion in a BST 插入操作：调用二叉搜索树的查找方法，直到找到leaf，若w是查找最终到达的leaf结点， 则用一个存储数据项(k,e)的新内部结点和两个外部子结点替代w。

☆Deletion in a BST 删除操作：最简单的情况为：

- 要删除的结点的两个子结点都为外部节点，则直接删除该结点并返回null.
- 若要删除的结点w的其中一个子结点z是一个外部节点，用z的兄弟结点代替w来重构T并返回该结点, 并同时从T中删除结点w和z。(就是用w的非外部节点的有东西的子结点替换掉w)
- 若要删除的结点w的两个子结点都是内部结点，则按中序遍历树T，找到w后的第一个内部结点y(找到比w大的第一个结点，也就是w的右子树中最左边的结点)， 将w中的元素存放在临时变量t中，并将y的数据项移到w中。删除y和y的左子结点x，**并用y的右子节点代替y**。最后返回存储在临时变量t中的元素。

AVL trees平衡树: Heigh-Balance Property: for any node n, the heights of n's left and right subtrees can differ by at most 1, 左子树和右子树的高度最多相差1。存储n个key的AVL tree的高度为O(logn).

☆证明：不直接找出一棵AVL树的上界，而是解决它的逆问题，即找出具有最少内部结点数的n(h)的AVL树，其高度为h。证明n(h)至少 呈指数增长，来导出存储n个关键字的AVL树高度为O(logn)。 n(1)=1, n(2)= 2, 对于h>=3, 高度为h且结点数最少的AVL树，满足其两颗子树都是结点数最少的AVL树，其中一棵高度为h-1,另一棵高度为h-2。 因此n(h)= 1+n(h-1)+n(h-2) for n>=3,

$$\text{因为}n(h-1) > n(h-2), \text{所以}n(h) > 2n(h-2), n(h) > 2^i n(h-2i),$$

因为已经知道h(1)和h(2)的base case， 选择i使其满足h-2i=1或2,

$$\text{即}i = \lceil h/2 \rceil - 1\text{，代入得}n(h) > 2^{H/2-1}\text{两边取对数得}h < 2logn(h) + 2$$

所以AVL树的高为O(logn)。

Lecture6 平衡树 2，4树

AVL Tree Insertion插入：插入操作就如同在二叉查找树的插入一样，查找直到找到一个leaf外部节点w，将这个外部结点w插入新数据项使w成为内部结点（再给w添加两个空的外部子结点）， 这个操作可能会破坏平衡，如原本高度为h-1和h-2的子树，经过这个操作可能变为h和h-2从而不平衡。

☆**修正不平衡操作：** 从新插入的w向树根查找直到找到第一个祖父结点z是不平衡结点的结点x，（也就是z是从w向根遍历遇到的第一个不平衡结点，z的左右子树高度不平衡，y是z的高度更高的子结点，y高度比其兄弟结点高2，x是z的孙结点也就是y的子结点。）使用三节点重构trinode restructuring:

- 根据**中序遍历**的顺序将x,y,z重命名为a,b,z， 将xyz的四个子结点按中序遍历的顺序命名为T0,T1,T2,T3。
- 用结点b来代替结点结点z
- 使a成为b的左子结点，并使T0 和T1分别成为a的左右子树。
- 使c成为b的右子结点，并使T2和T3分别成为c的左右子树。

☆AVL Tree Removal 删除：删除操作一开始如同在二叉查找树中的删除操作，假如操作导致了不平衡，则用三节点重构修复不平衡。从删除的子结点w开始往上向树根查找知道找到第一个不平衡结点z, z的左右子树高度不平衡, y是z的高度更高的子结点, y高度比其兄弟结点高2,☆**注意z不是w的祖先, x是y的高度更高的子结点，假如y的子结点高度一样，那么x为是与y在同一边的结点(假如y为左子结点，x就为y的左子节点)。**然后使用三节点重构，重构后可能会导致新的不平衡。

Multi-Way Search Tree 多路查找树: 每个内部结点最少两个子结点, 设v是树的一个结点，v有d个孩子（子结点）就称v是d结点, 并且v中存储了d-1个key-element元素, 树的性质如下：

- T的每个内部字结点至少有两个子结点，假设结点有d个子结点，则该结点为d结点，存储d-1个key-element item.

  结点的子结点为$v_1, v_2, v_3 \dots v_d$, 结点存储的键为$k_1, k_2 \dots k_{d-1}$.

- 在v1子树中的键key比k1小
- 在vi子树中的键key的值在ki-1和ki区间内(i=2,...d-1)
- 在vd子树中的键key比kd-1大

Multi-Way Inorder Traversal: 对于结点d,在中序遍历中遍历vi子树再遍历item(ki, value), 然后再遍历vi+1

Multi-Way Searching 多路树的查找: 对于具有子结点v1,v2...vd， 并存储键k1,k2...kd-1的结点v， 进行查找k的操作：

- k=ki(i=1,2...d-1), 查找成功
- k<k1, 继续查找v的子树v1
- ki-1<k<ki(i=2,3...d-1), 继续查找v的子树vi
- k>kd-1, 继续查找v的子树vd

(2,4)树: 一种特殊的多路查找树并具有以下性质：

- Node-Size Property: 每个内部结点最多有四个四结点
- Depth Property: 所有的外部节点在同一深度
- 根据结点的子结点数量， 内部结点可以被称作2-node, 3-node, 或4-node

☆(2,4)树高度定理：一个存储n个结点的（2，4）树高度为O(logn)。证明：

- h是存储了n个items的(2,4)树的高度
- 因为每一个深度i=0, 1... h-1至少有2^i个items，至多有4^i个items， 并且深度h上没有item,

  $n \geq 1 + 2 + 3 + \dots + 2^{h-1} = 2^h - 1$, 因此$h <= log(n+1)$。

☆**(2,4)树Insertion插入**: 进行查找直到leaf结点, 将新的item (k,o)插入这个leaf结点的父节点v中， 这个操作维持了深度的性质但是可能会违反大小的性质，比如4-结点插入后会变成5-结点并造成上溢。 为了修正结点v处的上溢，在结点v处进行分裂split操作：

- 用两个结点v' 和v''代替v，其中
- a. v'是一个3-结点， 它的子结点为v1，v2，v3， 并存储关键字k1和k2。

b. v''是一个2-结点，它的子结点为v4和v5，并存储关键字k4。

- 如果v是T的根，建立一个新的根节点u; 否则，设u是v的父节点。
- 将关键字k3插入u中，使v'和v''成为u的子结点，满足如果v是u的子结点，那么v'和v''分别成为u的子结点i和i+1。
- 可能还会造成父节点u的上溢，若造成上溢则再对u进行分裂操作
- 分裂操作数受限于树的高度，因此插入操作的总复杂度为O(logn)

☆**(2,4)树 Deletion删除**：将所有情况尽可能地归约到一种情况，也就是要被删除的数据项存储在其子结点是外部结点的结点v中。 假如要删除的数据项存储在只有内部子结点的结点z的第i个数据项（ki,oi)中。

- 则在z的第i+1个子结点的树中找到最左边的内部结点v
- 将z的数据项（ki,oi)与v的第一个数据项交换， 使要删除的数据项子结点为外部结点
- 最后简单的从结点v删除该数据项并删除v的第i个外部结点。

删除操作可能会违反大小性质，如v以前是个2-结点，删除之后会变成没有数据项的1-结点造成underflow下溢。为了修正下溢：

- Case1:如果v的相邻结点都是2-结点，那么进行**合并(fusion)**操作，将v与兄弟结点合并，建立新结点v'， 并从v的父节点u中移动一个关键字。（从父节点存储的关键字中抓下来一个存放在新合并的结点v'中）
- Case2: 检查v的一个直接近邻兄弟结点是否是3-结点或者4-结点。如果找到这样一个兄弟结点w，那么进行**转移（transfer)**操作， 将w的一个子结点转移到v中，将w的关键字转移到v和w的父结点u中，并将父节点u的关键字转移到v中。

结点v处的合并操作可能会在v的父节点u处引起新的下溢，这反过来又会触发u处的转移或合并。但最终删除操作受限于树的高度，总的复杂度为O(logn)。

Lecture 7 优先队列和排序算法

Priority Queue优先队列： 存储元素的容器，每个元素有一个key决定它的优先级。

用优先队列排序 PQ Sorting: 将所有元素逐一添加到空的优先队列中，之后再一个一个的从优先队列remove 最小元素。

两种Binary Tree: **full**:如果每一个结点没有子结点（是一个leaf结点）或者有两个子结点。**Complete**: 如果除了最后一层外所有层都是满的，并且最后一层所有的结点都排列在左侧。

用Heap堆结构实现优先队列：heap要求所有元素和他们的键存储在**complete** Binary tree中，也就是除了最后一层都为满的，且最后一层的结点都排列在左侧。

Min-Heap的性质：在一个堆T中，对于除根之外的每个节点v，存储在v中的关键字大于或等于存储在v的父结点中的关键字。

用数组来实现堆：指针从1开始，有一个last变量表示对堆中最后一个结点的引用，对于任何一个给定结点，假如它的位置在i:

- 它的左子结点假如存在的话在[2*i]
- 它的右子结点假如存在的话在[2*i+1]
- 它的父结点在[i/2]

存储n个关键字的堆T的高度为$h = \lceil log(n+1) \rceil$

☆高度证明：因为T是complete完全的， 对于depth深度 i=0,1,2...h-2的每一层都有2^i个结点，深度为h-1的层也就是最后一层至少有一个结点， n>=1+2+4+...+2^(h-2)+1, 根据等比数列1+2+...2^(h-2)= 2^(h-1)-1，

所以$n \geq 2^{h-1}$，$h \leq logn + 1$

☆堆-优先队列插入 insertion: 向堆中插入key k的操作：

- 找到数组中下一个空的元素z（last+1位置）， 也就是还未填满的最左外部结点z
- 将k存储在z并将z变为内部结点
- 用**swim或Up-heap bubbling**向上冒泡来恢复heap-order
- up-heap算法将k不断地和它的父节点交换直到达到根结点或者当k的父节点小于或等于k的时候终止， 因为堆高为O(logn)， 复杂度为O(logn)

☆堆-优先队列删除 Removal: 将堆顶元素删除的操作：

- 将根结点保存后与最后一个结点w交换
- 将交换后的w结点(也就是原来的根节点)存储的值删除，并将w变成外部节点leaf
- 用**sink或Down-heap bubbling**来恢复heap-order
- down-heap算法不断地将k与其较小的子结点比较（可以有不同实现比如默认往左子结点下沉，但可以比较两个子结点的大小来选择下沉）如果k比该子结点大则交换，直到到达外部节点leaf或者k比它的两个子结点都要小，因为堆高为O(logn)，复杂度为O(logn)。

存储n个items的堆-优先队列的性能：空间O(n), 插入和删除O(logn)，size,isEmpty,minKey,minElement都为O(1), 用优先队列排序heap-sort为O(nlogn)

Divide-and-Conquer分治法：分为Divide,Recur, Conquer三个步骤

- Divide: 如果输入大小特别小那么直接解决问题(base case)， 否则将输入分成两个或两个以上的包含不同元素的输入集合（拆分成子问题）
- Recur: 递归的解决以拆分出来的输入子集作为输入的子问题
- Conquer: 将子问题的解merge合并成原问题的解

☆**MergeSort归并排序：**

- Divide: 如果S中有0个或1个元素则直接返回S因为不需要排序， 否则(S至少有两个元素) 从S中把元素分别放入两个序列S1和S2中，S1和S2各包含大约S中的一半元素(n/2的向上或向下取整)
- Recur:递归的求解子问题S1和S2（对S1和S2递归的进行归并排序）
- Conquer: 归并有序序列S1和S2，使他们成为一个有序序列，并将其中的元素放回S

☆**归并算法的性能：**merge-sort tree的高度h为O(logn)，因为每一次递归都把输入减半，每一层进行排序的复杂度依然是O(n), 因为每一层依然还是有n个元素， 所以总的复杂度为层数*元素= logn*n =O(nlogn)

QuickSort快速排序：是一种具有随机性的基于分治法的排序算法：

- Divide: 随机取一个element X (称为pivot)来把S分为三部分：
  - L(less): 所有比X小的元素
  - E(equal): 跟X一样大的元素
  - G(greater): 比X大的元素
- Recur:递归的对L和G进行排序
- Conquer: 将排序完的L, E和G合并在一起

快速排序的性能：最糟糕的worst case发生在pivot这个切分元素是最大或者最小的元素的时候， 这时候L和G的大小一个为n-1另一个为0， 层数为n层，因为每一次都只能切出来一个元素， 总的运行时间n+(n-1)+(n-2)+...+2+1， 所以worst-case为O(n^2)

Lecture 8 分治法分析Master方法

Divide-and-Conquer分治法：分为Divide,Recur, Conquer三个步骤

- Divide: 如果输入大小特别小那么直接解决问题(base case)， 否则将输入分成两个或两个以上的包含不同元素的输入集合（拆分成子问题）
- Recur: 递归的解决以拆分出来的输入子集作为输入的子问题
- Conquer: 将子问题的解merge合并成原问题的解

分治法的分析: 一般的情况下我们用recurrence relation来分析分治法算法的运行时间，一般用T(n)来表示输入大小为n的运行时间。分治递归方程将T(n)与问题规模小于n的函数T(小于n的数)的值关联起来， 如T(n)= 2T(n/2)+bn, 并且具有一个 base case保证算法最后总能执行不用分治的最小问题。

第1种方法-迭代代换法**Substitution method**: 将T(n)函数不断地递归的应用于自身及子问题身上，通过不断地分治T(n)将其变为T(n)的子问题，再将子问题变为子问题的子问题....直到最后能达到base case， 将T(n)完全消去化成完全为n的形式。

如$T(n) = 2T(n/2) + bn = 2(2T(n/2) + b(n/2)) + bn = 2^2T(n/2^2) + 2bn = 2^iT(n/2^i) + ibn$

这个式子在$n/2^i = 1$时, 也就是$i = logn$时取到$T(1)$这个$basecase$将$T(n)$完全消除。$T(n) = bn + bnlogn.$

第2种方法-主方法Master method: 这是一种通用的方法，只要满足它所要求的条件，就能直接求出递归方程的时间复杂度。

a>=1,b>=1, f(n)是一个关于n的函数或表达式，T(n) = a T(n/b)+f(n), 算法的时间复杂度有以下几种情况：

$Case1:$ 如果$f(n) = O(n^{log_ba-e})$并且常数$e > 0(f(n)$多项式的小于$n^{log_ba})$，那么$T(n) = \Theta(n^{logba})$

$$Case2:\text{如果}f(n) = \Theta(n^{log_ba}),\text{那么}T(n) = \Theta(n^{log_ba}lgn)$$

$$Case3:\text{如果}f(n) = \Omega(n^{log_ba+e})\text{且常数}e > 0(f(n)\text{多项式的大于}n^{log_ba})$$

$$\text{并且要求}af(n/b) \leq cf(n)\text{满足}c < 1\text{和所有的更大的}n\text{，那么}T(n) = \Theta\text{（}f(n)\text{）}$$

多项式的更小：f(n) is polynomially smaller than g(n) if

$$f(n) = O(g(n)/n^e)for\ some\ e > 0.$$

多项式的更大：f(n) is polynomially larger than g(n) if

$$f(n) = \Omega(g(n)n^e)for\ some\ e > 0.$$

Lecture 9 贪心和动态规划

贪心算法：应用于优化问题， 贪心算法总是试图找出当前看起来最佳的选择， 也就是不断地选择局部最优解并希望最终这些局部最优解能带来全局最优解。贪心算法并不总是能求出问题的最优解，但有些问题采用此方法确实会得到最优解， 称这样的问题具有贪心选择greedy-choice 性质， 这个性质说明从一个良好定义的配置开始，通过一系列局部最优选择，可以得到全局最优解。

{0，1}背包问题：{0，1}背包问题并不能保证用贪心算法找到全局最优解， 因为贪心看见能放进去没超过w的就拿，拿到的不一定是最优解。

Fractional背包问题：假如物体可以被分成任意部分， 且具有benifit b 和权值w, 切分后benifit和权值都为切分的比重。 那么可以采用贪心的方法：先对每个物体进行切分，列出每个物体单位权重下的benifit**(value index = bi/wi)**，单位权重下最高的benifit(最高value index)的物体根据贪心法优先装入背包（用max-heap)， 等到装到溢出的时候，将要溢出的物体切片切成能放进背包的片塞入背包。

Fractional背包问题的复杂度为O(nlogn)， 因为使用一个max-heap来根据value index存储物品， 每一次对于max-heap的插入和删除操作耗时都是logn。

Interval Scheduling任务调度问题：给定带有开始时间和结束时间的n个任务， **要求最多的不冲突的task**，可以用最早结束时间将他们排序，然后用贪心算法每次选择结束时间最早的task调度。

Task Scheduling： 给定带有开始时间和结束时间的n个任务, **要求用最少的机器完全调度所有的任务**，可以用最早开始时间将他们排序，然后用贪心算法每次选择开始时间最早的，如果有机器有空闲就把这个任务分配给这个机器， 若无机器有空闲那么就new 一个machine。

☆**Dynamic programming动态规划**：与分治法类似，但是用指针指向在一个特殊的表中保存的已经计算过的值来替代重复的递归recursive call。

动态规划主要用于求解优化问题， 只要优化问题具有能够探索的某种结构， 就可以利用动态规划技术求解：

- 简单的子问题： 必须有一种方式能把全局优化问题分解成子问题， 每个子问题在结构上与原问题类似。
- Optimal substructure子问题的最优性: 原问题的最优解必定包括子问题的最优解。 如果包含子问题的次优解，则不能得到原问题的最优解
- Overlapping subproblems子问题的重叠性：公共子问题被多次递归的调用和计算，比如斐波那契数列中f(1),f(2)的反复被其他子问题调用， 这样的重叠在动态规划中能通过存储公共子问题的解来提升效率。

动态规划的基本思路：从底向上的解决问题， 做一个table来存放已经解决的子问题的答案，用这些保存的答案来解决更大的子问题。

✨{0，1}背包问题动态规划: S为原来的物品的序列， 定义一个Sk表示序列中前k个物品，S0=空， B[k,w]表示前k个物品在剩余w权重没装满的情况下的最大benifit。 B[0,w]=0, for each w<=W， 并且B[k,w] = B[k-1,w] if w<wk (当前这个物体weight超过了背包的容量w)， = max{B[k-1,w], bk+B[k-1,w-wk]} otherwise (假如当前这个物品放的进来，那么需要从现在的

w中减去wk，来寻找前k-1个物品在w-wk情况下最优的benifit值加上当前物品的bk，看看和不选择这个进背包的benifit哪一个更大)

Lecture 10 图，图遍历，最短路径

Graph 由一组顶点vertices(nodes)和一组连接顶点的边(edges)组成， G = (V,E)

Edges Types: 可以分为无向边undirected edge 和有向边directed edge。

图也可分为有向图（所有边为有向边）和无向图（所有边为无向边）

称由边连接的两个顶点为边的端点end vertice或endpoint. 如果两个顶点是同一条边的两个端点, 则称这两个顶点是相邻的adjacent. 如果一个顶点是边的一个端点, 则这条边依附（incident)这个顶点。 顶点的出边outgoing edge就是源点为这个顶点的有向边,顶点的入边incoming edge就是目的地为这个顶点的有向边。顶点的度degree为依附incident这个顶点的边数。 入度和出度为顶点的入边数和出边数。连接相同的两个顶点的边称为平行边parallel edges. 连接顶点自身的边为自环self-loop。Path是一个顶点和边交替组成的序列，在一个顶点开始并在一个顶点结束，每条边依附它的直接前驱和直接后继。 Simple path是结点和边都不重复的路径。 walk也是顶点和边交替组成的序列，在一个顶点开始并在一个顶点结束。 trail是没有重复边（但是可以有重复的顶点的）walk。 circuit是开始结点和结束结点都相同的walk(没规定不能重复)。Cycle是没有结点重复的circuit。

subgraph子图是一个顶点和边是原图的子集的图, spanning subgraph生成子图是有原图所有结点的子图。 图是connected连通的假如对于任意两个结点中间都有path. (任意两个结点都可以通过一系列路径相连)。 假如图G不是connected的， 它的连通子图为图G的连通分量connected components.

acylic图是没有任何circle的图， 树是connected acyclic undirected graph.

图的性质：

- 如果G是具有m条边的图，那么G中所有顶点的深度加起来为2m, 因为一条边连接着两个顶点，将所有顶点的degree相加=2*边数。
- 如果G是具有n个顶点， m条边的简单图(没有平行边和自环)。如果G是无向的，那么m≤n(n-1)/2， 如果G是有向的，那么m≤ n(n-1)。 因为一个结点最多和n-1其他节点相连，（n-1)+(n-2)+(n-3)+....1。

Tree是一个连通的无cycle图，森林是多个不连通的树。假设G是具有n个顶点， m条边的无向图：

- 如果G是连通的， 那么m≤n-1.
- 如果G是树，那么m=n-1.

- 如果G是森林， 那么m≤n-1.

Spanning tree生成树： 是一个spanning subgraph(有原图所有结点的子图) that is a tree. 生成树并不唯一除非本来的图就是个tree.

☆图的遍历：

深度优先Depth First Search: 采用回溯技术，travel as far as possible before we have to back up。 给定图的一个顶点V，先将V标记为访问过，对于V的所有incident的边， 假如边没有被访问过， 拿到这条边连接的另外一个顶点w，假如w没被访问过那么就把w和这条边标记为访问过，紧接着将w作为给定点再进行深度优先搜索， 假如w已经被访问过就把这条边标记为back edge。

```
for all edges e in G.IncidentEdges(v) do
  if unexplored(e)
    then w = G.opposite(v,w) 得到这条边连接的另外一个结点
      if unexplored(w)
        then label e as discovery edge, label w as visited
          DFS(G,w)
      else
        Label e as back edge
```

非递归的深度优先搜索：给定图的一个顶点V, 将V压入栈中，假如栈没空就得到栈顶元素u， 如果有边连接到没访问过的顶点w，就将w推入栈中并和边一起标记为已访问， 并把w的parent记为u， 否则将边标记为back edge。当所有边都访问过后，弹出栈顶元素u。

广度优先Breadth First Search: 给定图和一个顶点V，将V加入队列中并标记为已访问， 假如队列没空就删除并得到队头元素u， 对于u的每一条边和相连结点w，假如没访问过w，就将边和w标记为已访问，再将w加入队列。否则已经访问过的话就将边记为corss edge。

比较BFS和DFS: 在解答复杂连通性问题方面DFS更好， 同时DFS会产生一颗生成树满足所有非树边都为后边back edge(it is an edge(u,v) such that v is ancestor of node u but not part of DFS tree). BFS查找图的最短路径， 同时BFS会产生一颗生成树满足所有非树边都是交叉边cross edge（并没有祖先和子辈关系）

加权图Weighted Graph：每条边上都关联一个数值，称为边的权值weight. 一条路径的length(weight)就是这条路径上所有边的权值相加之和。

单源点最短路径Single-Source Shortest Path: 给定一个顶点v，找到顶点v到其他所有顶点的最短路径。

假如图中所有的边都没有负权重，可以采用一种基于贪心思想的带权重的广度优先搜索。也就是Dijkstra算法

☆Dijkstra算法：将G中顶点v的标记定义为D[u], 表示迄今为止发现的从v到u的最短距离的长度， 起初对于每一个u≠v，D[v]=0, D[u]=∞, 定义一个C作为顶点的云集初始为空并放入V，每一次选择一个不在C中且具有最小D[u]标记的顶点u放入C中，一旦把一个新顶点u放入C中，就更新与u相邻且在C之外的每个顶点z的标记D[z] (进行边的松弛)

边的松弛relaxation: if D[u]+w(u,z)<D[z] then D[z]=D[u]+w(u,z), 假如新发现的经过u的路径并不比原来的路径好，那么D[z]将保持不变。算法的核心思想是在初始化完成后每次把min heap中最小的顶点u给pop出来，对于u的所有连接顶点检查一下它们经过u的路径会不会更短，如果更短就更新。

对于带有负权重边的图，dijkstra算法会失败， 可以用Bellman-Ford算法，但是要求图是有向的（否则任意具有负权值的边可能直接蕴涵一个具有负权值得回路）这个算法确切的执行边松弛V-1次，因为假如没有负得cycle， 那么V到其他任何顶点得最短路径最多只会有V-1条边。

Lecture 11 最小生成树， 网络流

最小生成树Minimum Spanning Tree: 包含树的所有顶点并且这棵树上的总权值最小。

☆Prim算法： 一种贪心的寻找MST的方法， 主要思想类似于Dijkstra算法， 给定某个顶点V，定义顶点的初始云集C。然后在每次迭代中选择一条具有最小权值的边e=(v,u)它连接云集中一个顶点v与C外的一个顶点u并且没有创造cycle，并将u带入云集C中，直到创建完整个生成树。

☆Kruskal算法：也是一种贪心的方法， 不同于Prim算法从顶点来构建MST, Kruskal算法先对所有的边从小到大排序。 每一步都选择不会创造cycle的最小权重的边加入MST中。假如创造了cycle就丢弃掉， 直到所有的顶点都包括在了MST中。

这两种算法都使用了优先队列来实现

☆Boruvka算法：将每个顶点作为独立的component(独立的tree)，具体算法：

1. 输入是连通的无向的加权图

2. 将所有顶点初始化为独立的component，将MST初始化为空
3. 当有不止一个components存在时， 对每一个component找到一条连接其他component的权重最小的边，假如没加入过的话将边加入MST中

网络流Flow Network：

最大流问题是指从源点(source)开始最大量的运输给定商品到顶点t(汇点sink)的方法。

流网络flow network由以下几部分组成：

- 有向连通加权图G，其边上权值为非负整数，边e上的权值称为e的容量(capacity) c(e).
- G中两个不同的顶点s和t，分别称为源点(source)和汇点(sink)，满足s没有入边，t没有出边。

网络的流(flow)是给G中的每条边赋予的整数值f(e)，满足以下性质：

- 对于每条边e, 0≤f(e)≤c(e) （capacity）
- 对于不同于源点s和汇点t的每个 顶点v，Σf(入边)=Σf(出边)。

换句话说，一个流必须满足边容量的约束条件，并且满足对于不同于s和t的每个顶点v，流出v的总流量等于流入v的总流量。称f(e)为边e上的流， 流f的值表示为|f|， 等于从源点s流出的总流量， |f|=Σf(s的出边)， 也等于进入汇点t的总流量，|f| = Σf(t的入边)，流网络的最大流maximum flow是一个具有网络上所有流的最大值的流, 我们关心怎么样让|f|最大。

☆Cut割: 割用于把流网络中的顶点分为两个集合Vs和Vt， 满足s∈Vs并且t∈Vt， 称源点为u∈Vs,目的地为v∈Vt的边为前向边forward edge， 源点为v∈Vt,目的地为u∈Vs的边为后向边backward edge， 穿越割X的流f(X)等于割中前向边的流之和减去后向边的流之和， 换句话说f(X)是从割的Vs一侧流到割中另一侧Vt的**净流量**。割的容量c(X) 是所有前向边的容量的和（穿越割的任何流的上界）。

割的引理Lemma 1: N是流网络，f是N的一个流， 对于N的任何割X, f的值等于穿越割X的流， 即|f| = f(X)。 不论在什么地方切割流网络，分割出s和t，穿越那个割的流总是等于整个网络的流。

割的引理Lemma 2: N是流网络， X是N的一个割， 给定N的任何流f，穿越割X的流不会超过割X的容量，即f(X)≤c(X)。

结合引理1和引理2：

定理Theorem 1: 给定N的任意流f以及N的任意割X，f的值不会超过割X的容量， 即|f|≤c(X).

也就是说最大流的值不会超过最小割的容量。实际上是相等的。

剩余容量Residual capacity：假如e是从顶点u到v的一条边，流f的从u到v的剩余容量用Δf(u,v) = c(e)-f(e)表示， 从v到u的剩余容量则定义为Δf(v,u)=f(e)，直观上讲，剩余容量是从s向t推进流的过程中，f尚未完全利用的额外容量。从u到v的剩余容量是容量-已经存在的流的还未利用的部分，而反向的边的剩余容量是他归还给我们的流。

路径的剩余流量：为这条路径上所有边中最小的剩余容量。

增大路径Augmenting Path：指从source s到sink t的一条具有非零剩余容量的路径。也就是对于路径上的每一条边e，假如e是一条前向边那么f(e)<c(e)。 假如e是后向边那么f(e)>0.

引理 Lemma 3: 我们可以把增大路径上的剩余容量添加到这条增大路径现有的流上，得到另一个有效并且更大的流。

引理 Lemma 4: 如果网络中通不存在关于流f的增大路径，那么f就是一个最大流。 同时存在一个割X满足|f| = c(X)。

引理4和定理1可以得出定理2： 最大流的值=最小割的容量。

☆**最小割**：使得所有的前向边f(e)=c(e)， 所有的后向边f(e)=0的割。也就是所有的前向边的容量都利用完全并且没有后向边退回流量的割，这是这个流量网络所能达到的最大流量。

☆**Ford-Fulkerson算法**：把贪心法应用到增大路径的方法上。分阶段逐渐增大流的值每次沿着一条从源点到汇点的增大路径推进一定量的流。

- 起初每条边上的流为0
- 不断地重复直到网络中没有增大路径算法终止时：
    - 寻找一条增大路径
    - 计算这条增大路径的剩余容量的流量
    - 将这条增大路径剩余流量沿着这条增大路径推进（加入这条路径中）

Lecture 12 Bipartite Matching 二部图匹配

假如G是一个二部图，那么G可以被分成两个部分X和Y部分，图中的每条边连接X和Y中的一个结点。

Matching: 是一个边的集合，这些边不共享任何结点， 图中X或者Y部分的结点最多有一个另一部分的partner结点。

假如图中的结点没有任何matching中的边连接它，那它就是exposed(或者unmatched)。Matching is perfect if no vertex is exposed.

Maximum bipartite matching：寻找最大边数量的matching。

我们可以用network flow 流网络来reduce解决二部图匹配：

- 我们引进一个source node 并把它和所有X部分的结点相连接
- 然后引进一个sink node 把所有Y部分结点和它连接
- 现在问题变成了寻找这个图中的最大流
- 我们找到一条从source到sink的路径就把这条路径上所有边都反转
- 重复直到我们找不到从source到sink的路径
- 最终还是反转的边就是我们要找的matching

Lecture 13 数论和密码学

Greatest CommonDivisor最大公因子： 正整数a和b的最大公因子是整除a和b的最大整数，用gcd(a,b)表示。如果gcd(a,b)=1，称a和b是互素的relatively prime。

gcd(a,0) = gcd(0,a) = a， gcd(a,b)=gcd(|a|,|b|)

Euclidean欧几里得GCD算法：

引理：如果a,b,r,q都是整数， a=bq+r并且b≠0，那么gcd(a,b)=gcd(b,r)。

可以不断地递归调用gcd(b,a%b)直到b=0。

```
def gcd(a,b)

  输入非负整数a和b，a+b>0

 return gcd(b, a%b) if b>0 else a
```

给两个整数a和b，我们通常向找到两个整数s和t 使得：s*a+t*b = gcd(a,b)

☆扩展欧几里得算法：能计算gcd并同时计算s和t的值：

- r1=a; r2=b; s1=1; s2=0; t1=0; t2=1;
- 每一轮中 计算出q = r1/r2, r = r1-q*r2, s = s1-q*s2, t = t1-q*r2
- 然后更新r1=r2, r2=r, s1=s2,s2=s, t1=t2,t2=t

直到r2=0时， gcd(a,b)=r1, s=s1, t = t1

模运算：Zn表示小于n的非负整数集合： Zn = {0,1,...(n-1)} 称为模n的剩余集合。

模运算的性质：

- (a+b)mod n = [(a mod n)+(b mod n)] mod n
- (a-b)mod n = [(a mod n)-(b mod n)] mod n
- (a×b)mod n = [(a mod n)×(b mod n)] mod n

加法逆元： a+b = 0(mod n)

乘法逆元： a×b = 1(mod n), 0在Zn中不存在乘法逆元，假如a和n最大公因数≠1那么a在Zn中不存在乘法逆元， 假如最大公因数为1，那么有乘法逆元。

☆**扩展欧几里得算法可以找到b在Zn中的乘法逆元**，因为只有当b和n最大公约数为1时，b有乘法逆元， s×n+t×b = gcd(n,b)=1， 所以t×b mod n = 1

Zn* 乘法群multiplicative group：与n互素的1~n之间整数集合，如果n是素数，则Zn*包含1到n-1。

矩阵的模运算:

Single-variable Linear Equations: ax=b (mod n)可能没有解或者有有限的解， 假如gcd(a,n) = d，若d不能被b整除，就没有解。 若d整除b，则有d个解。

例子： 10x = 2 (mod 15), gcd(10, 15)=5, 因为5不能整除2， 没有解。

例子2： 14x=12 (mod 18)， gcd(14,18)=2， 因为2整除12，所以有两个解：

7x=6 (mod 9)， x = 6(7逆) (mod9), x0 = (6×4) (mod 9) = 6

x1 = x0+1×(18/2) = 15

Lecture 14 数论和密码学2

Modular Exponentiation: 根据模运算的性质可以拆很高次的为几个低次的相乘

$$3^{94}(mod\ 17), 94 = 64 + 16 + 8 + 4 + 2, 3^2 = 9, 3^4 = 81 = 13 = -4, 3^8 = (3^4)^2 = 16 = -1$$

$$3^{16} = (3^8)^2 = (-1)^2 = 1, 3^{64} = (3^{16})^4 = 1^4 = 1$$

$$所以3^{94} mod 17 = 1 * 1 * (-1) * (-4) * 9 = 36 mod 17 = 2$$

☆费马小定理

$$设p为素数，x是一个满足x\ mod\ p \neq 0的整数，那么x^{p-1} = 1(mod p)$$

推论： 设p为素数，对于每一个非零的Zp中的residue x，

x的乘法逆元为$x^{p-2}\ mod\ p$. 证明：$[(x\ mod\ p)x^{p-2}\ mod\ p]\ mod\ p = xx^{p-2}\ mod\ p = x^{p-1}\ mod\ p = 1$

注意：这里因为x在Zp中，所以x mod p = x，

☆欧拉函数： 一个正整数n的欧拉Φ函数（totient function）定义为小于或等于n且与n互素的正整数个数，即为Zn中存在乘法逆元的元素的个数，假如n是素数，则为n-1，Φ(p) = p-1. 假如n不是素数， 那么先列出n的所有素数因子，然后Φ(n) = n*(1-1/p1)*...(1-1/pn).

特殊情况当n=p1*p2， 也就是两个素数相乘时，Φ(n) = (p-1)(q-1)

n的欧拉函数还等于Zn*乘法群的size。 Zn*包含Φ(n)个元素。

☆欧拉定理：

设n是正整数，x是满足$gcd(x,n) = 1$的整数，那么$x^{\Phi(n)} = 1\ mod\ n$.

密码计算

M为明文plaintext， C为密文ciphertext.

对称加密算法：双方共享一个key K， K被用来加密和解密。

Subtitution cipher: 每一个字符在一个规则或者函数下被替换成别的字符：

加密就对字符进行函数运算$y = f(x)$, 解密就进行逆运算，$x = f^{-1}(y)$

One-time pad：用异或操作

公钥密码系统：给定消息M,加密函数E,解密函数D：

- D(E(M)) = M
- E和D都易于计算
- 从E导出D在计算上是不可行的
- E(D(M)) = M

☆RSA密码系统： 首先选择两个大素数p和q，设n=pq是其乘积，Φ(n) = (p-1)(q-1). 选择加密密钥e和解密密钥d满足：

- e和Φ(n)互素
- ed=1 (mod Φ(n))

也就是d是e mod Φ(n)的乘法逆元 ☆**用扩展欧几里得算法求**。 n和e的值构成公钥， 其中d是私钥。

加密：$C = M^e \ mod \ n$

解密：$M = C^d \ mod \ n$

例子： p=7,q=17, n=pq=7*17 = 119, Φ(n) = 6*16 = 96， e=5, d=77， 公钥(119,5), 私钥：77，

加密$M = 19$，$C = 19^5 \ mod \ 119 = 66$

解密66, $M = 66^{77} \ mod \ 119 = 19$