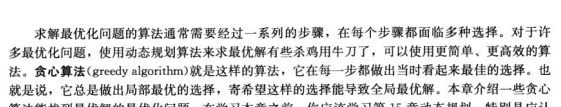
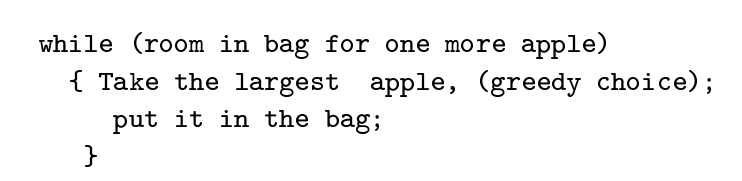
贪心算法，用来解Optimization problem//问你maximum或者minimum



通过局部最优解，试图达到全局最优解

例如，给你一个容量有限的包，怎么装最多的苹果



贪心算法就是永远拿最大的，

但实际上可能能装3个中等的，可能比两个最大的重

虽然并不永远是最优解，但在许多情况下可以达到最优解。

在那种特别难的算法里，它可以得到简单版本的相近算法

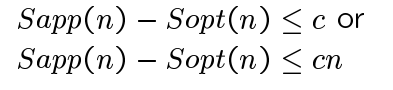
BST的例子告诉我们，这里用贪心算法，答案的差距是至少有n/4，至多却有logn，

因此并不算good approximation，

又给了个good approximation的例子：

Sapp代表渐进算法，Sopt代表optimal solution最佳算法，

如果差距是常数或者在n以下，我们可以接受



那么什么时候，我们能通过greedy approach得到

1.optimal solution最优解

2.good approximation， 近似解

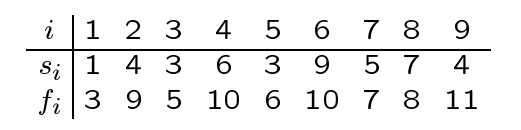
3.不合适not suitable(BST例子)

EX1.

Activity selection problem

选课问题，每个课有一个start time si与finish time fi

怎么选出最多的课



首先我们要明确限制，不同的课不能overlap重叠

1与5是compatible的 //**compatible 兼容的，指贪心算法中可以兼容的选择**

2与5就不行

如果是用贪心算法，

第一个选最早结束的，

剩下来选当前能做到的最早的，且尽快结束的

那么这是最优解吗

假设我们的解是e1,e2,ek

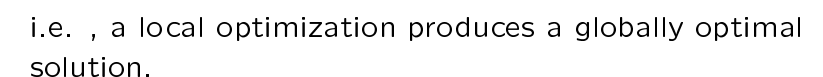
最优解是

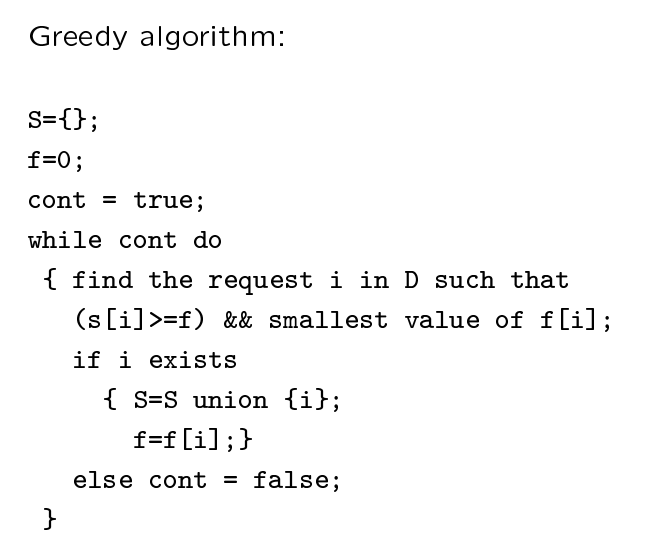
如果e1'≠e1

我们可以用E1替代E1'，因为我们第一个选的是最早结束的，我们才是最优值

剩下的同理，开始时间相同，我们总是选最早结束的

因此我们说这里的贪心算法成功生成最优解

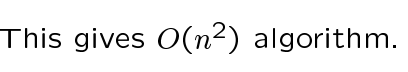




这就是贪心算法，如果满足情况，就UNION

现在算法我们需要每次都搜索整个array找到最先结束的

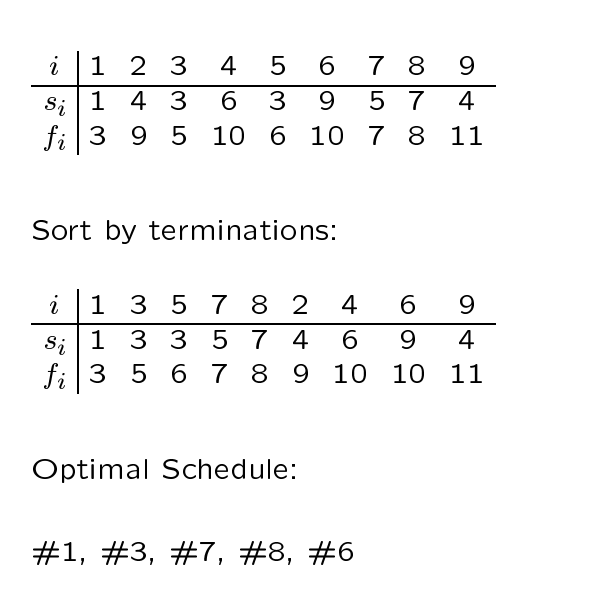
如果request没有sort，需要



优化版本，先用nlogn排序

然后对着sorted使用优化算法，一次for loop 循环就能O（n）搞定



.

注意要跟着final sort

Problem2 Scheduling

上一个的变种

现在我们需要把所有的request都排上，使用最小数量的房间

.

贪心算法思路

先按照开始时间排序

在第一个房间放第一个

如果没有冲突（第二个的开始时间在第一个结束时间后面），放入空闲房间

如果和所有房间都有冲突，开一个新房间

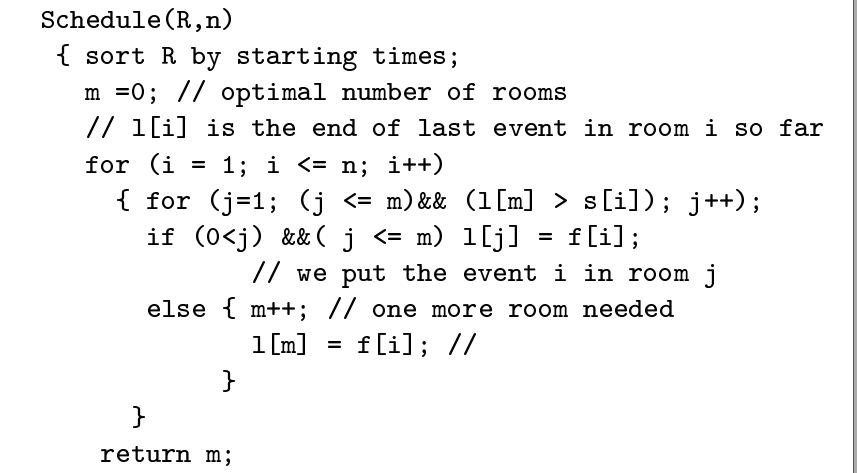
这是最优解吗

假设在同一个时间段，有m个冲突的，我们现在让它变成m+1格冲突的

那么无论如何，这第m+1个也是incompatible的

因此我们必须要m+1个房间

因此是最优解



算法有两个for，因此是N^2

我们能进一步优化吗

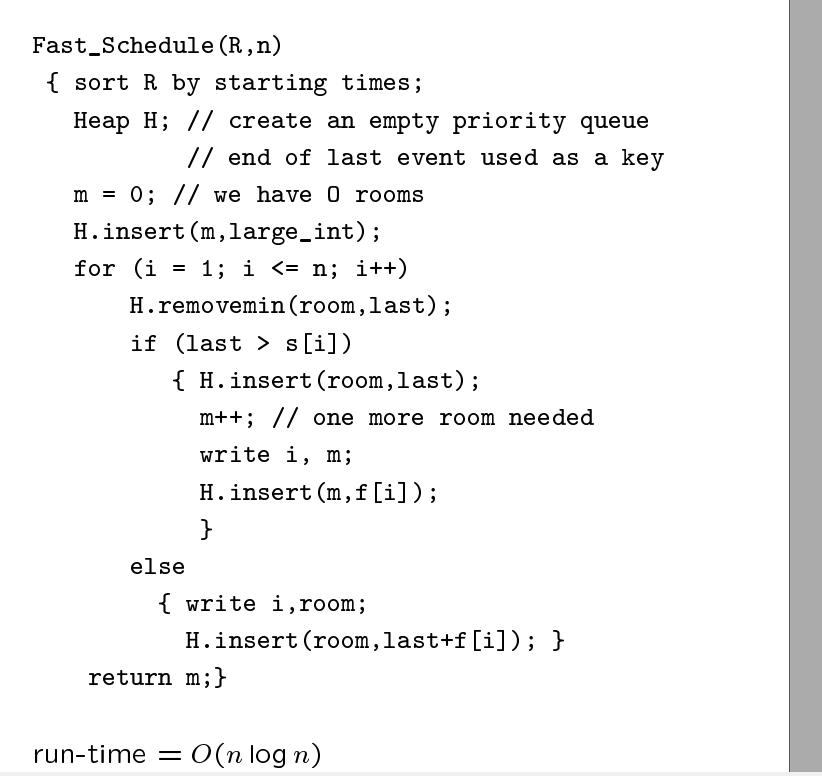
可以，他是一个一个线性搜索是否有房间满足下个request的

我们可以将request放在最早avaible的房间里

使用一个priority queue来存储每个房间的最新event的结束时间

用heap来装，heap的top必然是最早结束的（min heap）

在heap中，移除最小element的用时是O(logn



一个for loop，里面有一个logn

相乘=nlogn

3.General Scheduling Problem

BLAH BLAH说了一堆

最后greedy算法not work

greedy 算法什么时候work

他通过一系列的决定得到一个最终解

每一个决定都只能建立在local informaiton上

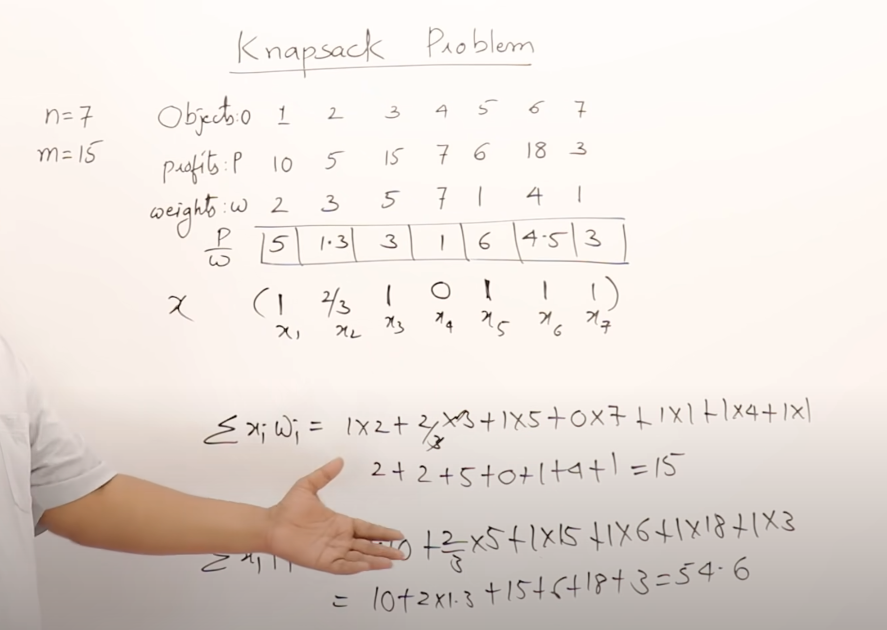
local decision将创建一个子问题，而子问题和原问题是same type的

4.Knapsack problem

knapsack problem有两种： 1. fractional knapsack problem ， 2. 0-1 knapsack

这个问题就是给你一个承重固定的包，给你一连串item，每个item有其profit 和重量，求包能装下的item，最大能多少profit

1.fractional knapsack problem, 在这种里面，item是可以分解的，我们可以取部分item，因此我们只要用利益除以重量，求单位质量 利益最高的item，简单地说，就想象成贵金属，啥贵拿啥



p/w 就是单位质量

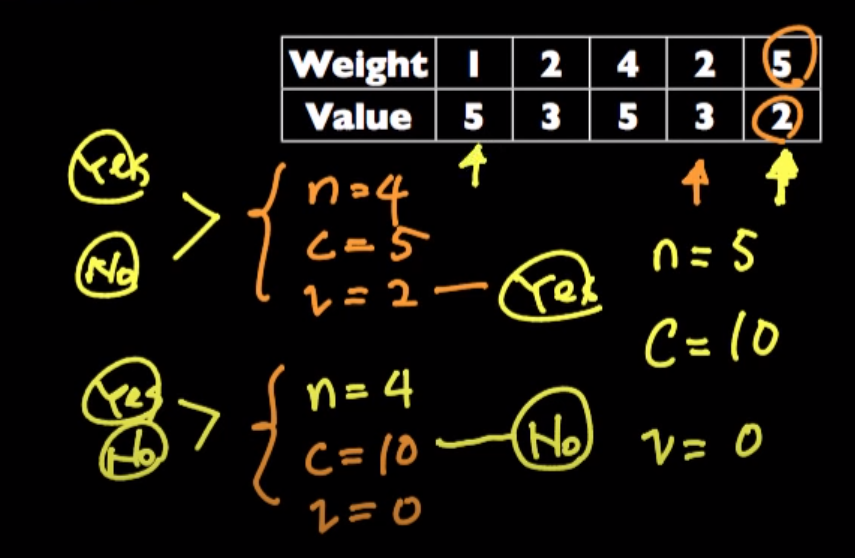
我们列个X表， 自己排序看看啥能拿1，最后质量不够的时候拿部分，质量用完拿0

2. 0-1 knapsack, 在这种里面，item是不可分解的，因此不能用greedy算法，而要用dynamic programming

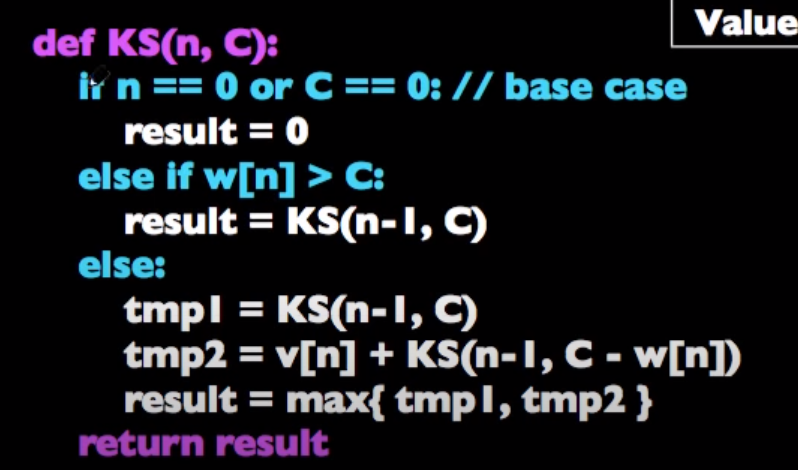
为啥不能用greedy，

假设我们前面还是拿单价最高的item，如果最后剩下一点空间，不能再放任何东西了

但是我们拿掉一个已经得到的高价item，剩下的空间能放两个中价item，那显然中价item这种拿法利益更高，greedy不是最优解



基础版的recursive解法，初始化一个pointer，先指向一个，看拿还是不拿，如果拿，扣除对应的C也就是背包容量，n-1也就是pointer往左移 ，最后会把所有可能性的排列组合都算一遍，不仅如此，在计算所有排列组合的时候，还会计算曾经计算过的排列组合，用时2^n



n与C等于0代表要么n已经没得选了，要么C=0空间满了

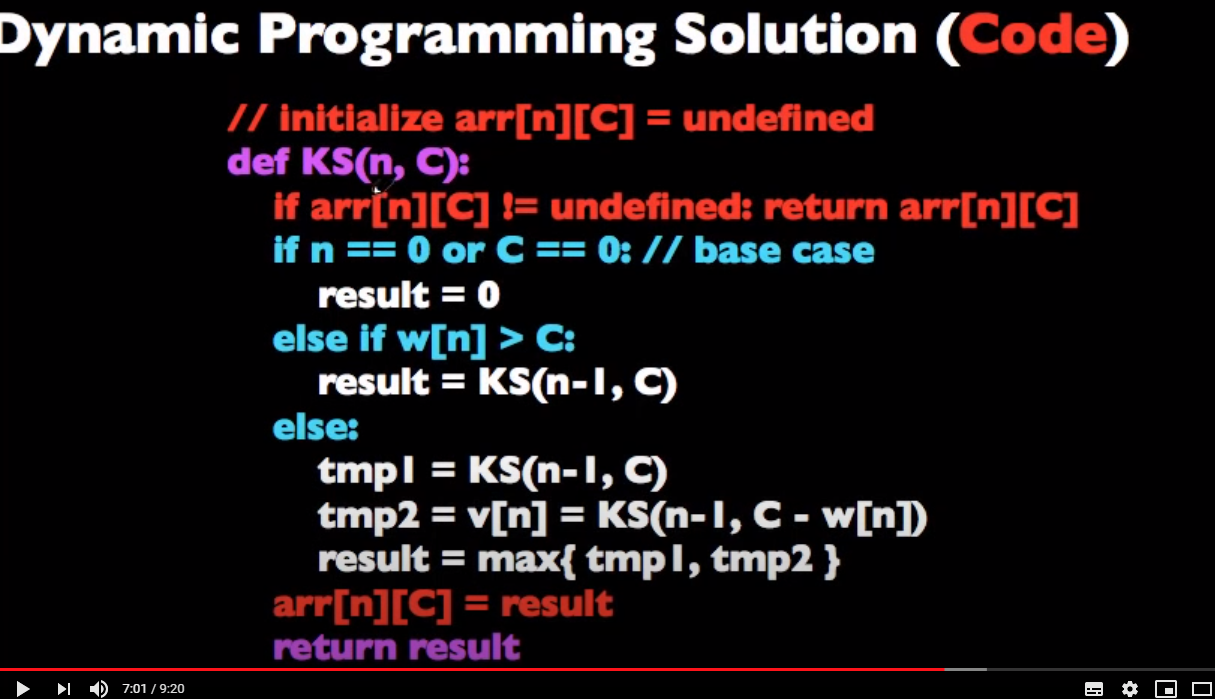
如果W[n]>c， 这个物体质量超过了，跳过

否则tmp1是选no

temp2是选拿了以后的YES

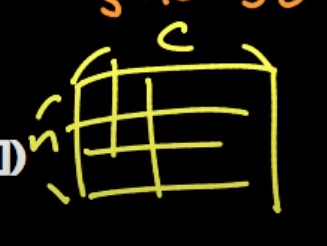
取MAX

显然计算tmp1与temp2是存在不必要的多余计算的



dynamic版本

就算所有的排列组合全部算出来，也就n\*C种

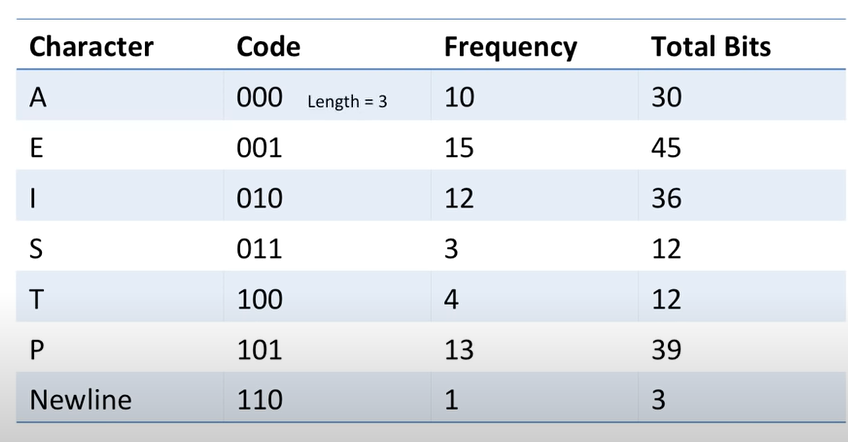


我们建立一个表，把各种都填进去，

Huffman codes

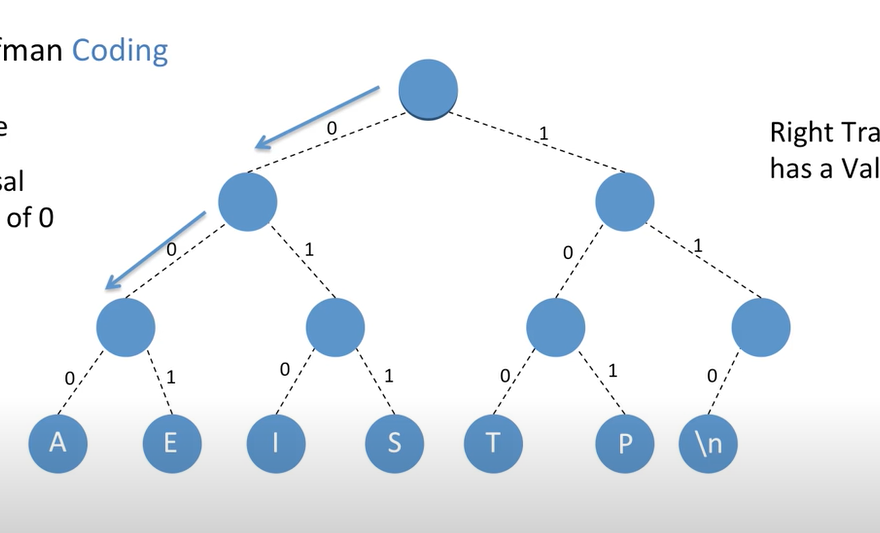
用来在不损失信息的情况下压缩文章大小，

通常来说，我们都用一套fixed-length code来表示char



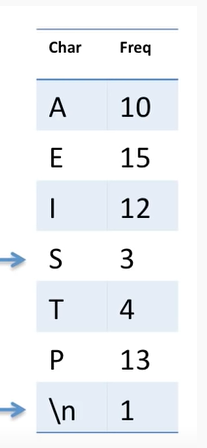
例如这个，我们都用length为3的code

构建成树的话左手边是0，右手是1



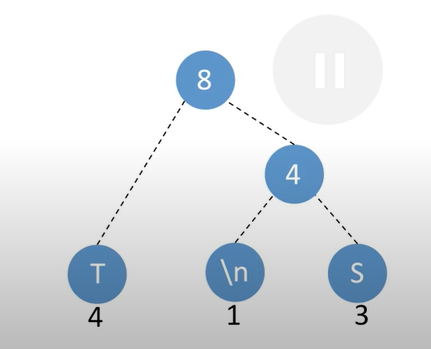
Huffman code的核心思路就是频率比较高的char，用简短的code表示，频率比较低的char，用长的char表示

思路：

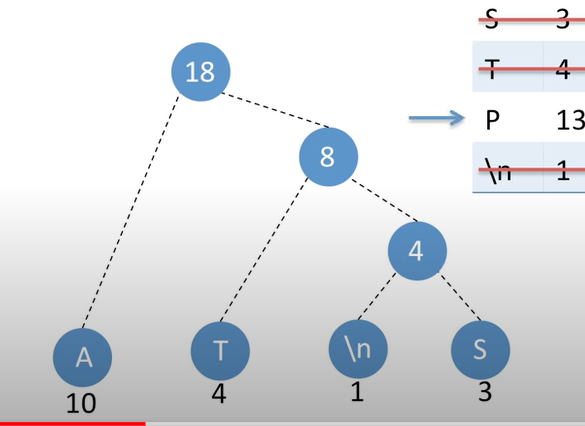
第一步先取两个最低频率的char

以他们为基础构建树，node相加作为父Node

第二步选第三小的，和我们构建的tree链接

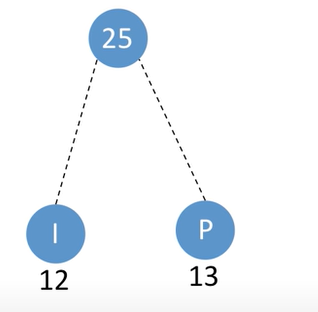
‘

只要先前一步生成的树的root<=接下来的node，就玩命添一个

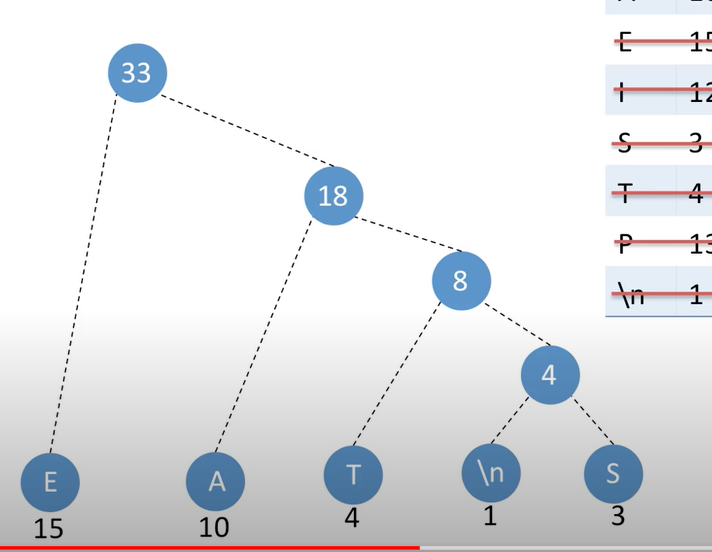


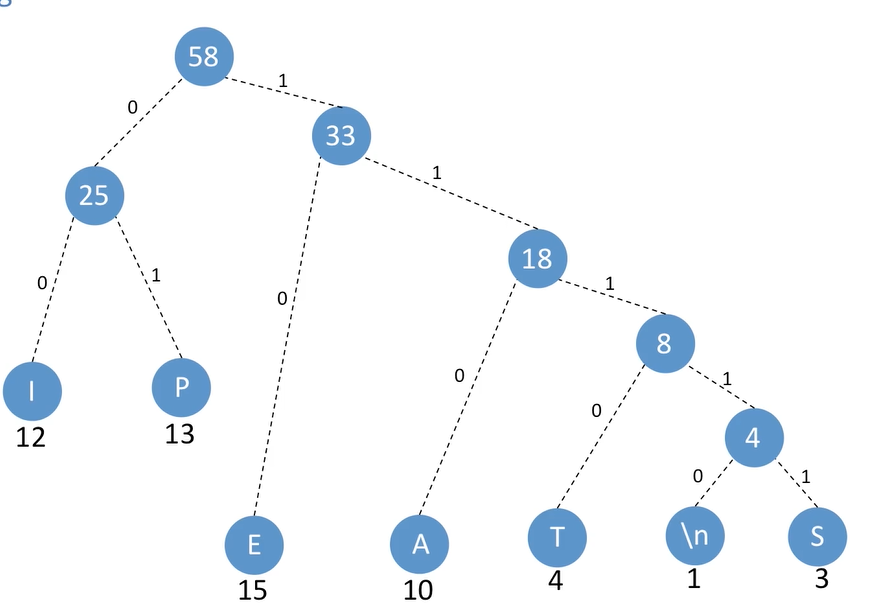
这时是剩下12 13,15，而18比他们大了，不能再添了

挑最小的两个，构建新tree

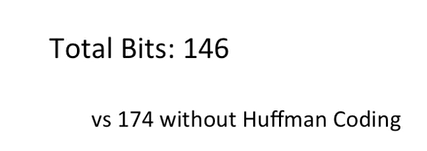
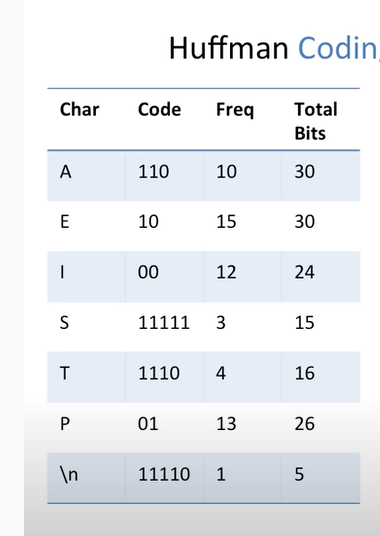


最后只剩下一个的时候，attach到root更小的 subtree //其实attach到那边都无所谓，，因为如果连接的时候，他们都是直接到root的，所用的code-length相同



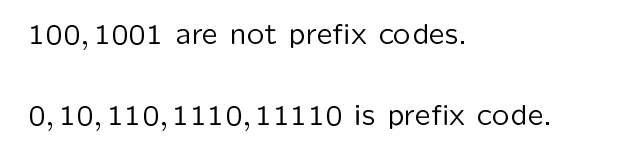


链接构建新tree以及对应code

压缩成功

Prefix-codes：前缀prefix不重复的code

例如



我们必须要使用prefix-codes否则我们不知道什么时候结束这个char

我们确保prefix code的方式就是使用full binary trees，也就是说char是leave, code=path to leave

//full binary tree代表每个node都有两个子node