ACM-ICPC 培训资料汇编

（3）

数据结构、动态规划分册

（版本号 **1.0.0**）

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求受训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新
2012 年 10 月

编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本分册大纲由黄李龙编写，内容由黄李龙、周洲、卢俊达、曹振海等分别编写和校核。

本分册内容大部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向acm@hrbust.edu.cn邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：http://acm.hrbust.edu.cn，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：http://acm.hdu.edu.cn

北京大学：http://poj.org

浙江大学：http://acm.zju.edu.cn

以下百度空间列出了比较全的国内外知名 OJ:
http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50

哈尔滨理工大学 ACM-ICPC 集训队

2012 年 12 月

目　录

第1章 数据结构

1.1 散列

参考文献：

《算法导论》

散列表 http://student.zjzk.cn/course_ware/data_structure/web/chazhao/chazhao9.4.1.htm

扩展阅读：

整数哈希介绍：http://www.cnblogs.com/napoleon_liu/archive/2010/12/29/1920839.html

各个字符串哈希函数比较：http://www.byvoid.com/blog/string-hash-compare/

编写：黄李龙　　校核：黄李龙

1.1.1 散列表的概念

散列方法不同于顺序查找、二分查找、二叉排序树及 B-树上的查找。它不以关键字的比较为基本操作，采用直接寻址技术。在理想情况下，无须任何比较就可以找到待查关键字，查找的期望时间为 O(1)。

1、散列表

设所有可能出现的关键字集合记为 U(简称全集)。实际发生(即实际存储)的关键字集合记为 K（|K|比|U|小得多）。

散列方法是使用函数 h 将 U 映射到表 T[0..m-1]的下标上（m=O(|U|)）。这样以 U 中关键字为自变量，以 h 为函数的运算结果就是相应结点的存储地址。从而达到在 O(1)时间内就可完成查找。

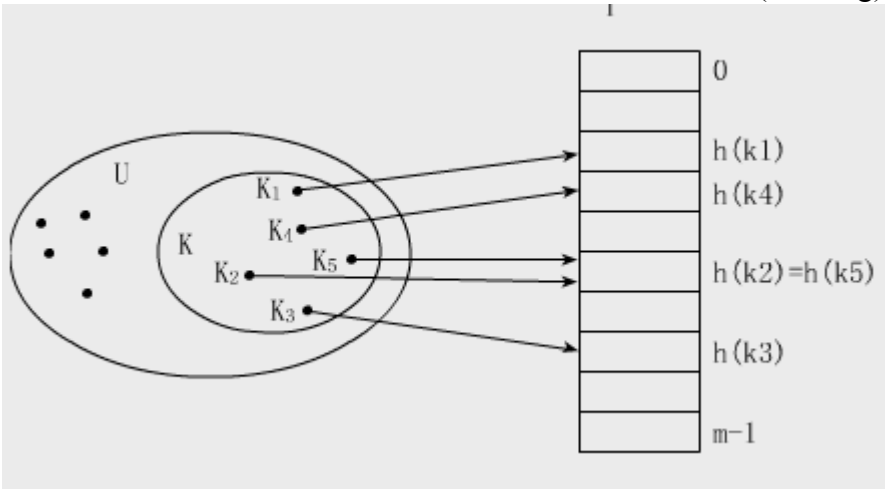
其中：

① h：U→{0，1，2，…，m-1} ，通常称 h 为散列函数(Hash Function)。散列函数 h 的作用是压缩待处理的下标范围，使待处理的|U|个值减少到 m 个值，从而降低空间开销。

② T 为散列表(Hash Table)。

③ h(Ki)(Ki∈U)是关键字为 Ki 结点存储地址(亦称散列值或散列地址)。

④ 将结点按其关键字的散列地址存储到散列表中的过程称为散列(Hashing)



用散列函数 h 将关键字映射到散列表中

3、散列表的冲突现象

（1）冲突

两个不同的关键字，由于散列函数值相同，因而被映射到同一表位置上。该现象称为冲突(Collision)或碰撞。发生冲突的两个关键字称为该散列函数的同义词(Synonym)。

【例】上图中的 k2≠k5，但 h(k2)=h(k5)，故 k2 和 K5 所在的结点的存储地址相同。

（2）安全避免冲突的条件

最理想的解决冲突的方法是安全避免冲突。要做到这一点必须满足两个条件：

①其一是|U|≤m

②其二是选择合适的散列函数。

这只适用于|U|较小，且关键字均事先已知的情况，此时经过精心设计散列函数 h 有可能完全避免冲突。

（3）冲突不可能完全避免

通常情况下，h 是一个压缩映像。虽然|K|≤m，但|U|>m，故无论怎样设计 h，也不可能完全避免冲突。因此，只能在设计 h 时尽可能使冲突最少。同时还需要确定解决冲突的方法，使发生冲突的同义词能够存储到表中。

（4）影响冲突的因素

冲突的频繁程度除了与 h 相关外，还与表的填满程度相关。

设 m 和 n 分别表示表长和表中填人的结点数，则将 α =n/m 定义为散列表的装填因子(Load Factor)。α 越大，表越满，冲突的机会也越大。通常取 α ≤1。

1.1.2 散列函数的构造方法

1、散列函数的选择有两条标准：简单和均匀。

简单指散列函数的计算简单快速；

均匀指对于关键字集合中的任一关键字，散列函数能以等概率将其映射到表空间的任何一个位置上。也就是说，散列函数能将子集 K 随机均匀地分布在表的地址集{0，1，…，m-1}上，以使冲突最小化。

2、常用散列函数

为简单起见，假定关键字是定义在自然数集合上。

（1）平方取中法

具体方法：先通过求关键字的平方值扩大相近数的差别，然后根据表长度取中间的几位数作为散列函数值。又因为一个乘积的中间几位数和乘数的每一位都相关，所以由此产生的散列地址较为均匀。

【例】将一组关键字(0100，0110，1010，1001，0111)平方后得

(0010000，0012100，1020100，1002001，0012321)

若取表长为 1000，则可取中间的三位数作为散列地址集：

(100，121，201，020，123)。

相应的散列函数用 C 实现很简单：

```
int Hash(int key){ //假设 key 是 4 位整数
  key*=key;  key/=100;  //先求平方值，后去掉末尾的两位数
  return key%1000;  //取中间三位数作为散列地址返回
 }
```

（2）除余法

该方法是最为简单常用的一种方法。它是以表长 m 来除关键字，取其余数作为散列地址，即 h(key)=key%m

该方法的关键是选取 m。选取的 m 应使得散列函数值尽可能与关键字的各位相关。m 最好为素数。

【例】若选 m 是关键字的基数的幂次，则就等于是选择关键字的最后若干位数字作为地址，而与高位无关。于是高位不同而低位相同的关键字均互为同义词。

【例】若关键字是十进制整数，其基为 10，则当 m=100 时，159，259，359，…，等均互为同义词。

（3）相乘取整法

该方法包括两个步骤：首先用关键字 key 乘上某个常数 A(0<A<1)，并抽取出 key.A 的小数部分；然后用 m 乘以该小数后取整。即：

$$h(key) = \lfloor m(key.A - \_key.A \_) \rfloor$$

该方法最大的优点是选取 m 不再像除余法那样关键。比如，完全可选择它是 2 的整数次幂。虽然该方法对任何 A 的值都适用，但对某些值效果会更好。Knuth 建议选取

$$A \approx (\sqrt{5} - 1)/2 = 0.61803398\ 87\ ...$$

该函数的 C 代码为：

```
int Hash(int key){
   double d=key *A;   //不妨设 A 和 m 已有定义
   return (int)(m*(d-(int)d)); //(int)表示强制转换后面的表达式为整数
  }
```

（4）随机数法

选择一个随机函数，取关键字的随机函数值为它的散列地址，即

h(key)=random(key)

其中 random 为伪随机函数，但要保证函数值是在 0 到 m-1 之间。

1.1.3 处理冲突的方法

通常有两类方法处理冲突：开放定址(Open Addressing)法和拉链(Chaining)法。前者是将所有结点均存放在散列表 T[0..m-1]中；后者通常是将互为同义词的结点链成一个单链表，而将此链表的头指针放在散列表 T[0..m-1]中。

1、开放定址法

（1）开放地址法解决冲突的方法

用开放定址法解决冲突的做法是：当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探查到开放的地址则表明表中无待查的关键字，即查找失败。

注意：

①用开放定址法建立散列表时，建表前须将表中所有单元(更严格地说，是指单元中存储的关键字)置空。

②空单元的表示与具体的应用相关。

【例】关键字均为非负数时，可用"-1"来表示空单元，而关键字为字符串时，空单元应是空串。

总之：应该用一个不会出现的关键字来表示空单元。

（2）开放地址法的一般形式

开放定址法的一般形式为： hi=(h(key)+di)％m 1≤i≤m-1

其中：

①h(key)为散列函数，di 为增量序列，m 为表长。

②h(key)是初始的探查位置，后续的探查位置依次是 hl，h2，…，hm-1，即 h(key)，hl，h2，…，hm-1 形成了一个探查序列。

③若令开放地址一般形式的 i 从 0 开始，并令 d0=0，则 h0=h(key)，则有：

hi=(h(key)+di)％m 0≤i≤m-1

探查序列可简记为 hi(0≤i≤m-1)。

（3）开放地址法堆装填因子的要求

开放定址法要求散列表的装填因子α≤l，实用中取α为 0.5 到 0.9 之间的某个值为宜。

（4）形成探测序列的方法

按照形成探查序列的方法不同，可将开放定址法区分为线性探查法、二次探查法、双重散列法等。

①线性探查法(Linear Probing)

该方法的基本思想是：

将散列表 T[0..m-1]看成是一个循环向量，若初始探查的地址为 d(即 h(key)=d)，则最长的探查序列为：

d，d+l，d+2，…，m-1，0，1，…，d-1

即:探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+1]，…，直到 T[m-1]，此后又循环到 T[0]，T[1]，…，直到探查到 T[d-1]为止。

探查过程终止于三种情况：

(1)若当前探查的单元为空，则表示查找失败（若是插入则将 key 写入其中）；

(2)若当前探查的单元中含有 key，则查找成功，但对于插入意味着失败；

(3)若探查到 T[d-1]时仍未发现空单元也未找到 key，则无论是查找还是插入均意味着失败(此时表满)。

利用开放地址法的一般形式，线性探查法的探查序列为：

hi=(h(key)+i)％m 0≤i≤m-1 //即 di=i

利用线性探测法构造散列表

【例 9.1】已知一组关键字为(26，36，41，38，44，15，68，12，06，51)，用除余法构造散列函数，用线性探查法解决冲突构造这组关键字的散列表。

解答:为了减少冲突，通常令装填因子α<l。这里关键字个数 n=10，不妨取 m=13，此时α≈0.77，散列表为 T[0..12]，散列函数为：h(key)=key％13。

由除余法的散列函数计算出的上述关键字序列的散列地址为(0，10，2，12，5，2，3，12，6，12)。

前 5 个关键字插入时，其相应的地址均为开放地址，故将它们直接插入 T[0]，T[10)，T[2]，T[12]和 T[5]中。

当插入第 6 个关键字 15 时，其散列地址 2(即 h(15)=15％13=2)已被关键字 41(15 和 41 互为同义词)占用。故探查 h1=(2+1)％13=3，此地址开放，所以将 15 放入 T[3]中。

当插入第 7 个关键字 68 时，其散列地址 3 已被非同义词 15 先占用，故将其插入到 T[4]中。

当插入第 8 个关键字 12 时，散列地址 12 已被同义词 38 占用，故探查 hl=(12+1)％13=0，而 T[0]亦被 26 占用，再探查 h2=(12+2)％13=1，此地址开放，可将 12 插入其中。

类似地，第 9 个关键字 06 直接插入 T[6]中；而最后一个关键字 51 插人时，因探查的地址 12，0，1，…，6 均非空，故 51 插入 T[7]中。

构造散列表的具体过程【参见动画演示 http://student.zjzk.cn/course_ware/data_structure/web/flashhtml/kaifang.htm】

聚集或堆积现象

用线性探查法解决冲突时，当表中 i,i+1，…，i+k 的位置上已有结点时，一个散列地址为 i，i+1，…，i+k+1 的结点都将插入在位置 i+k+1 上。把这种散列地址不同的结点争夺同一个后继散列地址的现象称为聚集或堆积(Clustering)。这将造成不是同义词的结点也处在同一个探查序列之中，从而增加了探查序列的长度，即增加了查找时间。若散列函数不好或装填因子过大，都会使堆积现象加剧。

【例】上例中，h(15)=2，h(68)=3，即 15 和 68 不是同义词。但由于处理 15 和同义词 41 的冲突时，15 抢先占用了 T[3]，这就使得插入 68 时，这两个本来不应该发生冲突的非同义词之间也会发生冲突。

为了减少堆积的发生，不能像线性探查法那样探查一个顺序的地址序列(相当于顺序查找)，而应使探查序列跳跃式地散列在整个散列表中。

②二次探查法(QuadraticProbing)

二次探查法的探查序列是：

hi=(h(key)+i*i)％m0≤i≤m-1//即 di=i2

即探查序列为 d=h(key)，d+12，d+22，…，等。

该方法的缺陷是不易探查到整个散列空间。

③双重散列法(DoubleHashing)

该方法是开放定址法中最好的方法之一，它的探查序列是：

hi=(h(key)+i*h1(key))％m0≤i≤m-1//即 di=i*h1(key)

即探查序列为：

d=h(key)，(d+h1(key))％m，(d+2h1(key))％m，…，等。

该方法使用了两个散列函数 h(key)和 h1(key)，故也称为双散列函数探查法。

注意：

定义 h1(key)的方法较多，但无论采用什么方法定义，都必须使 h1(key)的值和 m 互素，才能使发生冲突的同义词地址均匀地分布在整个表中，否则可能造成同义词地址的循环计算。

【例】若 m 为素数，则 h1(key)取 1 到 m-1 之间的任何数均与 m 互素，因此，我们可以简单地将它定义为：

h1(key)=key％(m-2)+1

【例】对例 9.1，我们可取 h(key)=key％13，而 h1(key)=key％11+1。

【例】若 m 是 2 的方幂，则 h1(key)可取 1 到 m-1 之间的任何奇数。

2、拉链法

（1）拉链法解决冲突的方法

拉链法解决冲突的做法是：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 T[0..m-1]。凡是散列地址为 i 的结点，均插入到以 T[i]为头指针的单链表中。T 中各分量的初值均应为空指针。在拉链法中，装填因子 α 可以大于 1，但一般均取 α ≤1。

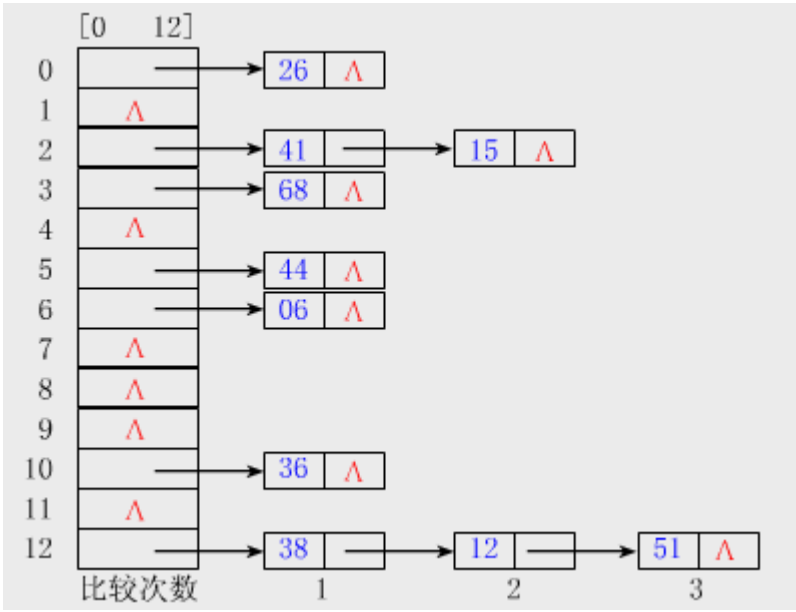
【例 9.2】已知一组关键字和选定的散列函数和例 9.1 相同，用拉链法解决冲突构造这组关键字的散列表。

解答：不妨和例 9.1 类似，取表长为 13，故散列函数为 h(key)=key％13，散列表为 T[0..12]。

注意：

当把 h(key)=i 的关键字插入第 i 个单链表时，既可插入在链表的头上，也可以插在链表的尾上。这是因为必须确定 key 不在第 i 个链表时，才能将它插入表中，所以也就知道链尾结点的地址。若采用将新关键字插入链尾的方式，依次把给定的这组关键字插入表中，则所得到的散列表如下图所示。

具体构造过程【参见动画演示 http://student.zjzk.cn/course_ware/data_structure/web/flashhtml/llf.htm】。



拉链法构造散列表示例

（2）拉链法的优点

与开放定址法相比，拉链法有如下几个优点：

(1)拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；

(2)由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；

(3)开放定址法为减少冲突，要求装填因子 α 较小，故当结点规模较大时会浪费很多空间。而拉链法中可取 α ≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；

(4)在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。而对开放地址法构造的散列表，删除结点不能简单地将被删结点的空间置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

（3）拉链法的缺点

拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

1.1.4 散列表上的运算

散列表上的运算有查找、插入和删除。其中主要是查找，这是因为散列表的目的主要是用于快速查找，且插入和删除均要用到查找操作。

1、散列表类型说明：

```
#define NIL -1 //空结点标记依赖于关键字类型，本节假定关键字均为非负整数
#define M 997 //表长度依赖于应用，但一般应根据。确定 m 为一素数
typedef struct{ //散列表结点类型
   KeyType key;
   InfoType otherinfo;  //此类依赖于应用
 }NodeType;
typedef NodeType HashTable[m];  //散列表类型
```

2、基于开放地址法的查找算法

散列表的查找过程和建表过程相似。假设给定的值为 K，根据建表时设定的散列函数 h，计算出散列地址 h(K)，若表中该地址单元为空，则查找失败；否则将该地址中的结点与给定值 K 比较。若相等则查找成功，否则按建表时设定的处理冲突的方法找下一个地址。如此反复下去，直到某个地址单元为空(查找失败)或者关键字比较相等(查找成功)为止。

（1）开放地址法一般形式的函数表示

```
int Hash(KeyType k，int i)
  { //求在散列表 T[0..m-1]中第 i 次探查的散列地址 hi，0≤i≤m-1
    //下面的 h 是散列函数。Increment 是求增量序列的函数，它依赖于解决冲突的方法
      return(h(K)+Increment(i))％m;  //Increment(i)相当于是 di
  }
```

若散列函数用除余法构造，并假设使用线性探查的开放定址法处理冲突，则上述函数中的 h(K)和 Increment(i)可定义为：

```
int h(KeyType K){ //用除余法求 K 的散列地址
   return K％m;
}

int Increment(int i){//用线性探查法求第 i 个增量 di
```

```
    return i; //若用二次探查法，则返回 i*i
  }
```

（2）通用的开放定址法的散列表查找算法：

```
int HashSearch(HashTable T，KeyType K，int *pos)
  { //在散列表 T[0..m-1]中查找 K，成功时返回 1。失败有两种情况：找到一个开放地址
    //时返回 0，表满未找到时返回-1。 *pos 记录找到 K 或找到空结点时表中的位置
    int i=0;  //记录探查次数
    do{
     *pos=Hash(K，i);  //求探查地址 hi
     if(T[*pos].key==K) return l;  //查找成功返回
     if(T[*pos].key==NIL) return 0;//查找到空结点返回
    }while(++i<m) //最多做 m 次探查
    return -1;  //表满且未找到时，查找失败
  } //HashSearch
```

注意：

上述算法适用于任何开放定址法，只要给出函数 Hash 中的散列函数 h(K)和增量函数 Increment(i)即可。但要提高查找效率时，可将确定的散列函数和求增量的方法直接写入算法 HashSearch 中，相应的算法【参见习题】。

3、基于开放地址法的插入及建表

建表时首先要将表中各结点的关键字清空，使其地址为开放的；然后调用插入算法将给定的关键字序列依次插入表中。

插入算法首先调用查找算法，若在表中找到待插入的关键字或表已满，则插入失败；若在表中找到一个开放地址，则将待插入的结点插入其中，即插入成功。

```
void Hashlnsert(HashTable T，NodeTypene w)
 { //将新结点 new 插入散列表 T[0..m-1]中
  int pos，sign；
  sign=HashSearch(T，new.key，&pos)；  //在表 T 中查找 new 的插入位置
  if(!sign) //找到一个开放的地址 pos
     T[pos]=new；  //插入新结点 new，插入成功
  else //插人失败
   if(sign>0)
      printf("duplicate key!")；  //重复的关键字
   else //sign<0
      Error("hashtableoverflow!")；  //表满错误，终止程序执行
 } //Hashlnsert

void CreateHashTable(HashTable T，NodeType A[]，int n)
 { //根据 A[0..n-1]中结点建立散列表 T[0..m-1]
  int i
  if(n>m) //用开放定址法处理冲突时，装填因子 α 须不大于 1
     Error("Load factor>1")；
  for(i=0;i<m；i++)
     T[i].key=NIL；  //将各关键字清空，使地址 i 为开放地址
```

```
    for(i=0;i<n；i++) //依次将 A[0..n-1]插入到散列表 T[0..m-1]中
      Hashlnsert(T，A[i])；
  } //CreateHashTable
```

4、删除

基于开放定址法的散列表不宜执行散列表的删除操作。若必须在散列表中删除结点，则不能将被删结点的关键字置为 NIL，而应该将其置为特定的标记 DELETED。

因此须对查找操作做相应的修改，使之探查到此标记时继续探查下去。同时也要修改插人操作，使其探查到 DELETED 标记时，将相应的表单元视为一个空单元，将新结点插入其中。这样做无疑增加了时间开销，并且查找时间不再依赖于装填因子。

因此，当必须对散列表做删除结点的操作时，一般是用拉链法来解决冲突。

注意：

用拉链法处理冲突时的有关散列表上的算法【参见练习】。

5、性能分析

插入和删除的时间均取决于查找，故下面只分析查找操作的时间性能。

虽然散列表在关键字和存储位置之间建立了对应关系，理想情况是无须关键字的比较就可找到待查关键字。但是由于冲突的存在，散列表的查找过程仍是一个和关键字比较的过程，不过散列表的平均查找长度比顺序查找、二分查找等完全依赖于关键字比较的查找要小得多。

（1）查找成功的 ASL

散列表上的查找优于顺序查找和二分查找。

【例】在例 9.1 和例 9.2 的散列表中，在结点的查找概率相等的假设下，线性探查法和拉链法查找成功的平均查找长度分别为：

ASL=(1×6+2×2+3×l+9×1)/10=2.2 //线性探查法

ASL=(1×7+2×2+3×1)/10=1.4 //拉链法

而当 n=10 时，顺序查找和二分查找的平均查找长度(成功时)分别为：

ASL=(10+1)/2=5.5 //顺序查找

ASL=(1×l+2×2+3×4+4×3)/10=2.9 //二分查找，可由判定树求出该值

（2） 查找不成功的 ASL

对于不成功的查找，顺序查找和二分查找所需进行的关键字比较次数仅取决于表长，而散列查找所需进行的关键字比较次数和待查结点有关。因此，在等概率情况下，也可将散列表在查找不成功时的平均查找长度，定义为查找不成功时对关键字需要执行的平均比较次数。

【例】例 9.1 和例 9.2 的散列表中，在等概率情况下，查找不成功时的线性探查法和拉链法的平均查找长度分别为：

ASLunsucc=(9+8+7+6+5+4+3+2+1+1+2+1+10)/13=59/13≈4.54

ASLunsucc=(1+0+2+1+0+1+1+0+0+0+1+0+3)/13≈10/13≈0.77

注意：

①由同一个散列函数、不同的解决冲突方法构造的散列表，其平均查找长度是不相同的。

②散列表的平均查找长度不是结点个数 n 的函数，而是装填因子 α 的函数。因此在设计散列表时可选择 α 以控制散列表的平均查找长度。

③ α 的取值

α 越小，产生冲突的机会就小，但 α 过小，空间的浪费就过多。只要 α 选择合适，散列表上的平均查找长度就是一个常数，即散列表上查找的平均时间为 O(1)。

④ 散列法与其他查找方法的区别

除散列法外，其他查找方法有共同特征为：均是建立在比较关键字的基础上。其中顺序查找是对无序集合的查找，每次关键字的比较结果为"="或"!="两种可能，其平均时间为 O(n)；其余的查找均是对有序集合的查找，每次关键字的比较有"="、"<"和">"三种可能，且每次比较后均能缩小下次的查找范围，故查找速度更快，其平均时间为 O(lgn)。而散列法是根据关键字直接求出地址的查找方法，其查找的期望时间为 O(1)。

1.1.5 散列表的应用

1.1.5.1 Hrbustoj 1287 数字去重和排序 II

用计算机随机生成了 N 个 0 到 1000000000（包含 0 和 1000000000）之间的随机整数（N≤5000000），对于其中重复的数字，只保留一个，把其余相同的数去掉。然后再把这些数从小到大排序。请你完成“去重”与“排序”的工作。

输入有 2 行，第 1 行为 1 个正整数，表示所生成的随机数的个数：
N
第 2 行有 N 个用空格隔开的正整数，为所产生的随机数。

输出也是 2 行，第 1 行为 1 个正整数 M，表示不相同的随机数的个数。第 2 行为 M 个用空格隔开的正整数，为从小到大排好序的不相同的随机数。

Sample Input
10
20 40 32 67 40 20 89 300 400 15
Sample Output
8
15 20 32 40 67 89 300 400

思路：题意很直白。数据是随机给出的，散列函数使用取余法，可以认为数据是平均分布的，冲突问题使用拉链法解决。当然也可以直接使用C++ STL的set容器，时间复杂度是$N\log_2 N$，题目所给时间能够承受。

C++代码：

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
using namespace std;

const int MP = 1007;
struct Node {
 int d;
 Node* next;
};
Node* pnd[MP+1];
Node nd[MP+1];
int n_cnt;
int a[1000+7+10];
int a_cnt;

int main()
{
```

```
    int n, d, p;
    while (EOF != scanf("%d", &n)) {
        memset(pnd, 0, sizeof(pnd));
        n_cnt = 0;
        a_cnt = 0;
        for (int i = 0; i < n; ++i) {
            scanf("%d", &d);
            p = d % MP;
            bool found = false;
            Node *pt = pnd[p];
            while (pt) {
                if (pt->d == d) {
                    found = true;
                    break;
                }
                pt = pt->next;
            }
            if (!found) {
                nd[n_cnt].d = d;
                nd[n_cnt].next = pnd[p];
                pnd[p] = &nd[n_cnt];
                n_cnt++;
                a[a_cnt++] = d;
            }
        }
        sort(a, a+a_cnt);
        printf("%d\n%d", a_cnt, a[0]);
        for (int i = 1; i < a_cnt; ++i) {
            printf(" %d", a[i]);
        }
        printf("\n");
    }
    return 0;
    }
```

1.1.5.2 POJ 2002 Squares

题意：在平面内给出 n 个点，问你这些点一共能组成几个不相等的正方形？

输入有多组测试数据，每组测试数据的第一行是一个整数 n，表示 n 个点，接下来 n 行，每行两个整数，表示一个坐标点，这 n 个点都不相同。（1<=n<=1000）

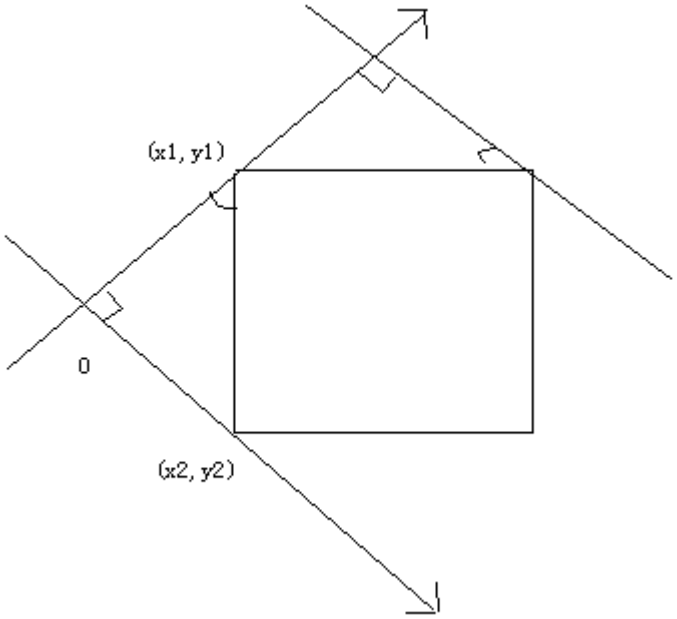
对于每组测试数据，输出组成的正方形数量。

思路：直接枚举四个点判断能否组成正方形肯定超时的，不可取。

普遍的做法是先枚举两个点，通过数学公式得到另外 2 个点，使得这四个点能够成正方形。然后检查散点集中是否存在计算出来的那两个点，若存在，说明有一个正方形。

先按他们的坐标从小到大排序，x 优先，之后是 y，这一步只是从插入顺序上优化一下之后的哈希查找，哈希函数使用取余法，把 x+y 和除 MOD 取余。

枚举正方形最左边的的两点坐标，然后计算出另外两点。证明就不具体展开了，可以参考下图，已知两个点，然后做出两个全等三角形。之后就得出结论(x1+|y1-y2|, y1+|x1-x2|),(x2+|y1-y2|,y2+|x1-x2|)。当然这只是一种情况，其他情况类似。



冲突的解决方法使用拉链法。

在枚举和统计的过程中，会重复统计。枚举方式的不同，统计结果也不一样，下面代码的枚举方式使得需要将统计的结果除以 2。

代码：

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
using namespace std;

const int M = 1031;
struct Point{
    int x, y;
};
Point p[1024];
int n;
int hash[M+8], next[1024];

bool cmp(const Point& a, const Point& b)
{
    if (a.x != b.x)
        return a.x < b.x;
    return a.y < b.y;
}
int hashcode(const Point &tp)
{
    return abs(tp.x + tp.y) % M;

}
bool hash_search(const Point &tp)
{
    int key = hashcode(tp);
    int i = hash[key];
    while (i != -1){
        if (tp.x == p[i].x && tp.y == p[i].y)
            return true;
        i = next[i];
    }
    return false;
}
void insert_hash(int i)
{
    int key = hashcode(p[i]);
    next[i] = hash[key];
```

```
    hash[key] = i;
}
int main()
{
    Point p3, p4;
    int dx, dy, ans;
    while (1 == scanf("%d", &n) && n){
        memset(hash, -1, sizeof(hash));
        memset(next, -1, sizeof(next));
        for (int i = 0; i < n; i++){
            scanf("%d %d", &p[i].x, &p[i].y);
        }
        sort(p, p+n, cmp);
        for (int i = 0; i < n ; i++) /// 排完后进行插入
            insert_hash(i);

        ans = 0;
        for (int i = 0; i < n; i++){
            for (int j = i+1; j < n; j++){
                int dx = p[j].x - p[i].x;
                int dy = p[j].y - p[i].y;
                p3.x = p[i].x + dy;
                p3.y = p[i].y - dx;
                if (hash_search(p3)){
                    p4.x = p[j].x + dy;
                    p4.y = p[j].y - dx;
                    if (hash_search(p4))
                        ans++;
                }
            }
        }
        printf("%d\n", ans/2);
    }
    return 0;
 }
```

1.1.5.3 POJ 1840 Eqs

有以下等式：$a1*x1^3+a2*x2^3+a3*x3^3+a4*x4^3+a5*x5^3=0$。x1，x2，x3，x4，x5 都就在区间[-50，50]之间的整数，且x1，x2，x3，x4，x5 都不等于 0。问：给定a1,a2,a3,a4,a5 的情况下，x1,x2,x3,x4,x5 共有多少种可能的取值？

输入有多组测试数据，每组测试数据有一行，包含 5 个整数，表示 a1，a2，a3，a4，a5。

对于每组测试数据输出一行，表示解的数量。

思路：

直接枚举x1 到x5 量会非常大，大概 100^5次，肯定会TLE。可以将式子变化一下：$a1*x1^3+a2*x2^3 = -(a3*x3^3+a4*x4^3+a5*x5^3)$，先把所有$a1*x1^3+a2*x2^3$ 结果算出来，放进哈希表里，然后再根据$-(a3*x3^3+a4*x4^3+a5*x5^3)$，枚举x3，x4，x5，然后再哈希表里查找枚举时计算的结果即可。

代码：

```
#include <stdio.h>
#include <string.h>
const int maxn = 5;
const int prime = 1280519;

struct Hash{
    int v;
    Hash* next;
};

Hash* hash[prime+1];
int c[maxn];
```

```
int x[maxn];
int ans;

int main()
{
    int t, p;
    Hash * ph;
    for (int i = 0; i < maxn; i++){
        scanf("%d", c+i);
    }
    memset(hash, 0, sizeof(hash));

    for (x[0] = -50; x[0] <= 50; x[0]++) if (x[0])
    for (x[1] = -50; x[1] <= 50; x[1]++) if (x[1])
    {
        t = -(x[0]*x[0]*x[0]*c[0] + x[1]*x[1]*x[1]*c[1]);
        p = (t>0?t:-t) % prime;
        ph = new Hash;
        ph->v = t;
        ph->next = hash[p];
        hash[p] = ph;
    }
    ans = 0;
    for (x[2] = -50; x[2] <= 50; x[2]++)if (x[2])
    for (x[3] = -50; x[3] <= 50; x[3]++)if (x[3])
    for (x[4] = -50; x[4] <= 50; x[4]++)if (x[4])
    {
        t = x[2]*x[2]*x[2]*c[2] + x[3]*x[3]*x[3]*c[3] + x[4]*x[4]*x[4]*c[4];
        p = (t>0?t:-t) % prime;
        ph = hash[p];
        while (ph){
            if (ph->v == t)
                ++ans;
            ph = ph->next;
        }
    }

    printf("%d\n", ans);
    return 0;
 }
```

其他题目：
POJ 3349 Snowflake Snow Snowflakes
POJ 2503 Babelfish
POJ 3274 Gold Balanced Lineup

1.1.6 附：字符串哈希函数

请参阅此文章：《各种字符串Hash函数比较》http://www.byvoid.com/blog/string-hash-compare/，主要是各种哈希函数的评测，下面是两个字符串哈希函数，推荐BKDR哈希函数，

```
// ELF Hash Function
unsigned int ELFHash(char *str)
{
 unsigned int hash = 0;
 unsigned int x  = 0;
 while (*str)
 {
     hash = (hash << 4) + (*str++);
     if ((x = hash & 0xF0000000L) != 0)
     {
         hash ^= (x >> 24);
         hash &= ~x;
     }
 }
```

```
    return (hash & 0x7FFFFFFF);
}

// BKDR Hash Function
unsigned int BKDRHash(char *str)
{
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;
    while (*str)
    {
        hash = hash * seed + (*str++);
    }
    return (hash & 0x7FFFFFFF);
}
```

1.2 并查集

参考文献：

《算法导论》

http://hi.baidu.com/bobo__bai/item/fbf57d110b72650fb88a1a09

编写：黄李龙　　校核：黄李龙

1.2.1 并查集基本原理

在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。这一类问题其特点是看似并不复杂，但数据量极大，若用正常的数据结构来描述的话，往往在空间上过大，计算机无法承受；即使在空间上勉强通过，运行的时间复杂度也极高，根本就不可能在比赛规定的运行时间（1～3 秒）内计算出试题需要的结果，只能采用一种全新的抽象的特殊数据结构——并查集来描述。

并查集是一种树型的数据结构，用于处理一些不相交集合的合并问题。

并查集的主要操作有：

初始化每个集合：init ();初始时每个节点（元素）都是独立的。

查找这个节点所在集合：find(v);我们用一个节点的标号表示这个节点处在哪个集合里，这个函数会返回 v 最上层的节点，也就是根。

合并两个不相交集合：join(x, y) 把 y 加到 x 集合里。

判断两个元素是否属于同一个集合：is_same(x, y)，如果 x 和 y 在同一集合内，则返回真，否则为假。

最后还会讲到一个简单高效的并查集优化。

并查样例

以 5 个节点为例，圆圈内为节点的标号，我们用数组 fa 记录每个节点的父节点，fa[i]表示 i 节点的父节点标号，当 fa[i]=i 时，节点 i 的父节点是他本身，那么 i 的所在的树的根上就是 i。

初始化，fa[i] = i，他们分别是自己的父点，现在他们是各自独立的。

fa[1] = 1;　fa[2] = 2; fa[3] = 3; fa[4] = 4; fa[5] = 5;

合并 1 和 2 时，2 的父节点变为 1，fa[2] = 1，此时有：

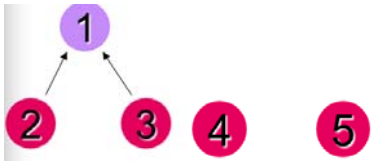
fa[1] = 1;　fa[2] = 1; fa[3] = 3; fa[4] = 4; fa[5] = 5;

如果现在我们分别查找 1 和 2 的所在树的根节点，1 节点所在树的根节点是 1，2 所在的树的根节点是 1，说明 1 和 2 是在同同一棵树内，也就是说 1 和 2 是在同一集合内的。

接下来我们合并 1 和 3，让 fa[3] = 1，即使得 3 的父节点为 1，有：

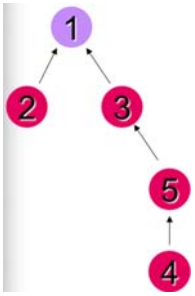
fa[1] = 1;　fa[2] = 1; fa[3] = 1; fa[4] = 4; fa[5] = 5;

很明显能知道 1,2,3 的最顶层节点都是 1。



我们再合并 4 和 5，让 4 的父节点为 5，即 fa[4] = 5。

最后，我们合并 5 和 3，让 5 的父节点为 3，即 fa[5] = 3。能得到下面的图。我们根据 fa[i]的值，能找到 4 所在树的根节点是 1，2 的所在树的根节点是 1，则可以判定 2 和 5 是属于同一棵树内的，同一个集合的。现在你应该能写出每个节点的 fa[i]值了。



那么现在开始写代码吧！

我们先定义个常量，MAX_SIZE，表示最后有多少个元素，用数字标号每个节点，在这里我们的节点标号从 0 开始。

```
cons int MAX_SIZE = 100005;
int fa[MAX_SIZE];
```

我们要有集合的初始化操作：

```
void    init {
        int i;
      for (i = 0; i < MAX_SET_SIZE; ++i) fa [i] = i;
}
```

Ok 了，接下来我们事先查找一个节点所在树的根节点的功能：

```
int find(int v)
{
```

```
if (fa[v] == v)     return v;   // 如果 fa[v] = v，那么 v 就是树的根节点了，返回 v
    return find (fa [v]); // 上面一句没成功，要找父节点的父节点
}
```

这么简单就 Ok 了！

还有一个合并集合的操作，我们把 y 并到 x 里：

```
void join(int x, int y)
{
    int fx = find(x), fy = find(y); // 先找到 x 和 y 各自的根节点
   if (fx != fy) { // 如果他们的根节点不一样，就是不在同一集合内，可以合并了
fa [fx] = fy; // 我们把 y 的的根节点的父节点设为 x 的最上层节点就行了
}
}
```

判断两个节点在同一个集合，Easy!

```
int is_same(int x, int y)
{
    return (find (x) == find(y)); // 判断它们的根节点是否一样就行了
}
```

1.2.2 并查集的时间复杂度分析和优化

好了，我们来看看他们运行所需时间。

init()操作循环 MAX_SIZE 次，时间复杂度 O(n)。

find()操作，这个有些难确定，不过它的最坏情况还是能确定的，如果对于每个节点，都有 fa[i] = i – 1，fa[0] = 0，这棵树在这种最坏情况下就退化成有 n 个元素，那么总的查找次数，也就是 find(v）调用次数为：1 + 2 + 3 + … + n，有(n+1)*n/2，时间复杂度 O(n*n)，如果节点很多，运行时间就会很慢。

join(x, y)操作，依赖于 find(v)操作，如果 find(v)快的话，join(x, y)也会很快。

一般来讲，init()只操作一次，find(v)和 join(x,y)操作 q 次，那么时间复杂度将达到 O(q*n)如果 q*n 很大的话，程序会跑得非常的慢

优化：路径压缩

主要优化 find()函数。find()函数的目的是查找到一个节点的根节点，那么找到一个节点 x 的根节点后 f，我们可以直接让 fa[x] = f，主要在再次查找 x 的父节点的时候，就能马上找到 x 节点的根节点。既然这样，我们就直接将查找路径上所有节点的父节点都设为 f，这样在查找这些节点的时候就能用两次 find()函数调用就找到了节点的根节点，大大加快了速度。

我们来实现代码：

```
int find(int v)
{
if (fa[v] == v)     return v;
    return fa[v] = find (fa [v]); // 路径压缩，直接赋值为找到的根节点
}
```

采用路径压缩后，每一次查询所用的时间复杂度为增长极为缓慢的 ackerman 函数的反函数——α（x）。这里 α 函数是 Ackerman 函数的某个反函数，在很大的范围内（人类目前观测到的宇宙范围估算有 10 的 80 次方个原子，这小于前面所说的范围）这个函数的值

可以看成是不大于 4 的，所以并查集的操作可以看作是线性的。

路径压缩可以写成非递归形式，你可以自己想想怎么写，这里就不作为重点写出来了。

还有一个优化，就是根据树的深度来合并集合，具体怎么实现，请你自己上网查一下，相信你能完成它。

1.2.3 并查集样例代码

数据结构：

```
const int MAX_SIZE = 100005;
int father[MAX_SET_SIZE];
```

初始化

```
void  init {
    int i;
    for (i = 0; i < MAX_SET_SIZE; ++i) fa [i] = i;
}
```

查找

```
int find(int v)
{
if (fa[v] == v)   return v;
    return fa[v] = find (fa [v]);
}
```

判断是否在同一集合内

```
int is_same(int x, int y)
{
    return (find (x) == find(y));
}
```

并查集合并

```
void join(int x, int y)
{
    int fx = find(x), fy = find(y);
    if (fx != fy)
        fa[fx] = fy;
}
```

1.2.4 例题讲解

下面讲解三道样例题目，分别是：

Hrbustoj 1073 病毒

POJ 2492 A Bug's Life

POJ 1182 食物链

1.2.4.1 Hrbustoj 1073 病毒

某种病毒袭击了某地区，该地区有 N(1≤N≤50000)人，分别编号为 0,1,...,N-1，现在 0 号已被确诊，所有 0 的直接朋友和间接朋友都要被隔离。例如：0 与 1 是直接朋友，1 与 2 是直接朋友，则 0、2 就是间接朋友，那么 0、1、2 都须被隔离。现在，已查明有 M(1≤M≤10000)个直接朋友关系。如：0,2 就表示 0,2 是直接朋友关系。

请你编程计算，有多少人要被隔离。

输入数据的第一行包含两个正整数 N(1≤N≤50000),M(1≤M≤100000)，分别表示人数和接触关系数量；

在接下来的 M 行中，每行表示一次接触，；

每行包括两个整数 U, V(0 <= U, V < N)表示一个直接朋友关系。
注意有多组测试数据。

输出数据仅包含一个整数，为共需隔离的人数（包含 0 号在内）。
思路：
与 0 有过直接或者间接接触的都是需要隔离的人，简单的并查集应用。
代码：

```
#include <stdio.h>
const int MAXN = 50005;
int f[MAXN];

int find(int x)
{
   if (x != f[x]) f[x] = find(f[x]);
   return f[x];
}
void union_set(int x, int y)
{
    int px = find(x);
    int py = find(y);
    if (px != py) f[px] = py;
}

int main()
{
   int cnt, f0, n, m, x, y;

   while (EOF != scanf("%d %d", &n, &m)) {
        for (int i = 0; i < MAXN; i++) f[i] = i;
        for (int i = 0; i < m; i++) {
           scanf("%d %d", &x, &y);
           union_set(x, y);
        }
        cnt = 1;
        f0 = find(0);
        for (int i = 1; i < n; i++) {
           if (f0 == find(i)) cnt++;
        }
        printf("%d\n", cnt);
   }
   return 0;
}
```

1.2.4.2 POJ 2492 A Bug's Life

题目大意：一个无聊的科学家说只有两个不同性别的昆虫能在一起，当然是在没有同性恋的情况下。给你几对能在一起的昆虫，问里面有没有同性恋，也就是交配是否有冲突。

输入一个数 t，表示测试组数。然后每组第一行两个数字 n,m，n 表示有 n 只昆虫，编号从 1—n,m 表示下面要输入 m 行交配情况，每行两个整数，表示这两个编号的昆虫为异性，要进行交配。 要求统计交配过程中是否出现冲突，即是否有两个同性的昆虫发生交配。

Sample Input

2
3 3
1 2

2 3
1 3
4 2
1 2
3 4
Sample Output

Scenario #1:
Suspicious bugs found!

Scenario #2:
No suspicious bugs found!

思路：

昆虫只有公和母两种情况，可以考虑昆虫之间的“偏移”关系，如果偏移次数是偶数次，说明是同一个性别的，奇数次则说明是异性。怎样实现呢？

用一个 rel[i]数组表示 i 昆虫相对根节点的偏移次数，初始时 rel[i]=0。考虑第一对昆虫 a 和 b 的关系，他们的性别是不一样的，如果把 a 作为根，那么 rel[b] = 1，这样 b 到 a 的偏移就是 rel[b] + rel[a]，结果是奇数。在每次加入一个可以交配的昆虫 a 和 b 时，先判断他们是否在同一集合内，不在则表示两只昆虫是可交配的，在同一个集合则要判断两只昆虫的偏移关系 rel[a]和 rel[b]，如果 rel[a]+rel[b]是偶数则两只昆虫是同性，说明找到了一对将要交配的同性昆虫，否则什么也不做。

此题关键是对昆虫偏移关系的维护，注意看代码中的写法。

```
#include <stdio.h>

const int MAXN = 2005;
int f[MAXN], rel[MAXN];

void init(int n)
{
    for (int i = 0; i <= n; i++) {
        f[i] = i;
        rel[i] = 0;
    }
}

int find(int x)
{
    if (x != f[x]) {
        int t = f[x];
        f[x] = find(f[x]);
        rel[x] = (rel[x] + rel[t]) % 2;
    }
    return f[x];
}

void union_set(int a, int b) // a, b are different gender
{
    int fa = find(a), fb = find(b);
    if (fa != fb) {
        f[fb] = fa;
        rel[fb] = (rel[a] - rel[b] + 1 + 2) % 2;
    }
}

int main()
{
    int caseN;
    scanf("%d", &caseN);
    for (int cs = 1; cs <= caseN; cs++) {
```

```
            int n, m;
            bool found = false;
            scanf("%d %d", &n, &m);
            init(n);
            for (int i = 0, a, b; i < m; i++) {
                scanf("%d %d", &a, &b);
                int ta = find(a), tb = find(b);
                if (ta != tb) {
                    union_set(a, b);
                } else if (rel[a] == rel[b]) {
                    found = true;
                }
            }
            printf("Scenario #%d:\n", cs);
            printf("%s\n", found ? "Suspicious bugs found!" : "No suspicious bugs
found!");
            if (cs < caseN) {
                printf("\n");
            }
        }
        return 0;
    }
```

1.2.4.3 POJ 1182 食物链

题意：动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B， B 吃 C，C 吃 A。

现有 N 个动物，以 1－N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

第一种说法是"1 X Y"，表示 X 和 Y 是同类。

第二种说法是"2 X Y"，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

1）当前的话与前面的某些真的话冲突，就是假话；

2）当前的话中 X 或 Y 比 N 大，就是假话；

3）当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N（1 <= N <= 50,000）和 K 句话（0 <= K <= 100,000），输出假话的总数。

Input

第一行是两个整数 N 和 K，以一个空格分隔。

以下 K 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。

若 D=1，则表示 X 和 Y 是同类。

若 D=2，则表示 X 吃 Y。

Output

只有一个整数，表示假话的数目。

Sample Input

100 7
1 101 1
2 1 2
2 2 3

2 3 3
1 1 3
2 3 1
1 5 5
Sample Output

3

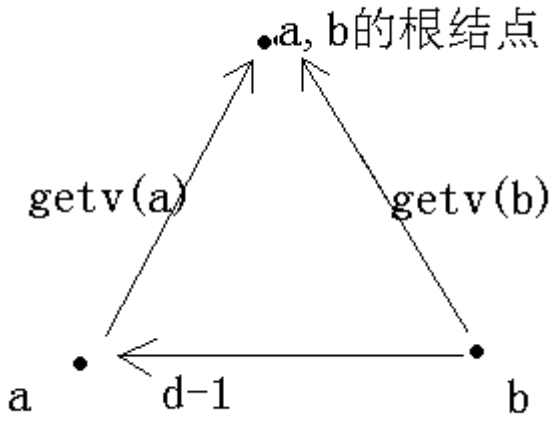
思路：这一题的基本思路是对每句话判断真假，同时记录由真话构成的食物链的结构，进行统计，最后输出统计结果。

关键是对每句话的判断上。

可以看出 3 个物种之间的关系是相对的，对称的，所以要记录的是动物间的关系，而非它们是什么物种。如果我们把相互间可以确定捕食关系的动物放在一个集合中，那么实际上在处理过程中我们可能会得到多个集合，同时会出现需要将两个集合合并的情况(例如 a 属于集合 A，b 属于集合 B，当 a，b 的关系确定时，集合 A，B 就需要合并为一个集合)，由这种动态处理集合的情况不难联想到并查集。

利用并查集来存储已知的捕食结构。当 f[a]=a 时，表示 a 是本集合的根结点，f[a]=b 表示 a，b 间有已经确定的捕食关系。这样想知道 a，b 间的捕食关系只要找到 a，b 的根结点即可，a，b 有相同根结点则可通过逻辑判断确定 a，b 关系，a，b 有不同根节点，说明 a，b 关系尚未确定。这样我们还需要一个和 f[n]对应的存储结构 vec[n]用以存储 f[a]与其父结点的捕食关系。由于关系都是相对的，而且有向的，我们可以以向量的思想来确定。

我们以 vec[a]=0 表示 a 结点与其父结点是同类，vec[a]=-1 表示 a 捕食其父结点，vec[a]=1 表示 a 被其父结点捕食。首先构造函数 getf(a)来得到 a 的根结点，构造 getv(a)来得到 a 与其根结点的关系向量。



当得到输入 d a b 时，若 ab 有相同根结点，则如上图，不难有 d-1==getv(b)-getv(a)时是真话

同样，a，b 有不同根结点时，这句话为真，可以确定两根结点的关系，a 到 b 的关系向量为:

getv(b)-(d-1)-getv(a);

以上就是以向量和并查集来考虑本题的思路。

参考：http://hi.baidu.com/bobo__bai/item/fbf57d110b72650fb88a1a09

参考代码：代码中用 pa 数组表示每个节点的父节点，ch 数组表示节点与它的父节点的关系，即向量的关系。

```
#include <stdio.h>
#include <string.h>
const int MAXN = 50005;
int pa[MAXN], ch[MAXN];
int n, k, lies;
void init_set()
{
   int i;
   for (i = 0; i < MAXN; i++) pa[i] = i;
   memset(ch, 0, sizeof(int)*MAXN);
   lies = 0;
}

int find_set(int x)
{
   if (pa[x] == x)
      return x;
   int xt = find_set(pa[x]);
   ch[x] = (ch[x] + ch[pa[x]] + 3) % 3;
   pa[x] = xt;
   return xt;
}
void union_set(int d, int x, int y)  // d, 表示 x 对 y 的关系，0， 同类，1，x 吃 y
{
   if (x > n || y > n )  {
      ++lies;
      return;
   }
   int fx = find_set(x);
   int fy = find_set(y);
   if (fx == fy)  {
      if ((ch[x] - ch[y] + 3) % 3 != d) {
         ++lies;
      }
   } else {
   ch[fx] = (d + ch[y] - ch[x] + 6) % 3;
pa[fx] = fy;
}
}
int main()
{
   int i, d, x, y;
   scanf("%d %d\n", &n, &k);
   init_set();
   for (i = 0; i < k; i++) {
      scanf("%d %d %d\n", &d, &x, &y);
      union_set(d - 1, x, y);
   }
   printf("%d", lies);
   return 0;
}
```

其它并查集题目：
HDU 1213_How Many Tables
POJ 1703_Find them, Catch them
Hrbustoj 1418 夏夜星空
POJ 1988 Cube Stacking

1.3 二叉堆

参考文献：
《算法导论》第 6 章 堆排序

二叉堆：http://www.nocow.cn/index.php/%E4%BA%8C%E5%8F%89%E5%A0%86

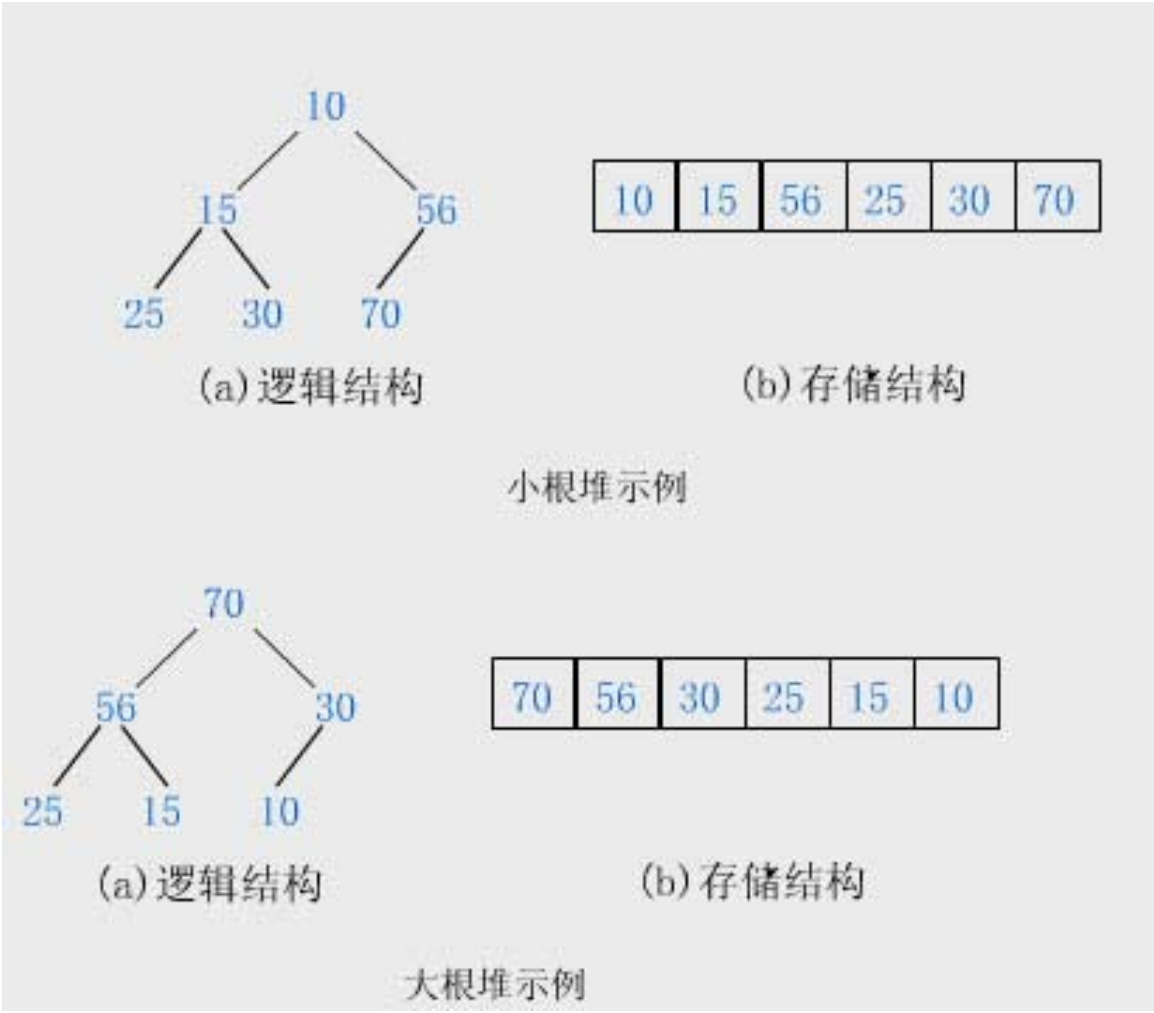
编写：黄李龙　　校核：黄李龙

1.3.1 二叉堆的概念

二叉堆(Binary Heap)是一种特殊的堆，二叉堆是完全二叉树或者是近似完全二叉树。二叉堆满足堆特性：父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值，且每个结点的左子树和右子树都是一个二叉堆（都是最大堆或最小堆）。

当父结点的键值总是大于或等于任何一个子节点的键值时为最大堆。 当父结点的键值总是小于或等于任何一个子节点的键值时为最小堆。

(注:一般把二叉堆简称为堆)



预备知识：我们将一棵二叉树从上到下，从左到右编号，我们可以发现，第 i 个节点的两个儿子的编号分别为 2*i，2*i+1。

堆其实是一颗完全二叉树，当且仅当它满足以下两个条件之一时，才能称之为堆：

(1)a[i]<=a[i*2]且 a[i]<=a[i*2+1](即小根堆，节点的值不大于两个儿子的值)

(2)a[i]>=a[i*2]且 a[i]>=a[i*2+1](即大根堆，节点的值不小于两个儿子的值)

注意：

①堆中任一子树亦是堆。

②以上讨论的堆实际上是二叉堆(Binary Heap)，类似地可定义 k 叉堆。

1.3.2 二叉堆的基本操作

数据结构表示：

使用一个一维数组表示，下标从 1 开始计算，如果是 C\C++语言，则忽略下标为 0 的元素。使用 n 表示队中元素的个数。

堆的两个基本操作

堆关键是理解两个基本操作的实现：

1、向堆插入一个结点(上升操作)：

在堆尾（a[n]）插入一个元素，然后不断和父结点（a[i/2]）比较，如果比父结点大（大根堆]）或小（小根堆）就交换，一直到堆顶或不再交换就结束。

C 语言代码：

```
void push_heap(int a[], int n) {
    int i = n;
    int x = a[n];
    while (i > 1 && a[i/2] < x) {
        a[i] = a[i/2];
        i /= 2;
    }
    a[i] = x;
}
```

(以上代码是大根堆的上升操作，小根堆只需将 a[i]>a[j]改为 a[i]<a[j]即可)

具体使用时，先在一维数组后面加入元素，然后调用 push_heap 函数进行调整。如下面的代码：

```
++n;
a[n] = x;
push_heap(a, n);
```

2、删除结点：

删除堆顶结点(下降操作)：

将堆顶结点（堆数组第一个）和堆尾结点（堆数组最后一个）交换，然后删除堆尾结点，将交换后的节点和左右儿子比较大小，然后选择两个儿子较大者（大根堆）或较小者（小根堆）再与节点进行比较大小，交换，更换节点编号，如此重复，直到满足堆的性质。

C 语言代码：

先给一个函数，函数功能是把根为 m 的非大根堆调整成大根堆，前提条件是这个非大根堆的子堆，也就是根的儿子必须是大根堆。

```
void heap(int a[], int n, int m) { // n 是数组下标范围，m 是 需要调整的根的下标
 int t;
 while(m*2 <= n) {
     m = m * 2;
     if (m < n && a[m] < a[m+1]) {
         m++;
     }
     if (a[m] > a[m/2]) {
         t = a[m];
         a[m] = a[m/2];
         a[m/2] = t;
     }
     else
         break;
 }
}
```

下面的代码是把一个堆的堆顶去掉，并调整堆。

```
void pop_heap(int a[], int n) {
 int x;
 x = a[1];
 a[1] = a[n];
 a[n] = x;
 n--;
 heap(a, n, 1);
}
```

(以上代码是大根堆的下降操作，小根堆只需将所有的‘>'换为'<'即可)

不难得出，两个操作的复杂度都是$O(\log_2 N)$。

删除堆内结点：

该操作作为扩展内容进行介绍，具体代码请读者自己实现，代码不难，可以利用删除堆顶元素时使用的 heap 函数进行调整。

如果要删除的结点是堆内结点，则需先判断是需要下降还是需要上升。

算法是：使用末尾替换需要删除的节点，首先尝试下降操作，如果无法下降，则尝试上升操作。

1.3.3 堆排序

堆排序利用了大根堆(或小根堆)堆顶记录的关键字最大(或最小)这一特征，使得在当前无序区中选取最大(或最小)关键字的记录变得简单。

①先将初始数组 a[1..n]建成一个大根堆,此堆为初始的无序区

②再将关键字最大的记录 a[1](即堆顶)和无序区的最后一个记录 a[n]交换，由此得到新的无序区 a[1..n-1]和有序区 a[n]，且满足 a[1..n-1] ≤ a[n]

③由于交换后新的根 a[1]可能违反堆性质，故应将当前无序区 a[1..n-1]调整为堆。

然后再次将 a[1..n-1]中关键字最大的记录 a[1]和该区间的最后一个记录 a[n-1]交换，由此得到新的无序区 a[1..n-2]和有序区 a[n-1..n]，且仍满足关系 a[1..n-2]≤R[n-1..n]，同样要将 a[1..n-2]调整为堆。

重复②③的步骤，直到无序区只有一个元素为止，此时 a[1..n]就是一个有序的数组了。

C 语言代码：

```
//首先建堆，从最底层的元素开始建堆，一直递推到堆顶，即下标为 1 的数据元素。
void make_heap(int a[], int n) {
 int i;
 for (i = n/2; i > 0; --i) heap(a, n, i);
}
// 堆排序
void headp_sort(int a[], int n) {
   int i;
   make_heap(a, n); // 先把无序的元素建立成堆
 for (i = n; i > 1; --i) {
     int t = a[1];
     a[1] = a[i];
     a[i] = t;
     heap(a, n - i, 1);
 }
}
```

可以得出堆排的时间复杂度约为 O(n*logn)。

1.3.4 经典题目

1.3.4.1 poj 3253 Fence Repair

题意:给几根木板，要你把他们连接起来，每一次连接的花费是他们的长度之和。问最少需要多少钱？

输入数据的第一行是模板个数 N(1 ≤ N ≤ 20000)，接下来有 N 行，第 i+1 行一个整数 Li （1 ≤ Li ≤ 50000），表示第 i 根模木板。

输出最少的花费。

思路：跟哈夫曼编码的过程相似，是贪心题，只是哈夫曼编码跟这个相似。

每次只取两根长度最小的木板，然后连接成一根木板，放入剩余的木板集合中，然后重复上述过程，知道剩余木板集合中只剩下一根木板，每次连接木板的花费的和就是所求答案。

代码如下：

```
#include <stdio.h>
#include <string.h>
const int maxn = 20000+2;

int l[maxn];
int n, h, min;
long long ans;

void heap_sort(int x)
{
   int lg, lr, t;
   while ((x<<1) <= h){
      lg = x << 1;
      lr = (x<<1)+1;
      if (lr <= h && l[lg] > l[lr])
         lg = lr;
      if (l[x] > l[lg]){
         t = l[x];
         l[x] = l[lg];
         l[lg] = t;
         x = lg;
      }
      else
         break;
   }
}
int main()
{
  int i;
 while (1==  scanf("%d", &n)){
     for (i = 1; i <= n; i++)
         scanf("%d", &l[i]);

     h = n;
     for (i = n/2; i > 0; i--)
         heap_sort(i);

     ans = 0;

     while (h > 1){
         min = l[1];
         l[1] = l[h];
         h--;
         heap_sort(1);

         min += l[1];
         l[1] = min;
         heap_sort(1);

         ans += min;
     }
     printf("%I64d\n", ans); // POJ 是 Windows 平台，输出 64 位整数需要 I64d
 }
   return 0;
```

```
}
```

1.3.4.2 POJ 2442 Sequence

题意：有m个序列，每个序列 n个非负数，从每个序列中选择一个数组成一个序列并计算和，总共有 n^m 个和，输出最小的n个和。

思路：直接计算复杂度太高。故考虑类似动态规划的思想。先求出前 k-1 个序列的最小 n 个元素，存到 A 数组中，A 数组中的数都是 k-1 个数的和。然后利用第个 k 序列的数生成 B 数组，B 数组是 k 个数和，B 数组也有 n 个。转移时，用第 k 个序列中的每一项 li (1<= i <= n) 与 A 数组中的每个数求和，若和比 B 数组中的最大值小，则需要更新 B 数组中的最小的 n 个元素。

程序实现时，使用滚动数组轮流使用 A 和 B 数组。为了简便，直接使用 C++ STL 库里的二叉堆操作函数。

C++代码：

```
#include <cstdio>
#include <algorithm>
using namespace std;

const int MAXN = 2005, MAXM = 105;
int a[2][MAXN];

int main() {
 int runs;
 scanf("%d", &runs);
 while (runs--) {
     int n, m;
     scanf("%d%d", &m, &n);
     for (int i = 0; i < n; ++i) scanf("%d", a[0] + i);
     make_heap(a[0], a[0] + n);
     int q = 0;
     for (int h = 1; h < m; ++h, q = 1-q) {
         int t;
         scanf("%d", &t);
         for (int i = 0; i < n; ++i) a[1-q][i] = a[q][i] + t;
         for (int i = 1; i < n; ++i) {
             scanf("%d", &t);
             for (int j = 0; j < n; ++j) {
                 int t2 = a[q][j] + t;
                 if (t2 < a[1-q][0]) {
                     pop_heap(a[1-q], a[1-q] + n);
                     a[1-q][n-1] = t2;
                     push_heap(a[1-q], a[1-q] + n);
                 }
             }
         }
     }
     sort(a[q], a[q] + n);
     for (int i = 0; i < n-1; ++i) {
         printf("%d ", a[q][i]);
     }
     printf("%d\n", a[q][n-1]);
 }
 return 0;
}
```

1.3.4.3 POJ 2010 Moo University - Financial Aid

题意：给定 C 头牛的 CSAT 分数 score[i]，和需要的资费 aid[i]，求上述 C 头牛的一个 N 元子集，使得其中位数最大，而资费总和<=f（特定的值）

思路：先将牛以 score 从小到大排序。

于是可以枚举[N/2..C-N/2]之间的每头牛为中位数点，那么要满足题目条件则有：

选择第 i 个牛为中位数点，那么设 i 点之前的(n/2)个 financial aid 为 before[i]，before[i]

不含 i 号牛，设 i 点之后的(n/2)个 financial aid 为 after[i]，不含 i 号牛，则有 fa[i](i 点的 financial aid 值）+ before[i] + afeter[i] <= F 的时候，满足条件。

由于牛的序列是以 score 递增的，从大到小扫一遍，遇到的第一个结果就是答案，时间复杂度 O(n)。

那么如何确定 before[]和 after[]呢？

分别维护一个大根堆，初始的时候最大堆的元素的数目为(n/2) 个，那么每次更新一头牛 i 的时候，若这个点的 fa[i] > 大根堆堆顶点的 fa，那么更新 before[]（afeter[]）值即可。具体实现看代码。

于是只需要log(n)的时间来维护这个before[i]/after[i]值，故总的复杂度为$O(N\log_2 N)$。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 100000 + 100;
struct Node {
 int s, f;
 bool operator < (const Node &B) const {
     return s < B.s;
 }
} node[MAXN];

int before[MAXN];
int heap[MAXN];
int heap_size, heap_sum;

void adjust(int c)
{
 int *p = heap - 1;
 int val = p[c];
 while (c * 2 <= heap_size) {
     int lc = c * 2;
     if (lc < heap_size && p[lc+1] > p[lc]) {
         ++lc;
     }
     if (p[lc] > val) {
         p[c] = p[lc];
         c = lc;
     } else {
         break;
     }
 }
 p[c] = val;
}

void init_heap(int l, int r)
{
 heap_sum = 0;
 for (int i = l; i <= r; ++i) {
     heap[i - l] = node[i].f;
     heap_sum += node[i].f;
 }
 for (int i = heap_size / 2; i > 0; --i) {
     adjust(i);
 }
}

void update_heap(int f)
{
 int *p = heap - 1;
 if (f < p[1]) {
     heap_sum -= p[1] - f;
```

```
        p[1] = f;
        adjust(1);
    }
}

int main()
{
    int n, c, f;
    while (EOF != scanf("%d %d %d", &n, &c, &f)) {
        for (int i = 0; i < c; ++i) {
            scanf("%d %d", &node[i].s, &node[i].f);
        }
        sort(node, node + c);
        heap_size = n / 2;
        init_heap(0, heap_size - 1);
        for (int i = heap_size; i < c - heap_size; ++i) {
            before[i] = heap_sum;
            update_heap(node[i].f);
        }
        int ans = -1;
        init_heap(c - heap_size, c - 1);
        for (int i = c - heap_size - 1; i >= heap_size; --i) {
            int t = before[i] + heap_sum + node[i].f;
            if (0 <= t && t <= f) {
                ans = node[i].s;
                break;
            }
            update_heap(node[i].f);
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

1.3.4.4 POJ 3481 Double Queue

题意：在一个银行中，每天都会有客户来办理业务。每个客户用一个正整数 K 标识，因为银行人手不多，客户在办理时需要等待其他用户办理。在客户办理业务前，银行为用户提供一个优先级 P，根据优先级来确定先处理哪个用户的业务，并且有时候是优先级高的先服务，有时是优先级的先服务。银行系统将对客户的服务分为以下几种请求：

0	停止系统的服务
1 K P	加入 K 用户的业务请求，优先级为 P
2	先服务一个最高优先级的客户
3	先服务一个最低优先级的客户

输入：

每一行为一个请求，并且最后一行的请求为 0，表示停止服务。输入保证有类型为 1 的请求，每个客户的标识都不一样，并且没有两个请求的优先级是一样的。客户标识小于 1000000，优先级 P 小于 10000000，一个客户可以有多次请求，每次请求的优先级都不同。

输出：

对于每个类型为 2 或者 3 的请求，输出服务的客户标识。如果没有客户在请求，则输出 0。

思路：

如果仅考虑一个取得最高的优先级，则该题就是简单的堆的应用。除了去最高优先级外，还有娶最低优先级的操作，因为每个请求的优先级都不一样，我们可以用优先级标记每一个请求是否已经处理过，即已经从队列中删除，在每次取最高优先级或者最低优先级

时，要判断这个优先级的请求是否已经处理过了，处理过了的话则继续取下一个优先级的客户。取到后标记对应优先级的请求为删除状态即可。

```
/*
 * POJ 3481 Double Queue
 * 两个堆，用关联映射之间的值
 * */
#include <cstdio>
#include <cstring>
#include <utility>
#include <algorithm>
using namespace std;

typedef pair<int, int> PII;

const int MAXN = 1000006;
struct KP {
 int val, id;
 int fix;
};
struct greaterKP {
 bool operator () (const KP &a, const KP &b) const {
     return a.val > b.val;
 }
};

struct lessKP {
 bool operator () (const KP &a, const KP &b) const {
     return a.val < b.val;
 }
};

int fix_cnt;
bool del[MAXN];
int sz_max, sz_min;
KP kp_max[MAXN], kp_min[MAXN];

template <class cmp>
int get(KP a[], int &sz) {
 while (sz > 0 && del[a[0].fix]) {
     pop_heap(a, a + sz, cmp());
     --sz;
 }
 if (sz == 0) return 0;
 else {
     del[a[0].fix] = 1;
     return a[0].id;
 }
}
template <class cmp>
void ins(KP a[], int &sz, const KP &tmp) {
 a[sz++] = tmp;
 push_heap(a, a+sz, cmp());
}

void init() {
 sz_max = sz_min = 0;
 fix_cnt = 0;
}

int main() {
 int cmd;
 while (EOF != scanf("%d", &cmd)) {
     if (cmd == 0) {
         continue;
     }
     if (cmd == 1) {
```

```
            int k, p;
            scanf("%d%d", &k, &p);
            KP t;
            t.val = p;
            t.id = k;
            t.fix = fix_cnt;
            del[fix_cnt++] = 0;
            ins<lessKP>(kp_max, sz_max, t);
            ins<greaterKP>(kp_min, sz_min, t);
        } else if (cmd == 2) {
            int a = get<lessKP>(kp_max, sz_max);
            printf("%d\n", a);
        } else if (cmd == 3) {
            int a = get<greaterKP>(kp_min, sz_min);
            printf("%d\n", a);
        } else {
        }
    }
    return 0;
}
```

其他题目：
POJ 3481 Double Queue
POJ 1442 Black Box

1.4 树状数组

参考文献：
树状数组：http://www.cnblogs.com/Creator/archive/2011/09/10/2173217.html
扩展阅读：

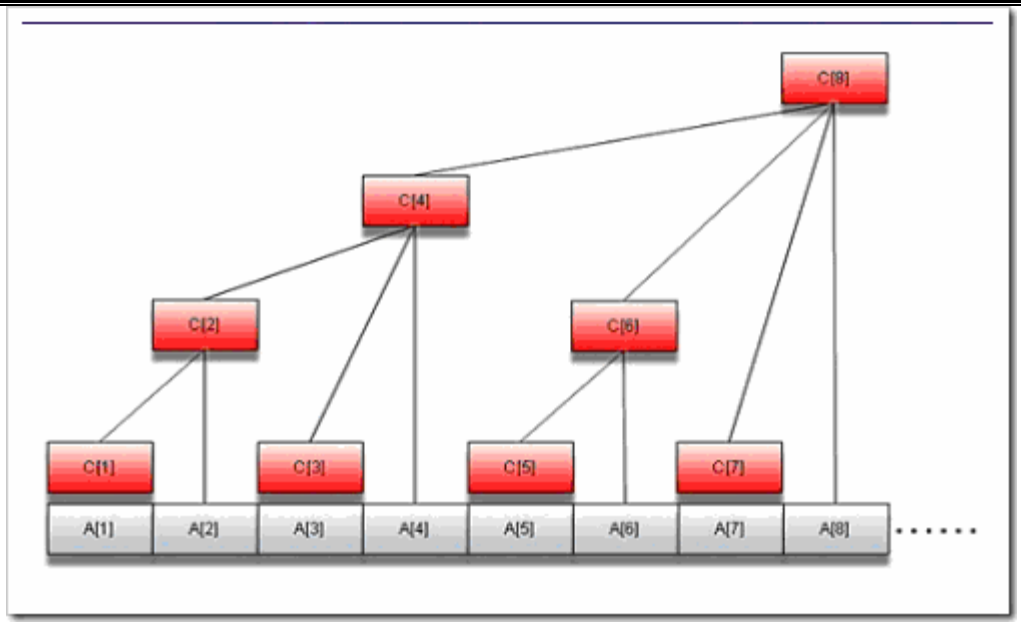
编写：黄李龙　　校核：黄李龙

1.4.1 基本原理

在一个数组中。若你需要频繁的计算一段区间内的和，你会怎么做？，最最简单的方法就是每次进行计算，但是这需要 O(N)的时间复杂度，如这个需求非常的频繁，那么这个操作就会占用大量的 CPU 时间，进一步想一想，你有可能会想到使用空间换取时间的方法，把每一段区间的值一次记录下来，然后存储在内存中，将时间复杂度降低到 O（1），的确，对于目前的这个需求来说，已经能够满足时间复杂度上的要求，尽管带来了线性空间复杂度的提升。

但若是我们的源数据需要频繁的更改怎么办？使用上面的方案，我们需要大量的更新我们保存到内存中的区间和，而且这中间的很多更新的影响是重叠的，我们需要重复计算。例如对于数组 array[10],更新了 array[4]值，需要更新区间[4,5],[4,5,6],在更新[4,5,6]需要又一次的计算[4,5]，这样的更新带来了非常多的重复计算，为了解决这一问题，树状数组应运而生了。

当要频繁的对数组元素进行修改,同时又要频繁的查询数组内任一区间元素之和的时候,可以考虑使用树状数组.树状数组是一种非常优雅的数据结构.先来看看一张树状结构的图片：



图中 C[1]的值等于 A[1]，C[2]的值等于 C[1]+A[2]=A[1]+a[2],C[4]的值=C[2]+C[3]=A[1]+A[2]+A[3]+A[4],假设我们现在需要更改元素 a[2],那么它将只影响到得 c 数组中的元素有 c[2],c[4],c[8],我们只需要重新计算这几个值即可，减少了很多重复的操作。这就是树状结构大致的一个存贮示意图。

下面看看它的定义：假设 a[1...N]为原数组,定义 c[1...N]为对应的树状数组：

c[i] = a[i - 2^k + 1] + a[i - 2^k + 2] + ... + a[i]（其中 k 为 i 的二进制表示末尾 0 的个数）。

下面枚举出 i 由 1...5 的数据，可见正是因为上面的 a[i - 2^k + 1]...a[i]的计算公式保证了我们 C 数组的正确意义，至于证明过程，请读者自己查找资料。

i		K		
1	$(1)_2$	0	1-2^0+1=1…1	c[1]=a[1]
2	$(10)_2$	1	2-2^1+1=1…2	c[2]=a[1]+a[2]=c[1]+a[2]
3	$(11)_2$	0	3-2^0+1=3…3	c[3]=a[3]
4	$(100)_2$	2	4-2^2+1=1…4	c[4]=a[1]+a[2]+a[3]+a[4]=c[2]+c[3]+a[4]
5	$(101)_2$	0	5-2^0+1=5…5	c[5]=a[5]

1.4.1.1 基本操作

对于 C[i]=a[i - 2^k + 1]...a[i]的定义中，比较难以逐磨的 k，他的值等于 i 这个数的二进制表示末尾 0 的个数.如 4 的二进制表示 0100,此时 k 就等于 2，而实际上我们还会发现 2^k 就是前一位的权值,即 0100 中,2^2=4，刚好是前一位数 1 的权值.所以所以 2^k 可以表示为 n&(n^(n-1))或更简单的 n&(-n)，例如：

（为了表示简便，假设现在一个 int 型为 4 位,最高位为符号位。）

int i=3&(-3);　　此时 i=1，3 的二进制为 0011,-3 的二进制为 1101(负数存的是补码）所以 0011&1101=1

int j=4&(-4);　　此时 j=4，理由同上。

所以计算 2^k 我们可以用如下代码：

```
int lowbit(int x)//计算 lowbit
```

```
{
    return x&(-x); // 也可以写成 return x & (x ^ (x – 1));
}
```

这个操作的时间复杂度是 O(1)。

1.4.1.2 求和操作

在上面的示意图中，若我们需要求 sum[1..7]个元素的和，仅需要计算 c[7]+c[6]+c[4]的和即可，究竟时间复杂度怎么算呢？一共要进行多少次求和操作呢？

求 sum[1..k],我们需查找 k 的二进制表示中 1 的个数次就能得到最终结果，具体为什么，请见代码 i-=lowbit(i)注释

```
int sum(int i)//求前 i 项和
{
    int s=0;
    while(i>0)
    {
        s += c[i];
        i -= lowbit(i);
    }
    return s;
}
```

时间复杂度是 O(logN)。

代码中“i -= lowbit(i)”的解释，这一步实际上等价于将 i 的二进制表示的最后一个 1 剪去，再向前数当前 1 的权个数（例子在下面），而 n 的二进制里最多有 log(n)个 1，所以查询效率是 log(n)，在示意图上的操作即可理解为依次找到所有的子节点。

以求 sum[1..7]为例,二进制为 0111,右边第一个 1 出现在第 0 位上,也就是说要从 a[7]开始向前数 1 个元素(只有 a[7]),即 c[7];

然后将这个 1 舍掉,得到 6,二进制表示为 0110,右边第一个 1 出现在第 1 位上,也就是说要从 a[6]开始向前数 2 个元素(a[6],a[5]),即 c[6];

然后舍掉用过的 1,得到 4,二进制表示为 0100,右边第一个 1 出现在第 2 位上,也就是说要从 a[4]开始向前数 4 个元素(a[4],a[3],a[2],a[1]),即 c[4].

所以 s[7]=c[7]+c[6]+c[4]。

1.4.1.3 给源数组加值操作

在上面的示意图中，假设更改的元素是 a[2],那么它影响到得 c 数组中的元素有 c[2],c[4],c[8]，我们只需一层一层往上修改就可以了,这个过程的最坏的复杂度也不过 O(logN);

```
void add(int i,int val)
{
    while(i<=n)
    {
        c[i] += val;
        i += lowbit(i);

    }
}
```

时间复杂度是 O(logN)。

代码中“i += lowbit(i)”的解释， i+（i 的二进制中最后一个 1 的权值,即 2^k）,在示意图上的操作即为提升一层，到上一层的节点，这个过程实际上也只是一个把末尾 1 后补 0 的过程（例子在下面）。

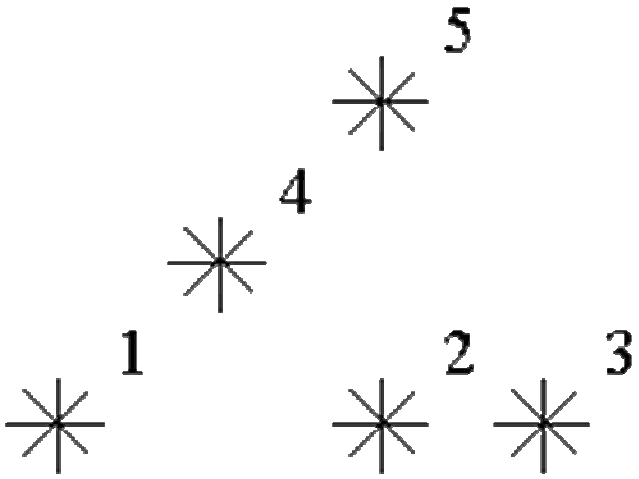
以修改 a[2]元素为例，需要修改 c[2],2 的二进制为 0010,末尾补 0 为 0100,即 c[4]，4 的二进制为 0100，在末尾补 0 为 1000 即 c[8]。所以我们需要修改的有 c[2],c[4],c[8]。

关于树状数组还有二维树状数组和三维数组数组等，建议把二维树状数组掌握。

1.4.2 树状数组例题

1.4.2.1 POJ 2352 Stars

天空中有一些星星，这些星星都在不同的位置，每个星星有个坐标。
如果一个星星的左下方（包含正左和正下)有 k 颗星星，就说这颗星星是 k 级的.比如，在下面的例图中，星星 5 是 3 级的（1，2，4 在它左下）。
星星 2，4 是 1 级的。例图中有 1 个 0 级，2 个 1 级，1 个 2 级，1 个 3 级的星。



求出各个级别的星星的个数。

思路：
算法有很多种,最实用的是树状数组。
每个星星的级别定义就是在它的左下角有多少颗星星，我们可以先对 Y 排序,然后就是查找每个坐标前面的坐标中 X 比他小的又多少个，这样就能算出星星的级别了，这需要树状数组。从这里我们可以看出，树状数组存储的是某一段范围内有多少个点。
原始题目中已经说明星星已经按照 Y 坐标排好序输入了，所以可以不用排序了。
C++代码：（下面的代码只是利用了树状数组的知识，并没有把相应的 add 和 sum 函数等写进去，读者可以自己修改）

```
#include <stdio.h>
#include <string.h>

const int MAXN = 15000+5;
const int MAXXY = 32000;

int level[MAXN];
int f[MAXXY*3+3];
```

```
int n;

int main()
{
   while (EOF != scanf("%d", &n)) {
      memset(level, 0, sizeof (level));
      memset(f, 0, sizeof(f));
      for (int i = 0; i < n; i++) {
         int x, y, s = 0, k = 1, l = 0, r = MAXXY;
         scanf("%d %d", &x, &y);
         while (l < r) {
            int mid = (r - l) / 2 + l;
            if (mid < x) { // right
               l = mid + 1;
               s += f[k];
               k = (k << 1) + 1;
            } else { // left
               r = mid;
               f[k]++;
               k <<= 1;
            }
         }
         s += f[k];
         f[k]++;
         level[s]++;
      }
      for (int i = 0; i < n; i++) {
         printf("%d\n",level[i]);
      }
   }
   return 0;
}
```

1.4.2.2 hrbustoj 1400 汽车比赛

题意：

XianGe 非常喜欢赛车比赛尤其是像达喀尔拉力赛，这种的比赛规模很大，涉及到很多国家的车队的许多车手参赛。XianGe 也梦想着自己能举办一个这样大规模的比赛，XianGe 幻想着有许多人参赛，那是人山人海啊，不过 XianGe 只允许最多 100000 人参加比赛。

这么大规模的比赛应该有技术统计，在 XianGe 的比赛中所有车辆的起始点可能不同，速度当然也会有差异。XianGe 想知道比赛中会出现多少次超车（如果两辆车起点相同速度不同也算发生一次超车）。

输入：

本题有多组测试数据，第一行一个整数 n，代表参赛人数，接下来 n 行，每行输入两个数据，车辆起始位置 Xi 和速度 Vi（0<Xi,Vi<1000000）

输出：

输出比赛中超车的次数，每组输出占一行。

思路：

我们可以这样考虑，位置靠前的速度慢的车肯定会被位置靠后速度快的车超过，如果我们按照车的位置从小到大排序，并按照这个顺序不断的用树状数组统计，对于当前位置为 i 的车，它肯定是会被位置比 i 小，速度快的车超过，我们用树状数组统计出车速从 1 到车 i 的速度 vi 之间有多少量车 s，因为当前统计了多少量车已经知道，即位置为 i 的车就是加入统计的第 i 量车，有 i-s 量车会超过车 i，统计后输出这个结果即可。

C 代码：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define DATSIZ 100005
#define lowbit(x) ((x)&(-(x)))

typedef struct
{
   int x, v;
}CAR;

CAR car[DATSIZ];
int bit[DATSIZ*10];

int cmp(const void* a, const void* b)
{
   CAR* c = (CAR*)a;
   CAR* d = (CAR*)b;
   if(c->x == d->x) return d->v - c->v;
   return c->x - d->x;
}

void update(int bit[], int pos, int dt, int nMax)
{
   int i;
   for(i = pos; i <= nMax; i += lowbit(i)) {
      bit[i] += dt;
   }
}

int getsum(int bit[], int pos)
{
   int res = 0;
   while(pos > 0) {
      res += bit[pos];
      pos -= lowbit(pos);
   }
   return res;
}

int main(void)
{
   int n, i, nMax;
   long long sum;
```

```
    while(scanf("%d", &n) != EOF) {
        memset(bit, 0, sizeof(bit));
        nMax = 0;
        for(i = 1; i <= n; i++) {
            scanf("%d%d", &car[i].x, &car[i].v);
            if(nMax < car[i].v) nMax = car[i].v;
        }
        qsort(car+1, n, sizeof(car[0]), cmp);
        sum = 0;
        for(i = 1; i <= n; i++) {
            update(bit, car[i].v, 1, nMax);
            sum += i - getsum(bit, car[i].v);
        }
        printf("%lld\n", sum);
    }
    return 0;
}
```

1.4.2.3 hrbustoj 1161 leyni

题意：

Leyni 被人掳走，身在水深火热之中...

小奈叶为了拯救 Leyni，独自一人前往森林深处从静竹手中夺回昏迷中的 Leyni。

历经千辛万苦，小奈叶救出了 Leyni，但是静竹为此极为恼怒，决定对他们发起最强烈的进攻。

不过小奈叶有一个叫做能量保护圈的道具，可以保护他们。

这个保护圈由 n 个小的小护盾围成一圈，从 1 到 n 编号。当某一块小护盾受到攻击的时候，小护盾就会抵消掉这次攻击，也就是说对这一块小护盾的攻击是无效攻击，从而保护圈里的人，不过小护盾在遭到一次攻击后，需要 t 秒进行冷却，在冷却期间受到的攻击都是有效攻击,此时他们就会遭到攻击， 即假设 1 秒时受到攻击并成功防御，到 1+t 秒时冷却才结束并能进行防御，在 2 到 t 受到的都是有效攻击。

现在小奈叶专心战斗，Leyni 昏迷，他们无法得知小护盾遭受的有效攻击次数，他们需要你的帮助。

输入：

第一行是一个整数 T，表示有多少组测试数据。

第一行是三个整数，n,q,t，n 表示保护圈的长度，q 表示攻击的询问的总次数，t 表示能量盾的冷却时间。

接下来的 q 行，每行表示受到的攻击或者她询问某范围内的能量盾被攻击的次数。

攻击：

Attack a

表示编号为 a 的小护盾受到一次攻击， 保证 1 <= a <= n

询问：

Query a b

表示询问编号从 a 到 b 的小护盾(包括 a 和 b)总共受到了多少次有效攻击。保证 1<=a,b<=n

第 k 次攻击发生在第 k 秒，询问不花费时间。

1 <= n,q <=100000

1 <= t <= 50。

输出：

每一组测试数据，先输出一行"Case i:",i 表示第 i 组测试数据，从 1 开始计数。之后对于每一个询问，输出该范围内的小护盾受到的有效攻击次数，一个询问一行。

思路：

题目要对每个查询输出某个护盾承受了多少次的有效攻击，即没有保护罩的时候。因为每个护盾有冷却时间，冷却时间内的攻击都是有效攻击，我们用一个树状数组统计护盾受到的攻击次数；还要记录每个护盾的最后开始冷却时间，也就是最后一次成功防御的时间，每次攻击某快护盾的时候，先检查这块护盾的冷却时间，如果冷却时间已经过了，说明可以承受这次攻击，如果冷却时间没过，则受到一次有效攻击。

C++代码：

```
#include <stdio.h>
#include <string.h>

const int maxn = 100000;
const int maxt = 50;

int c[maxn+1];
int n, q, t;
int p[maxn+1];

inline int lowbit(int x) {
    return (x)&(-x);
}
int init() {
    memset(c, 0, sizeof(c));
}

void add(int x, int v) {
    int i = x;
    for (; x <= n; x += lowbit(x)) {
        c[x] += v;
    }
}
int getSum(int x) {
    int s = 0;
    for (; x > 0; x -= lowbit(x)) {
        s += c[x];
    }
    return s;
}

int main()
{
    int Case, T;
    char cmd[16];
    int attack_cnt;
    int i, a, b;

    scanf("%d", &T);
```

```
    for (Case = 1; Case <= T; ++Case) {
        scanf("%d %d %d", &n, &q, &t);
        //printf("%d %d %d\n", n, q, t);
        for (i = 0; i <= n; ++i) {
            p[i] = -t;
        }
        printf("Case %d:\n", Case);
        init();
        attack_cnt = 0;
        for (i = 0; i < q; ++i) {
            scanf("%s", cmd);
            if (cmd[0] == 'A') {
                attack_cnt++;
                scanf("%d", &a);
                if (p[a]+t <= attack_cnt) {
                    p[a] = attack_cnt;
                } else {
                    add(a, 1);
                }
            } else if (cmd[0] == 'Q'){
                scanf("%d %d", &a, &b);
                if (a > b) {
                  int t = a;
                  a = b;
                  b = t;
                }
                printf("%d\n", getSum(b) - getSum(a-1));
            }
        }
    }
    return 0;
}
```

1.4.3 其他推荐例题

POJ 2085 Inversion

此题要想出一个构造方法构造出答案，构造时会用到二分和树状数组找第 k 大的数。

POJ 1195 Mobile phones

二维树状数组的应用。

hrbustoj 1451 Imagine

二维树状数组的应用

1.5 线段树

扩展阅读：

胡浩的博客：http://www.notonlysuccess.com/index.php/segment-tree-complete/

编写：卢俊达　　校核：黄李龙

1.1.1 线段树的介绍

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。

对于线段树中的每一个非叶子节点[a,b]，它的左儿子表示的区间为[a,(a+b)/2]，右儿子表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的子节点数目为 N，即整个线段区间的长度。

上面介绍的是基本的线段树结构，但只有这些并不能做什么，就好比一个程序有输入没输出，根本没有任何用处。

最简单的应用就是记录线段有否被覆盖，并随时查询当前被覆盖线段的总长度。那么此时可以在结点结构中加入一个变量 int count；代表当前结点代表的子树中被覆盖的线段长度和。这样就要在插入（删除）当中维护这个 count 值，于是当前的覆盖总值就是根节点的 count 值了。

另外也可以用于查找一个有序数组给定区间里的最大值或者对该区间求和。

上面两种应用属于初级的线段树应用，多数是较难的应用，例如对有序数组进行区间成段更新、区间成段加减等等的操作。这些较难应用一般需要利用到“延迟标记”。

延迟标记的作用是延迟对子区间的更新，优点是更新操作不必“进行到底”。例如将上图中的有序数组的下标在区间[1,3]上的数全部加 1，再全部减 7。利用延时标记思想则两次更新均只需更新到第二层，延迟标记将记录操作“减 6”，当查询操作涉及的区间在[1,3]内时再执行更新操作。

1.1.2 线段树模板代码

```
#define MAXSIZE 200000
int val[MAXSIZE+1];
struct node
{
    int max;
    int left;
    int right;
}tree[MAXSIZE*3];
int max(int x,int y)
{
    return x>y?x:y;
}
int create(int root,int left,int right)
//以 root 为根节点建树。
{
    tree[root].left=left;
    tree[root].right=right;
    if(left==right)
        return tree[root].max=val[left];
    int a,b,middle=(left+right)/2;
```

```
        a=create(2*root,left,middle);
        b=create(2*root+1,middle+1,right);
        return tree[root].max=max(a,b);
    }
    int calculate(int root,int left,int right)
    //以 root 为根节点的线段树中，求取区间[left,right]中的最大值。
    {
        if(tree[root].left>right||tree[root].right<left)
            return 0;
        if(left<=tree[root].left&&tree[root].right<=right)
            return tree[root].max;
        int a,b;
        a=calculate(2*root,left,right);
        b=calculate(2*root+1,left,right);
        return max(a,b);
    }
    int updata(int root,int pos,int val)
    //以 root 为根节点的线段树中，将位置 pos 的元素更新为 val。
    {
        if(pos<tree[root].left||tree[root].right<pos)
            return tree[root].max;
        if(tree[root].left==pos&&tree[root].right==pos)
            return tree[root].max=val;
        int a,b;
        a=updata(2*root,pos,val);
        b=updata(2*root+1,pos,val);
        return tree[root].max=max(a,b);
    }
```

1.1.3 经典题目

1.1.3.1 HDU 1754 I Hate It

1．题目出处/来源

[HDU][1754][线段树] I Hate It

2．题目描述

给定一个序列和两种操作：

Q 操作，表示这是一条询问操作，询问区间[a,b]当中的元素的最大值是多少。

U 操作，表示这是一条更新操作，要求把元素 A 的值更改为 B。

3．分析

区间节点内封装变量 max 用于记录该节点所表示区间的最大值。更新和查询操作均返回区间最大值。本题意在考验对线段树的基本应用。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define MAXSIZE 200000
int val[MAXSIZE+1];
struct node
{
    int max;
    int left;
    int right;
}tree[MAXSIZE*3];
int max(int x,int y)
```

```
{
    return x>y?x:y;
}
int create(int root,int left,int right)
//以 root 为根节点建树。
{
    tree[root].left=left;
    tree[root].right=right;
    if(left==right)
        return tree[root].max=val[left];
    int a,b,middle=(left+right)/2;
    a=create(2*root,left,middle);
    b=create(2*root+1,middle+1,right);
    return tree[root].max=max(a,b);
}
int calculate(int root,int left,int right)
//以 root 为根节点的线段树中，求取区间[left,right]中的最大值。
{
    if(tree[root].left>right||tree[root].right<left)
        return 0;
    if(left<=tree[root].left&&tree[root].right<=right)
        return tree[root].max;
    int a,b;
    a=calculate(2*root,left,right);
    b=calculate(2*root+1,left,right);
    return max(a,b);
}
int updata(int root,int pos,int val)
//以 root 为根节点的线段树中，将位置 pos 的元素更新为 val。
{
    if(pos<tree[root].left||tree[root].right<pos)
        return tree[root].max;
    if(tree[root].left==pos&&tree[root].right==pos)
        return tree[root].max=val;
    int a,b;
    a=updata(2*root,pos,val);
    b=updata(2*root+1,pos,val);
    return tree[root].max=max(a,b);
}
int main()
{
    int n,m;
    while(~scanf("%d%d",&n,&m))
    {
        for(int i=1;i<=n;i++)
            scanf("%d",&val[i]);
        create(1,1,n);
        for(int i=0;i<m;i++)
        {
            char op;
            int a,b;
            scanf("\n%c%d%d",&op,&a,&b);
            if(op=='Q')
                printf("%d\n",calculate(1,a,b));
            else
                updata(1,a,b);
        }
    }
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

1.1.3.2 HDU 1698 Just a Hook

1．题目出处/来源

[HDU][1698][线段树] Just a Hook

2．题目描述

一个长度为 n 的序列，初始化序列中的元素为 1。接下来执行 Q 次更新操作，每次操作将区间[x,y]上的值更新为 z。所有操作结束后，求出序列的总和。

3．分析

考察队线段树的成段更新。本题需要用到延迟标记，属于对延迟标记的最基本应用。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define MAXSIZE 100000
int val[MAXSIZE+1];
struct node
{
    int total;
    //区间属性，这里是 total，表示区间元素和。
    int left;
    int right;
    int mark;
    //mark 表示延迟标记
}tree[MAXSIZE*3];
int create(int root,int left,int right)
//以 root 为根节点建树。
{
    tree[root].mark=0;
    tree[root].left=left;
    tree[root].right=right;
    if(left==right)
        return tree[root].total=val[left];
    int middle=(left+right)/2;
    return tree[root].total=create(2*root,left,middle)+create(2*root+1,middle+1,right);
}
void updata_mark(int root)
//更新 root 的子节点的延迟标记。
{
    if(tree[root].mark)
    {
        tree[root].total=tree[root].mark*(tree[root].right-tree[root].left+1);
        if(tree[root].left!=tree[root].right)
            tree[root*2].mark=tree[root*2+1].mark=tree[root].mark;
        tree[root].mark=0;
    }
}
int calculate(int root,int left,int right)
//以 root 为根节点的线段树中，求取区间[left,right]的元素和。
{
    updata_mark(root);
    if(tree[root].left>right||tree[root].right<left)
        return 0;
    if(left<=tree[root].left&&tree[root].right<=right)
```

```
            return tree[root].total;
        return calculate(2*root,left,right)+calculate(2*root+1,left,right);
    }
    int updata(int root,int left,int right,int val)
    //以 root 为根节点的线段树中，将区间[left,right]中的元素更新为 val。
    {
        updata_mark(root);
        if(tree[root].left>right||tree[root].right<left)
            return tree[root].total;
        if(left<=tree[root].left&&tree[root].right<=right)
        {
            tree[root].mark=val;
            return tree[root].total=val*(tree[root].right-tree[root].left+1);
        }
        return tree[root].total=updata(2*root,left,right,val)+updata(2*root+1,left,right,val);
    }
    int main()
    {
        int t;
        scanf("%d",&t);
        for(int i=1,n,q;i<=t;i++)
        {
            scanf("%d%d",&n,&q);
            for(int j=1;j<=n;j++)
                val[j]=1;
            create(1,1,n);
            for(int j=0,x,y,z;j<q;j++)
            {
                scanf("%d%d%d",&x,&y,&z);
                updata(1,x,y,z);
            }
            printf("Case %d: The total value of the hook is %d.\n",i,calculate(1,1,n));
        }
    }
```

5．思考与扩展：可以借鉴北大培训教材中做法。

1.1.3.3 POJ 3468 A Simple Problem with Integers

1．题目出处/来源

[POJ][3468][线段树] A Simple Problem with Integers

2．题目描述

你有 n 个整数，A1、A2、…、AN。你需要处理两种操作。

将给定范围内的每个数字加上一个给定的值。

求一个给定范围内的数字的总和。

3．分析

线段树成段增减的模板题，考研对延迟标记的应用能力。这里对延迟标记的操作与成段更新稍有不同。在更新延迟标记后，需要执行一次 updata_mark()操作，意在更新当前节点，因为更新操作需要返回当前节点的最新信息。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define MAXSIZE 100000
int total_edge,val[MAXSIZE+1];
```

```
struct node
{
    long long total;
    //区间属性，这里是 total，表示区间元素和。
    int left;
    int right;
    long long mark;
    //mark 表示延迟标记
}tree[MAXSIZE*3];
long long create(int root,int left,int right)
//以 root 为根节点建树。
{
    tree[root].mark=0;
    tree[root].left=left;
    tree[root].right=right;
    if(left==right)
        return tree[root].total=val[left];
    int middle=(left+right)/2;
    return tree[root].total=create(2*root,left,middle)+create(2*root+1,middle+1,right);
}
void updata_mark(int root)
//更新 root 的子节点的延迟标记。
{
    if(tree[root].mark)
    {
        tree[root].total+=tree[root].mark*(tree[root].right-tree[root].left+1);
        if(tree[root].left!=tree[root].right)
        {
            tree[root*2].mark+=tree[root].mark;
            tree[root*2+1].mark+=tree[root].mark;
        }
        tree[root].mark=0;
    }
}
long long calculate(int root,int left,int right)
//以 root 为根节点的线段树中，求取区间[left,right]的元素和。
{
    updata_mark(root);
    if(tree[root].left>right||tree[root].right<left)
        return 0;
    if(left<=tree[root].left&&tree[root].right<=right)
        return tree[root].total;
    return calculate(2*root,left,right)+calculate(2*root+1,left,right);
}
long long updata(int root,int left,int right,int val)
//以 root 为根节点的线段树中，将区间[left,right]中的元素加上 val。
{
    updata_mark(root);
    if(tree[root].left>right||tree[root].right<left)
        return tree[root].total;
    if(left<=tree[root].left&&tree[root].right<=right)
    {
        tree[root].mark+=val;
        updata_mark(root);
        return tree[root].total;
    }
    return tree[root].total=updata(2*root,left,right,val)+updata(2*root+1,left,right,val);
}
```

```
int main()
{
    int n,q;
    while(~scanf("%d%d",&n,&q))
    {
        for(int i=1;i<=n;i++)
            scanf("%d",&val[i]);
        create(1,1,n);
        char op;
        for(int i=0;i<q;i++)
        {
            scanf("\n%c",&op);
            if(op=='Q')
            {
                int a,b;
                scanf("%d%d",&a,&b);
                printf("%I64d\n",calculate(1,a,b));
            }
            else
            {
                int a,b,c;
                scanf("%d%d%d",&a,&b,&c);
                updata(1,a,b,c);
            }
        }
    }
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

1.1.3.4 POJ 3832 Posters

1．题目出处/来源

[POJ][3832][线段树] Posters

2．题目描述

平面上有一些“回型”图案，每一个“回型”是由一个大矩形中间挖去一个小矩形构成，大小矩形的四边都平行于坐标轴。

现在有 n 个不同大小的“回型”图案，他们可能互相重叠，请求出被他们所覆盖的平面的总面积。

3．分析

首先，将每个一个回型划分为四个矩形。将每个矩形的与 x 轴平行的边按离 x 轴距离由近及远顺序插入线段树中（底边插入，顶边删除）。

在对每条线段执行插入之前，先计算下当前线段与上一条线段之间的覆盖面积。用 tree[1].len*当前线段与前一个线段之间的距离（高度差），tree[1].len 表示图形的宽。

当遍历完所有线段时，就会得出覆盖面积。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define MAXSIZE 50000
#define N 50000
struct node
{
    int total;
```

```
    //total 表示该节点所表示的区间被线段覆盖的次数
    int left;
    int right;
    int len;
    //该区间内，线段的长度。
}tree[MAXSIZE*3];
struct line
{
    int pos;
    //pos 表示 y 轴坐标
    int left;
    int right;
    bool top;
    //判断该线段是否是矩形的上边界。
}edge[N*8];
int total_edge,n;
//一共 total_edge 个线段，n 个回型。
void create(int root,int left,int right)
//建树，明确每个节点对应的区间
{
    tree[root].left=left;
    tree[root].right=right;
    if(left==right)
        return ;
    int middle=(left+right)/2;
    create(2*root,left,middle);
    create(2*root+1,middle+1,right);
}
void updata(int root,int left,int right,int val)
//将线段插入或删除。
{
    if(tree[root].left>right||tree[root].right<left)
        return ;
    else if(left<=tree[root].left&&tree[root].right<=right)
        tree[root].total+=val;
    else
    {
        updata(2*root,left,right,val);
        updata(2*root+1,left,right,val);
    }
    if(tree[root].total>0)
        tree[root].len=tree[root].right-tree[root].left+1;
    else if(tree[root].right==tree[root].left)
        tree[root].len=0;
    else
        tree[root].len=tree[2*root].len+tree[2*root+1].len;
}
void add_edge(int pos,int left,int right,bool top)
//记录一条线段的信息
{
    if(left>right)
        return;
    edge[total_edge].pos=pos;
    edge[total_edge].left=left;
    edge[total_edge].right=right;
    edge[total_edge].top=top;
    total_edge++;
}
int cmp(const void* m,const void* n)
```

```
//将线段按照 y 轴坐标由小到大排序。
{
    return (*(line*)m).pos-(*(line*)n).pos;
}
void init()
//初始化及输入
{
    total_edge=0;
    for(int i=1;i<MAXSIZE*3;i++)
    {
        tree[i].len=0;
        tree[i].total=0;
    }
    for(int i=0;i<n;i++)
    {
        int x1,y1,x2,y2,x3,y3,x4,y4;
        scanf("%d%d%d%d%d%d%d%d",&x1,&y1,&x2,&y2,&x3,&y3,&x4,&y4);
        add_edge(y1+1,x1+1,x3,false);
        add_edge(y2+1,x1+1,x3,true);
        add_edge(y1+1,x3+1,x4,false);
        add_edge(y3+1,x3+1,x4,true);
        add_edge(y4+1,x3+1,x4,false);
        add_edge(y2+1,x3+1,x4,true);
        add_edge(y1+1,x4+1,x2,false);
        add_edge(y2+1,x4+1,x2,true);
    }
}
int main()
{
    create(1,1,MAXSIZE+1);
    while(scanf("%d",&n)&&n)
    {
        init();
        qsort(edge,total_edge,sizeof(edge[0]),cmp);
        long long total_size=0;
        for(int i=0,pre=1;i<total_edge;i++)
        {
            total_size+=(long long)tree[1].len*(edge[i].pos-pre);
            //这里注意，tree[1].len 必须转化为 long long 型
            updata(1,edge[i].left,edge[i].right,edge[i].top?-1:1);
            pre=edge[i].pos;
        }
        printf("%I64d\n",total_size);
    }
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

1.6 随机平衡二叉查找树

参考文献：

清华大学计算机科学与技术系・郭家宝《随机平衡二叉查找树 Treap 的分析与应用》

数据结构之Treap[董的博客]：http://dongxicheng.org/structure/treap/

编写：黄李龙　　校核：黄李龙

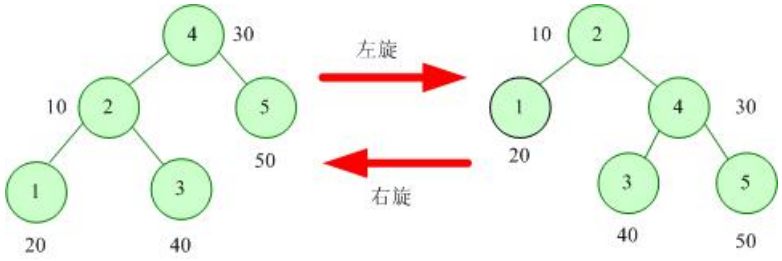
1.6.1 概述

同 splay tree 一样，treap 也是一个平衡二叉树，不过 Treap 会记录一个额外的数据，即优先级。Treap 在以关键码构成二叉搜索树的同时，还按优先级来满足堆的性质。因而，Treap=tree+heap。这里需要注意的是，Treap 并不是二叉堆，二叉堆必须是完全二叉树，而 Treap 可以并不一定是。

1.6.2 Treap 基本操作

为了使 Treap 中的节点同时满足 BST 性质和最小堆性质，不可避免地要对其结构进行调整，调整方式被称为旋转。在维护 Treap 的过程中，只有两种旋转，分别是左旋转(简称左旋)和右旋转(简称右旋)。

左旋一个子树，会把它的根节点旋转到根的左子树位置，同时根节点的右子节点成为子树的根；右旋一个子树，会把它的根节点旋转到根的右子树位置，同时根节点的左子节点成为子树的根。




```
struct Treap_Node
{

  Treap_Node *left,*right; //节点的左右子树的指针
  int value,fix; //节点的值和优先级
};

void Treap_Left_Rotate(Treap_Node *&a) //左旋 节点指针一定要传递引用
 {
      Treap_Node *b=a->right;
      a->right=b->left;
      b->left=a;
      a=b;
}
 void Treap_Right_Rotate(Treap_Node *&a) //右旋 节点指针一定要传递引用
 {
      Treap_Node *b=a->left;
      a->left=b->right;
       b->right=a;
      a=b;
}
```

1.6.3 Treap 的操作

同其他树形结构一样，treap 的基本操作有：查找，插入，删除等。

查找

同其他二叉树一样，treap 的查找过程就是二分查找的过程，复杂度为 O(lg n)。

1.6.3.1 插入

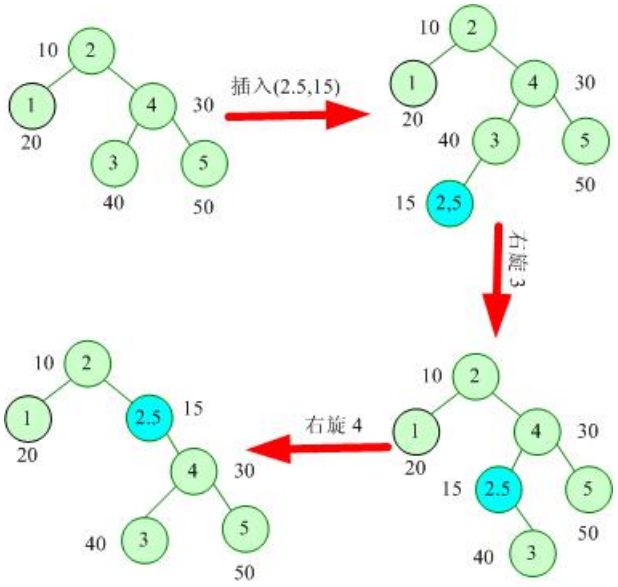
在 Treap 中插入元素，与在 BST 中插入方法相似。首先找到合适的插入位置，然后建立新的节点，存储元素。但是要注意新的节点会有一个优先级属性，该值可能会破坏堆序，因此我们要根据需要进行恰当的旋转。具体方法如下：

1. 从根节点开始插入；

2. 如果要插入的值小于等于当前节点的值，在当前节点的左子树中插入，插入后如果左子节点的优先级小于当前节点的优先级，对当前节点进行右旋；

3. 如果要插入的值大于当前节点的值，在当前节点的右子树中插入，插入后如果右子节点的优先级小于当前节点的优先级，对当前节点进行左旋；

4. 如果当前节点为空节点，在此建立新的节点，该节点的值为要插入的值，左右子树为空，插入成功。




```
Treap_Node *root;
void Treap_Insert(Treap_Node *&P,int value) //节点指针一定要传递引用
 {
   if (!P) //找到位置，建立节点
  {
    P=new Treap_Node;
    P->value=value;
    P->fix=rand();//生成随机的修正值
  }
  else if (value <= P->value)
  {
    Treap_Insert(P->left,r);
    if (P->left->fix < P->fix)
      Treap_Right_Rotate(P);//左子节点修正值小于当前节点修正值，右旋当前节点
   }
  else
  {
    Treap_Insert(P->right,r);
    if (P->right->fix < P->fix)
      Treap_Left_Rotate(P);//右子节点修正值小于当前节点修正值，左旋当前节点
   }
}
```

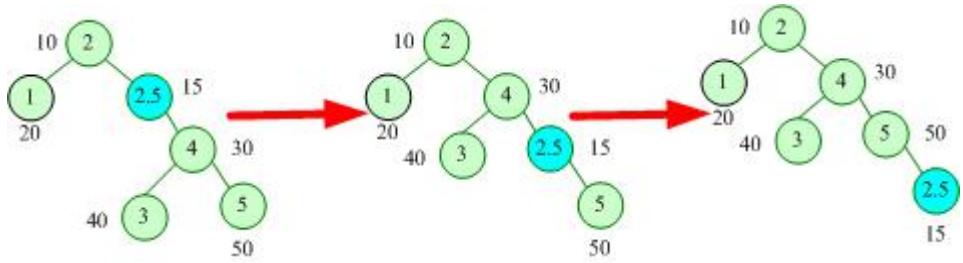
1.6.3.2 删除

与 BST 一样，在 Treap 中删除元素要考虑多种情况。我们可以按照在 BST 中删除元素同样的方法来删除 Treap 中的元素，即用它的后继(或前驱)节点的值代替它，然后删除它的后继(或前驱)节点。

上述方法期望时间复杂度为 O(logN)，但是这种方法并没有充分利用 Treap 已有的随机性质，而是重新得随机选取代替节点。我们给出一种更为通用的删除方法，这种方法是基于旋转调整的。首先要在 Treap 树中找到待删除节点的位置，然后分情况讨论：

情况一，该节点为叶节点或链节点，则该节点是可以直接删除的节点。若该节点有非空子节点，用非空子节点代替该节点的，否则用空节点代替该节点，然后删除该节点。

情况二，该节点有两个非空子节点。我们的策略是通过旋转，使该节点变为可以直接删除的节点。如果该节点的左子节点的优先级小于右子节点的优先级，右旋该节点，使该节点降为右子树的根节点，然后访问右子树的根节点，继续讨论；反之，左旋该节点，使该节点降为左子树的根节点，然后访问左子树的根节点，这样继续下去，直到变成可以直接删除的节点。




```
BST_Node *root;
void Treap_Delete(Treap_Node *&P,int *value) //节点指针要传递引用
{
  if (value==P->value) //找到要删除的节点 对其删除
  {
    if (!P->right || !P->left) //情况一，该节点可以直接被删除
    {
      Treap_Node *t=P;
      if (!P->right)
        P=P->left; //用左子节点代替它
      else
        P=P->right; //用右子节点代替它
      delete t; //删除该节点
    }
     else //情况二
    {
      if (P->left->fix < P->right->fix) //左子节点修正值较小，右旋
      {
        Treap_Right_Rotate(P);
        Treap_Delete(P->right,r);
      }
      else //左子节点修正值较小，左旋
      {
        Treap_Left_Rotate(P);
         Treap_Delete(P->left,r);
       }
    }
  }
   else if (value < P->value)
```

```
    Treap_Delete(P->left,r); //在左子树查找要删除的节点
  else
    Treap_Delete(P->right,r); //在右子树查找要删除的节点
}
```

1.7 Treap 应用

Treap 可以解决 splay tree 可以解决的所有问题，具体参见另一篇文章：《数据结构之伸展树》http://dongxicheng.org/structure/splay-tree/

可以这样定义结构体：

```
struct Treap_Node
{
  Treap_Node *left,*right; //节点的左右子树的指针
  int value,fix,weight,size; //节点的值，优先级，重复计数（记录相同节点个数，节省空间），子树大小
  inline int lsize(){ return left ?left->size ?0; } //返回左子树的节点个数
  inline int rsize(){ return right?right->size?0; } //返回右子树的节点个数
};
```

1.8 总结

Treap 作为一种简洁高效的有序数据结构，在计算机科学和技术应用中有着重要的地位。它可以用来实现集合、多重集合、字典等容器型数据结构，也可以用来设计动态统计数据结构。

以上内容都是一些介绍，写得不是很详尽，非常推荐清华大学计算机科学与技术系·郭家宝《随机平衡二叉查找树 Treap 的分析与应用》的论文，要学好 Treap 这篇论文非常值得一看。

1.8.1 经典题目

1.8.1.1 POJ 1442 Black Box

题意：
我们有一个黑色盒子，有两种操作：
ADD(X)：把整数 X 放入黑色盒子中；
GET：先把 i 加 1，然后求第 i 小的数，在 GET 操作前，i 初始为 0。
下面是操作的示范：

1	ADD(3)	0 3	
2	GET	1 3	3
3	ADD(1)	1 1, 3	
4	GET	2 1, 3	3
5	ADD(-4)	2 -4, 1, 3	
6	ADD(2)	2 -4, 1, 2, 3	
7	ADD(8)	2 -4, 1, 2, 3, 8	
8	ADD(-1000)	2 -1000, -4, 1, 2, 3, 8	
9	GET	3 -1000, -4, 1, 2, 3, 8	1

10 GET 4 -1000, -4, 1, 2, 3, 8 2
11 ADD(2) 4 -1000, -4, 1, 2, 2, 3, 8

输入

第一行是两个整数 M 和 N。第二行是 M 的整数，表示按顺序添加的 M 个数，即执行 ADD 操作分别是 A1、A2 ••• AM。第三行有 N 个数正整数，分别是 u1，u2 ••• uN，表示当黑盒内有 ui 个数的时候，执行 GET 操作，要输出第 i 小的数。

输出

对已每次 GET 操作，输出其结果。

分析：求第 k 小的数，直接套 Treap 树模板即可。注意判断当前添加数后，是否能输出一个 GET 操作的结果即可。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

const int MAXN = 30005;

struct TreapNode {
 TreapNode* ch[2];
 int sz, fix;
 int val;
 int ch_sz(int c) { return ch[c] ? ch[c]->sz : 0; }
};

template <const int MAXN = 100005>
struct Treap {
 TreapNode *root;
 int mcnt, pcnt;
 TreapNode mem[MAXN], *pool[MAXN];

 TreapNode * new_node(int val = 0) {
     TreapNode *x;
     if (pcnt >= 0) x = pool[pcnt--];
     else x = mem + mcnt++;
     x->ch[0] = x->ch[1] = NULL;
     x->sz = 1;
     x->fix = rand();
     x->val = val;
     return x;
 }
 /* c=0 左旋，c=1 右旋 */
 void rotate(TreapNode* &x, int c) {
     TreapNode * y = x->ch[!c];
     x->ch[!c] = y->ch[c];
     y->ch[c] = x;
     x = y;
     y = x->ch[c];
     y->sz = y->ch_sz(0) + y->ch_sz(1) + 1;
     x->sz = x->ch_sz(0) + x->ch_sz(1) + 1;
 }

 void init() {
     mcnt = 0;
     pcnt = -1;
     root = NULL;
 }
```

```
void insert(TreapNode* &x, int val) {
    if (x == NULL) {
        x = new_node(val);
    } else if (val <= x->val) {
        insert(x->ch[0], val);
        x->sz++;
        if (x->ch[0]->fix < x->fix) rotate(x, 1);
    } else {
        insert(x->ch[1], val);
        x->sz++;
        if (x->ch[1]->fix < x->fix) rotate(x, 0);
    }
}
void insert(int val) {
    insert(root, val);
}

bool del(TreapNode* &x, int val) {
    if (NULL == x) return false;
    bool ret;
    if (x->val < val) {
        ret = del(x->ch[1], val);
    } else if (x->val > val) {
        ret = del(x->ch[0], val);
    } else {
        if (NULL == x->ch[0] || NULL == x->ch[1]) {
            pool[++pcnt] = x;/* 放入内存池进行删除 */
            if (NULL != x->ch[0]) x = x->ch[0];
            else x = x->ch[1];
            ret = true;
        } else {
            if (x->ch[0]->fix < x->ch[1]->fix) {
                rotate(x, 1);
                ret = del(x->ch[1], val);
            } else {
                rotate(x, 0);
                ret = del(x->ch[0], val);
            }
        }
    }
    if (ret) x->sz--;
    return ret;
}

int get_rank(TreapNode* x, int val, int cur) {
    if (val == x->val) {
        return x->ch_sz(0) + cur + 1;
    } else if (val < x->val) {
        return get_rank(x->ch[0], val, cur);
    } else {
        return get_rank(x->ch[1], val, cur + x->ch_sz(0) + 1);
    }
}
TreapNode* get_kth(TreapNode* x, int k) {
    int lsz = x->ch_sz(0);
    if (k <= lsz) {
        return get_kth(x->ch[0], k);
    } else if (k > lsz + 1) {
        return get_kth(x->ch[1], k - (lsz + 1));
    } else {
        return x;
    }
}
int get_kth(int k) {
    return get_kth(root, k)->val;
}
int size() {
    return root->sz;
```

```
 }
};

int a[MAXN], u[MAXN];
Treap<MAXN> tr;

int main() {
 int m, n;
 srand(time(0));
 while (EOF != scanf("%d%d", &m, &n)) {
     for (int i = 0; i < m; ++i) scanf("%d", &a[i]);
     for (int i = 0; i < n; ++i) scanf("%d", &u[i]);
     sort(u, u + n);
     tr.init();
     int j = 0, g = 0;
     for (int i = 0; i < m; ++i) {
         tr.insert(a[i]);
         int sz = tr.size();
         while (j < n && sz == u[j]) {
             ++g;
             printf("%d\n", tr.get_kth(g));
             ++j;
         }
         if (j == n) break;
     }
 }
 return 0;
}
```

1.8.1.2 POJ 3481 Double Queue

题意：在一个银行中，每天都会有客户来办理业务。每个客户用一个正整数 K 标识，因为银行人手不多，客户在办理时需要等待其他用户办理。在客户办理业务前，银行为用户提供一个优先级 P，根据优先级来确定先处理哪个用户的业务，并且有时候是优先级高的先服务，有时是优先级的先服务。银行系统将对客户的服务分为以下几种请求：

0	停止系统的服务
1 K P	加入 K 用户的业务请求，优先级为 P
2	先服务一个最高优先级的客户
3	先服务一个最低优先级的客户

输入：

每一行为一个请求，并且最后一行的请求为 0，表示停止服务。输入保证有类型为 1 的请求，每个客户的标识都不一样，并且没有两个请求的优先级是一样的。客户标识小于 1000000，优先级 P 小于 10000000，一个客户可以有多次请求，每次请求的优先级都不同。

输出：

对于每个类型为 2 或者 3 的请求，输出服务的客户标识。如果没有客户在请求，则输出 0。

思路：

Treap 树能查询树中数的最大值的和最小值，我们可以根据每个用户的优先级 P 进行比较来构造 Treap 树，插入和删除节点也是根据 P 进行。剩下的工作就是查找和输出了。

直接套用 Treap 书模板就行了。

```
/*
 * POJ 3481 Double Queue
```

```
 * Treap 树解
 * */
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <ctime>
#include <utility>
using namespace std;

typedef pair<int, int> PII;

const int MAXN = 1000006;

struct TreapNode {
 TreapNode *ch[2];
 int sz, fix;
 int lsz() { return ch[0] ? ch[0]->sz : 0; }
 int rsz() { return ch[1] ? ch[1]->sz : 0; }
 int val, id;
};
struct Treap {
 int mcnt, pcnt;
 TreapNode *root, mem[MAXN], *pool[MAXN];

 void init() {
     mcnt = 0;
     pcnt = -1;
     root = NULL;
 }

 TreapNode * new_node(int val, int id) {
     TreapNode* x;
     if (pcnt >= 0) x = pool[pcnt--];
     else x = mem + mcnt++;
     x->ch[0] = x->ch[1] = NULL;
     x->sz = 1;
     x->fix = rand();
     x->val = val;
     x->id = id;
     return x;
 }

 void rotate(TreapNode* &x, int c) {
     TreapNode* y = x->ch[!c];
     x->ch[!c] = y->ch[c];
     y->ch[c] = x;
     x = y;
     y = x->ch[c];
     y->sz = y->lsz() + y->rsz() + 1;
     x->sz = x->lsz() + x->rsz() + 1;
 }

 void insert(TreapNode* &x, int val, int id) {
     if (x == NULL) {
         x = new_node(val, id);
     } else {
         int c = x->val < val;
         insert(x->ch[c], val, id);
         x->sz++;
         if (x->fix > x->ch[c]->fix) rotate(x, !c);
     }
 }
 void insert(int val, int id) {
     insert(root, val, id);
 }

 PII del_max(TreapNode * &x) {
     PII ii;
     if (NULL == x->ch[1]) {
```

```
            ii = make_pair(x->val, x->id);
            pool[++pcnt] = x;
            if (x->ch[0]) x = x->ch[0];
            else x = NULL;
        } else {
            ii = del_max(x->ch[1]);
            x->sz--;
        }
        return ii;
    }
    PII del_max() {
        if (root) return del_max(root);
        else return make_pair(0, 0);
    }
    PII del_min(TreapNode * &x) {
        PII ii;
        if (NULL == x->ch[0]) {
            ii = make_pair(x->val, x->id);
            pool[++pcnt] = x;
            if (x->ch[1]) x = x->ch[1];
            else x = NULL;
        } else {
            ii = del_min(x->ch[0]);
            x->sz--;
        }
        return ii;
    }
    PII del_min() {
        if (root) return del_min(root);
        else return make_pair(0, 0);
    }

};
Treap tr;

int main() {
 int cmd;
 tr.init();
 while (EOF != scanf("%d", &cmd)) {
     if (cmd == 0) {
         tr.init();
         continue;
     }
     if (cmd == 1) {
         int k, p;
         scanf("%d%d", &k, &p);
         tr.insert(p, k);
     } else if (cmd == 2) {
         PII ii = tr.del_max();
     //  printf("hig %d\n", ii.first);
         printf("%d\n", ii.second);
     } else if (cmd == 3) {
         PII ii = tr.del_min();
 //      printf("low %d\n", ii.first);
         printf("%d\n", ii.second);
     } else {
     }
 }
 return 0;
}
```

1.8.1.3 NOI 2004 郁闷的出纳员

题目：

OIER 公司是一家大型专业化软件公司，有着数以万计的员工。作为一名出纳员，我的任务之一便是统计每位员工的工资。这本来是一份不错的工作，但是令人郁闷的是，我

们的老板反复无常，经常调整员工的工资。如果他心情好，就可能把每位员工的工资加上一个相同的量。反之，如果心情不好，就可能把他们的工资扣除一个相同的量。我真不知道除了调工资他还做什么其它事情。

工资的频繁调整很让员工反感，尤其是集体扣除工资的时候，一旦某位员工发现自己的工资已经低于了合同规定的工资下界，他就会立刻气愤地离开公司，并且再也不会回来了。每位员工的工资下界都是统一规定的。每当一个人离开公司，我就要从电脑中把他的工资档案删去，同样，每当公司招聘了一位新员工，我就得为他新建一个工资档案。

老板经常到我这边来询问工资情况，他并不问具体某位员工的工资情况，而是问现在工资第 k 多的员工拿多少工资。每当这时，我就不得不对数万个员工进行一次漫长的排序，然后告诉他答案。

好了，现在你已经对我的工作了解不少了。正如你猜的那样，我想请你编一个工资统计程序。怎么样，不是很困难吧？

输入

第一行有两个非负整数 n 和 min。n 表示下面有多少条命令，min 表示工资下界。

接下来的 n 行，每行表示一条命令。命令可以是以下四种之一：

名称	格式	作用
I 命令	I_k	新建一个工资档案，初始工资为 k。如果某员工的初始工资低于工资下界，他将立刻离开公司。
A 命令	A_k	把每位员工的工资加上 k
S 命令	S_k	把每位员工的工资扣除 k
F 命令	F_k	查询第 k 多的工资

_（下划线）表示一个空格，I 命令、A 命令、S 命令中的 k 是一个非负整数，F 命令中的 k 是一个正整数。

在初始时，可以认为公司里一个员工也没有。

输出

输出文件的行数为 F 命令的条数加一。

对于每条 F 命令，你的程序要输出一行，仅包含一个整数，为当前工资第 k 多的员工所拿的工资数，如果 k 大于目前员工的数目，则输出-1。

输出文件的最后一行包含一个整数，为离开公司的员工的总数。

样例输入

```
9 10
I 60
I 70
S 50
F 2
I 30
S 15
A 5
F 1
F 2
```

样例输出

10
20
-1
2

【约定】

- I 命令的条数不超过 100000
- A 命令和 S 命令的总条数不超过 100
- F 命令的条数不超过 100000
- 每次工资调整的调整量不超过 1000
- 新员工的工资不超过 100000

来源：NOI 2004

分析：（参考《随机平衡二叉查找树 Treap 的分析与应用》）与一般的修改不同，这道题要求对所有人修改，如果一个一个进行的话，修改工资的时间复杂度高达 O(N)。如果我们反过来考虑，定义一个“基准值”，把所有人的工资看作“相对工资”，就是相对于基准值。这样每次修改所有人工资仅仅需要修改基准值就行了。于是变成了一个动态统计问题，建立一个 Treap，存储相对工资。为了方便考虑，定义基准值为 delta，相对工资 V 对应的实际工资为 F[V]，则有 F[V]=V+delta，V=F[V]-delta。定义工资下限为 lowbound 这是一个实际的下限，存储相对下限就是 lowbound-delta。

对于 I_k 插入一个新的工资记录值 k，k 为实际工资，对应的相对工资为 k-delta，应把 k-delta 插入 Treap。

对于 A_k，将基准值 delta 增加 k。对于 S_k，将基准值 delta 减少 k，然后在 Treap 中删除所有小于(lowbound-delta)的元素。

由于我们总是查询第 k 多的工资，如果我们是从小到大放置数据，求(总数-k+1)小的工资即可。如果我们是从大到小放置数据，直接求排名为第 k 的数即可。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <ctime>

const int MAXN = 100005;

struct TreapNode {
 TreapNode *ch[2];
 int sz, fix;
 int val;
};

struct Treap {
#define LSZ(x) ((x).ch[0] ? (x).ch[0]->sz : 0)
#define RSZ(x) ((x).ch[1] ? (x).ch[1]->sz : 0)
 int del_cnt, delta;
 int mcnt, pcnt;
 TreapNode *root, mem[MAXN], *pool[MAXN];

 TreapNode *new_node(int val) {
     TreapNode* x;
     if (pcnt >= 0) x = pool[pcnt--];
     else x = mem + mcnt++;
     x->ch[0] = x->ch[1] = NULL;
     x->sz = 1;
     x->fix = rand();
```

```
    x->val = val;
    return x;
}

void rotate(TreapNode* &x, int c) {
    TreapNode *y = x->ch[!c];
    x->ch[!c] = y->ch[c];
    y->ch[c] = x;
    x = y;
    y = x->ch[c];
    y->sz = LSZ(*y) + RSZ(*y) + 1;
    x->sz = LSZ(*x) + RSZ(*x) + 1;
}

void insert(TreapNode* &x, int val) {
    if (x == NULL) {
        x = new_node(val);
    } else {
        int c = x->val < val;
        insert(x->ch[c], val);
        x->sz++;
        if (x->fix > x->ch[c]->fix) rotate(x, !c);
        x->sz = LSZ(*x) + RSZ(*x) + 1;
    }
}

void insert(int val) {
    insert(root, val - delta);
}

void del_low(TreapNode* &x, int val) {
    if (x == NULL) return;
    del_low(x->ch[0], val);
    x->sz = LSZ(*x) + RSZ(*x) + 1;
    if (x->val < val) {
        del_low(x->ch[1], val);
        pool[++pcnt] = x;
        x = x->ch[1];
        del_cnt++;
    }
    if (x) {
        if (x->ch[1] && x->fix > x->ch[1]->fix) {
            rotate(x, 0);
        }
        x->sz = LSZ(*x) + RSZ(*x) + 1;
    }
}
void del_low(int val) {
    del_low(root, val - delta);
}

TreapNode * find_kth(TreapNode*x, int k) {
    if (x == NULL) return NULL;
    int lsz = LSZ(*x);
    if (k <= lsz) {
        return find_kth(x->ch[0], k);
    } else if (k == lsz + 1) {
        return x;
    } else {
        return find_kth(x->ch[1], k - lsz - 1);
    }
}
int find_kth(int k) {
    if (k < 1 || k > size()) return -1;
    TreapNode *x = find_kth(root, k);
    if (x) return x->val + delta;
    return -1;
}
```

```
    int size() {
        if (root) return root->sz;
        return 0;
    }
    void init() {
        mcnt = 0;
        pcnt = -1;
        root = NULL;
        del_cnt = 0;
        delta = 0;
    }
};

Treap tr;

int main() {
    int n, low;
    srand(time(0));
    while (EOF != scanf("%d%d", &n, &low)) {
        int k;
        char s[4];
        tr.init();
        while (n--) {
            scanf("%s%d", s, &k);
            if (s[0] == 'I') {
                if (k >= low) {
                    tr.insert(k);
                }
            } else if (s[0] == 'A') {
                tr.delta += k;
            } else if (s[0] == 'S') {
                tr.delta -= k;
                tr.del_low(low);
            } else if (s[0] == 'F') {
                printf("%d\n", tr.find_kth(tr.size() - k + 1));
            } else {
                for(;;);
            }
        }
        printf("%d\n", tr.del_cnt);
    }
    return 0;
}
```

1.8.2 其他例题

POJ 2761 Feed the dogs

可以用 Treap 树解决，因为区间是互不相交的，所以可以对区间进行排序，然后不断的加入的下一个区间的 dog，并处理请求。

其实这题最好用划分树解决。

POJ 2085 Inversion

构造法求序列，用二分树状数组或者 treap 树找第 k 大的数即可。

1.9 伸展树(Splay Tree)

大部分内容来自 Crash 的论文。

参考文献：

Crash《运用伸展树解决数列维护问题》

杨思雨《伸展树的基本操作与应用》

董的博客-伸展树：http://dongxicheng.org/structure/splay-tree/

编写：黄李龙　　校核：黄李龙

1.9.1 概述

二叉查找树（Binary Search Tree，也叫二叉排序树，即 Binary Sort Tree）能够支持多种动态集合操作，它可以用来表示有序集合、建立索引等，因而在实际应用中，二叉排序树是一种非常重要的数据结构。

从算法复杂度角度考虑，我们知道，作用于二叉查找树上的基本操作（如查找，插入等）的时间复杂度与树的高度成正比。对一个含 n 个节点的完全二叉树，这些操作的最坏情况运行时间为 O(log n)。但如果因为频繁的删除和插入操作，导致树退化成一个 n 个节点的线性链（此时即为一个单链表），则这些操作的最坏情况运行时间为 O(n)。为了克服以上缺点，很多二叉查找树的变形出现了，如红黑树、AVL 树，Treap 树等。

二叉查找树的一种改进数据结构－伸展树（Splay Tree）。它的主要特点是不会保证树一直是平衡的，但各种操作的平摊时间复杂度是 O(log n)，因而，从平摊复杂度上看，二叉查找树也是一种平衡二叉树。另外，相比于其他树状数据结构（如红黑树，AVL 树等），伸展树的空间要求与编程复杂度要小得多。

1.9.2 基本操作

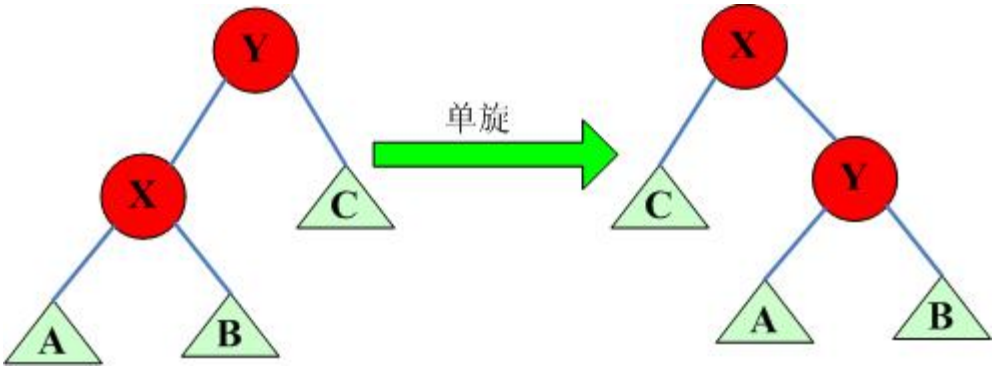
伸展树的出发点是这样的：考虑到局部性原理（刚被访问的内容下次可能仍会被访问，查找次数多的内容可能下一次会被访问），为了使整个查找时间更小，被查频率高的那些节点应当经常处于靠近树根的位置。这样，很容易得想到以下这个方案：每次查找节点之后对树进行重构，把被查找的节点搬移到树根，这种自调整形式的二叉查找树就是伸展树。每次对伸展树进行操作后，它均会通过旋转的方法把被访问节点旋转到树根的位置。

为了将当前被访问节点旋转到树根，我们通常将节点自底向上旋转，直至该节点成为树根为止。“旋转”的巧妙之处就是在不打乱数列中数据大小关系（指中序遍历结果是全序的）情况下，所有基本操作的平摊复杂度仍为 O（log n）。

伸展树主要有三种旋转操作，分别为单旋转，一字形旋转和之字形旋转。为了便于解释，我们假设当前被访问节点为 X，X 的父亲节点为 Y（如果 X 的父亲节点存在），X 的祖父节点为 Z（如果 X 的祖父节点存在）。

（1）单旋转

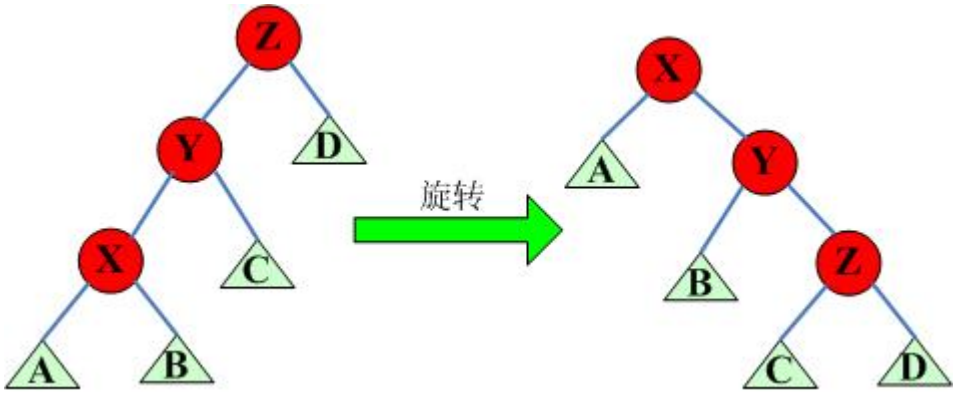
节点 X 的父节点 Y 是根节点。这时，如果 X 是 Y 的左孩子，我们进行一次右旋操作；如果 X 是 Y 的右孩子，则我们进行一次左旋操作。经过旋转，X 成为二叉查找树 T 的根节点，调整结束。



上图从左到右，表示的是右旋 X 节点操作，即将 X 节点顺时针旋转；从右到左，是左旋 Y 节点的操作，即将 Y 节点逆时针旋转。

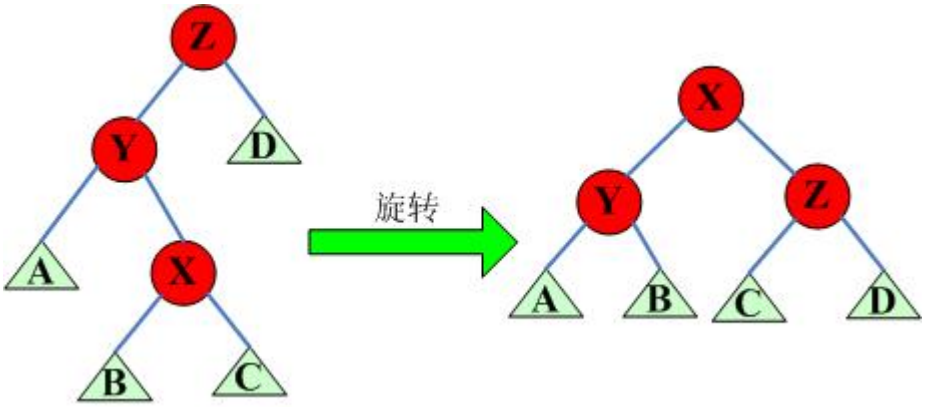
（2）一字型旋转

节点 X 的父节点 Y 不是根节点，Y 的父节点为 Z，且 X 与 Y 同时是各自父节点的左孩子或者同时是各自父节点的右孩子。这时，我们进行一次左左旋转操作或者右右旋转操作。我们先旋转 Y，再旋转 X。我们称这种旋转为一字形旋转。



（3）之字形旋转

节点 X 的父节点 Y 不是根节点，Y 的父节点为 Z，X 与 Y 中一个是其父节点的左孩子而另一个是其父节点的右孩子。这时，我们进行一次左右旋转操作或者右左旋转操作。



通常来说，每进行一种操作后都会进行一次伸展(Splay)操作，这样可以保证每次操作的平摊时间复杂度是$O(\log_2 N)$。关于证明可以参见相关书籍和论文，如《数据结构与算法分析—C语言描述》美·Mark Allen Weiss。

既然可以把任何一个结点转到根，那么也就可以把任意一个结点转到其到根路径上任何一个结点的下面（特别地，转到根就是转到空结点 Null 的下面）。下面的利用伸展树维护数列就要用到将一个结点转到某个结点下面。

最后附上 Splay 操作的代码：

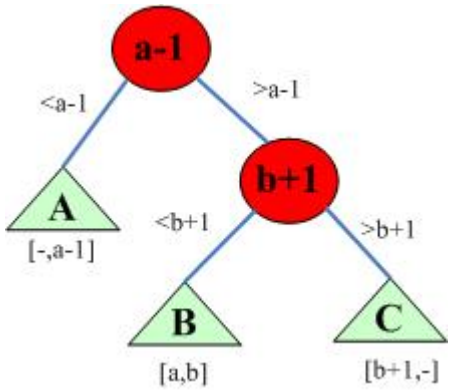
```
// node 为结点类型，其中 ch[0]表示左结点指针，ch[1]表示右结点指针
// pre 表示指向父亲的指针
void Rotate(node *x, int c) // 旋转操作，c=0 表示左旋，c=1 表示右旋
{
 node *y = x->pre;
 y->ch[! c] = x->ch[c];
 if (x->ch[c] != Null) x->ch[c]->pre = y;
 x->pre = y->pre;
 if (y->pre != Null)
 if (y->pre->ch[0] == y) y->pre->ch[0] = x; else y->pre->ch[1] = x;
 x->ch[c] = y, y->pre = x;
 if (y == root) root = x; // root 表示整棵树的根结点
}
void Splay(node *x, node *f) // Splay 操作，表示把结点 x 转到结点 f 的下面
{
```

```
        for ( ; x->pre != f; )
            if (x->pre->pre == f) // 父结点的父亲即为 f，执行单旋转
                if (x->pre->ch[0] == x) Rotate(x, 1); else Rotate(x, 0);
            else
            {
                node *y = x->pre, *z = y->pre;
                if (z->ch[0] == y)
                    if (y->ch[0] == x)
                        Rotate(y, 1), Rotate(x, 1); // 一字形旋转
                    else
                        Rotate(x, 0), Rotate(x, 1); // 之字形旋转
                else
                    if (y->ch[1] == x)
                        Rotate(y, 0), Rotate(x, 0); // 一字形旋转
                    else
                        Rotate(x, 1), Rotate(x, 0); // 之字形旋转
        }
    }
```

1.9.3 在伸展树中对区间进行操作

首先我们认为伸展树的中序遍历即为我们维护的数列，那么很重要的一个操作就是怎么在伸展树中表示任意一个区间。比如我们要提取区间[a,b]，那么我们将 a 前面一个数对应的结点转到树根，将 b 后面一个结点对应的结点转到树根的右边，那么根右边的左子树就对应了区间[a,b]。

其中的道理也是很简单的，将 a 前面一个数对应的结点转到树根后， a 及 a 后面的数就在根的右子树上，然后又将 b 后面一个结点对应的结点转到树根的右边，那么[a,b]这个区间就是下图中所示的 B 子树。



利用这个，我们就可以实现线段树的一些功能，比如回答对区间的询问。我们在每个结点上记录关于以这个结点为根的子树的信息，然后询问时先提取区间，再直接读取子树的相关信息。还可以对区间进行整体修改，这也要用到和线段树类似的延迟标记技术，就是对于每个结点，再额外记录一个或多个标记，表示以这个结点为根的子树是否被进行了某种操作，并且这种操作影响其子结点的信息值。当然，既然记录了标记，那么旋转和其他一些操作中也就要相应地将标记向下传递。

到目前为止，伸展树只是实现了线段树能够实现的功能，下面两个功能将是线段树无法办到的。

如果我们要在 a 后面插入一些数，那么我们先把这些插入的数建成一棵伸展树，我们可以利用分治法建立一棵完全平衡的二叉树，就是说每次把最中间的作为当前区间的根，然后左右递归处理，返回的时候进行维护。接着将 a 转到根，将 a 后面一个数对应的结点转到根结点的右边，最后将这棵新的子树挂到根右子结点的左子结点上。

还有一个操作就是删除一个区间[a,b]内的数，像上面一样，我们先提取区间，然后直接删除那棵子树，即可达到目的。最后还需注意的就是，每当进行一个对数列进行修改的

操作后，都要维护伸展树，一种方法就是对影响到的结点从下往上执行 Update 操作。但还有一种方法，就是将修改的结点旋转到根，因为 Splay 操作在旋转的同时也会维护每个结点的值，因此可以达到对整个伸展树维护的目的。最后还有一个小问题，因为数列中第一个数前面没有数字了，并且最后一个数后面也没有数字了，这样提取区间时就会出一些问题。为了不进行过多的特殊判断，我们在原数列最前面和最后面分别加上一个数，在伸展树中就体现为结点，这样提取区间的时候原来的第 k 个数就是现在的第 k ?1 个数。并且我们还要注意，这两个结点维护的信息不能影响到正确的结果。

下面看一下新的 Splay 操作的程序（能对结点信息进行维护）：

```
// node 为结点类型，其中 ch[0]表示左结点指针，ch[1]表示右结点指针
// pre 表示指向父亲的指针
void Rotate(node *x, int c) // 旋转操作，c=0 表示左旋，c=1 表示右旋
{
node *y = x->pre;
Push_Down(y), Push_Down(x);
// 先将 Y 结点的标记向下传递（因为 Y 在上面），再把 X 的标记向下传递
y->ch[! c] = x->ch[c];
if (x->ch[c] != Null) x->ch[c]->pre = y;
x->pre = y->pre;
if (y->pre != Null)
if (y->pre->ch[0] == y) y->pre->ch[0] = x; else y->pre->ch[1] = x;
x->ch[c] = y, y->pre = x, Update(y); // 维护 Y 结点
if (y == root) root = x; // root 表示整棵树的根结点
}
void Splay(node *x, node *f) // Splay 操作，表示把结点 x 转到结点 f 的下面
{
    for (Push_Down(x) ; x->pre != f; ) // 一开始就将 X 的标记下传
        if (x->pre->pre == f) // 父结点的父亲即为 f，执行单旋转
            if (x->pre->ch[0] == x) Rotate(x, 1); else Rotate(x, 0);
        else
        {
            node *y = x->pre, *z = y->pre;
            if (z->ch[0] == y)
                if (y->ch[0] == x)
                    Rotate(y, 1), Rotate(x, 1); // 一字形旋转
                else
                    Rotate(x, 0), Rotate(x, 1); // 之字形旋转
            else
                if (y->ch[1] == x)
                    Rotate(y, 0), Rotate(x, 0); // 一字形旋转
                else
                    Rotate(x, 1), Rotate(x, 0); // 之字形旋转
        }
    Update(x); // 最后再维护 X 结点
}
```

可能有人会问，为什么在旋转的时候只对 X 结点的父亲进行维护，而不对 X 结点进行维护，但是 Splay 操作的最后却又维护了 X 结点？原因很简单。因为除了一字形旋转，在 Splay 操作里我们进行的旋转都只对 X 结点进行，因此过早地维护是多余的；而在一字形旋转中，好像在旋转中没有对 X 的父亲进行维护，但后面紧接着就是旋转 X 结点，又会对 X 的父亲进行维护，也是没问题的。这样可以节省不少冗余的 Update 操作，能减小程序隐含的常数。最后我们看看怎么样实现把数列中第 k 个数对应的结点转到想要的位置。对于这个操作，我们要记录每个以结点为根子树的大小，即包含结点的个数，然

后从根开始，每次决定是向左走，还是向右走，具体见下面的代码：

```
// 找到处在中序遍历第 k 个结点，并将其旋转到结点 f 的下面
void Select(int k, node *f)
{
    int tmp;
    node *t;
    for (t = root; ; ) // 从根结点开始
    {
        Push_Down(t); // 由于要访问 t 的子结点，将标记下传
        tmp = t->ch[0]->size; // 得到 t 左子树的大小
        if (k == tmp + 1) break; // 得出 t 即为查找结点，退出循环
        if (k <= tmp) // 第 k 个结点在 t 左边，向左走
            t = t->ch[0];
        else // 否则在右边，而且在右子树中，这个结点不再是第 k 个
            k -= tmp + 1, t = t->ch[1];
    }
    Splay(t, f); // 执行旋转
}
```

1.9.4 实例分析—NOI 2005 维护数列（Sequence）

题目的意思很简单：维护一个数列，支持以下几种操作：

1. 插入：在当前数列第 posi 个数字后面插入 tot 个数字；若在数列首位插入，则 posi 为 0。
2. 删除：从当前数列第 posi 个数字开始连续删除 tot 个数字。
3. 修改：从当前数列第 posi 个数字开始连续 tot 个数字统一修改为 c 。
4. 翻转：取出从当前数列第 posi 个数字开始的 tot 个数字，翻转后放入原来的位置。
5. 求和：计算从当前数列第 posi 个数字开始连续 tot 个数字的和并输出。
6. 求和最大子序列：求出当前数列中和最大的一段子序列，并输出最大和。

首先，对于 1、2 两个操作，前面已经分析过了。而 3、4 两个操作为修改操作，因此我们只需增加两个标记：same 和 rev ，分别表示这棵子树是否被置为一

个数和是否被翻转。对于第 5 个操作，我们给每个结点维护一个 sum，即可解决问题。第 6 个操作的实现是最为复杂的，我们需要维护 4 个值：value 、MaxL 、MaxR 和 MaxM ，分别表示这个结点的值、这个子树表示数列左起最大连续和、右起最大连续和以及这个数列和最大的子序列。其中，MaxL 、MaxR 和 MaxM 的值可以由 value 和这个结点子树子树的相关信息得到。以 MaxL 为例，我们要考虑三种情况：只在左子树对应的序列中、全部左子树的序列和这个结点以及横跨左右子树（左子树全部包含）。MaxR 和 MaxM 的维护可以类比一下（注意 MaxM 有五种情况）。

然后由于要执行 Select 操作，因此每个结点还要维护一个 size 。最终得到每个结点要维护 8 个标记或信息：same 、rev 、value 、sum、size 、MaxL 、MaxR 、MaxM 。

因为我们要询问的是子序列和以及最大和子序列，因此上文中提及的额外增加的两个结点的 value 、MaxL 、MaxR 、MaxM 应为一个很小的数（比如-100000），

而 sum 应该为 0，这样才不会影响正常的询问结果。最后注意的是，每种修改操作（插入、删除、修改和翻转）过后，要对伸展树的信息重新维护。按照前面说的，我们将修改的结点（即根结点右子结点的左子结点） Splay 到根的位置。这样就能保证新的伸展树的值都是正确的。

还有一点就是删除操作要回收空间，不然会使用过多的空间（大概要 100MB）。并且

由于 C++的指针回收操作过慢，因此我们人工压一个栈回收结点指针。

附 C++代码，封装成结构体便于操作，代码结构也更良好。

（测试数据在网上搜索能找到）

```
/* NOI2005 维护数列
 * 可以做模板用
 * */
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 4000000 + 6;
struct SplayTree {
struct Node {
    Node *ch[2], *pre;
    int sz;
    int val, sum;
    bool rev, same;
    int mmax, lmax, rmax;
};
#define keyTree (root->ch[1]->ch[0])
Node *root, *null_node;
Node mem[MAXN];
Node *que[MAXN], *pool[MAXN];
int mem_cnt, top;

void push_up(Node *x) {
    if (x == null_node) return;
    x->sz = 1;
    x->sum = x->lmax = x->rmax = x->val;
    int mmax = x->val;
    if (x->ch[0] != null_node) {
        push_down(x->ch[0]); /* 有可能还没更新完成 */
        x->lmax = max(x->ch[0]->lmax, x->ch[0]->sum + x->lmax);
        x->rmax = max(x->val, x->ch[0]->rmax + x->val);
        mmax = max(mmax, max(x->ch[0]->mmax, max(x->lmax, max(x->rmax,
x->ch[0]->rmax+x->val))));
        x->sz += x->ch[0]->sz;
        x->sum += x->ch[0]->sum;
    }
    if (x->ch[1] != null_node) {
        push_down(x->ch[1]); /* 有可能还没更新完成 */
        x->lmax = max(x->lmax, x->sum + x->ch[1]->lmax);
        x->rmax = max(x->ch[1]->rmax, x->rmax + x->ch[1]->sum);
        mmax = max(mmax, max(x->ch[1]->mmax, max(x->lmax, x->rmax)));
        mmax = max(mmax, max(0, (x->ch[0]==null_node)?0:x->ch[0]-
>rmax)+x->val+x->ch[1]->lmax);
        x->sz += x->ch[1]->sz;
        x->sum += x->ch[1]->sum;
    }
    x->mmax = mmax;
}
void push_down(Node *x) {
    if (x == null_node) return;
    if (x->same) {
        x->same = 0;
        x->sum = x->val * x->sz;
```

```
            if (x->val > 0) {
                x->mmax = x->lmax = x->rmax = x->sum;
            } else {
                x->mmax = x->lmax = x->rmax = x->val;
            }
            x->ch[0]->same = x->ch[1]->same = 1;
            x->ch[0]->val = x->ch[1]->val = x->val;
        }
        if (x->rev) {
            x->rev = 0;
            swap(x->ch[0], x->ch[1]);
            swap(x->lmax, x->rmax);
            x->ch[0]->rev ^= 1;
            x->ch[1]->rev ^= 1;
        }
    }
    /** 旋转操作，c=0 表示左旋，逆时针，c=1 表示右旋，顺时针 */
    void rotate(Node *x, int c) {
        Node *y = x->pre;
        push_down(y);
        push_down(x);
        y->ch[!c] = x->ch[c];
        if (x->ch[c] != null_node) x->ch[c]->pre = y;
        x->pre = y->pre;
        if (y->pre != null_node) y->pre->ch[ y == y->pre->ch[1] ] = x;
        x->ch[c] = y;
        y->pre = x;
        push_up(y);
    }
    /** 伸展的操作，将 x 节点转到 f 下，f 要在 x 之上，f 为 null_node 时表示 x 转为根 */
    void splay(Node *x, Node *f) {
        push_down(x);
        while (x->pre != f) {
            if (x->pre->pre == f) {
                rotate(x, x->pre->ch[0] == x);
            } else {
                Node *y = x->pre, *z = y->pre;
                bool c = (z->ch[0] == y);
                if (y->ch[c] == x) {
                    rotate(x, !c);
                    rotate(x, c);
                } else {
                    rotate(y, c);
                    rotate(x, c);
                }
            }
        }
        push_up(x);
        if (f == null_node) root = x;
    }
    bool select(int k, Node *f) {
        Node *x = root;
        for (;;) {
            push_down(x); /* 需要先推下去，有可能这个区间会翻转，这样就不会弄错左右子
树的节点数了*/
            int sz = x->ch[0]->sz;
            if (sz + 1 == k) break;
            if (k <= sz) {
                x = x->ch[0];
```

```
        } else {
            k -= sz + 1;
            x = x->ch[1];
        }
        if (x == null_node) {
            return false;
        }
    }
    splay(x, f);
    return true;
}
int size() const {
    return root->sz - 2;
}
Node* new_node(int val) {
    Node *x;
    if (top >= 0) x = pool[top--];
    else x = mem + mem_cnt++;
    x->pre = x->ch[0] = x->ch[1] = null_node;
    x->sz = 1;
    x->val = x->sum = x->mmax = x->lmax = x->rmax = val;
    x->rev  = x->same = 0;
    return x;
}
void make_tree(Node *&x, Node *f, int a[], int tot) {
    if (tot <= 0) return;
    int m = (tot - 1) >> 1;
    x = new_node(a[m]);
    x->pre = f;
    make_tree(x->ch[0], x, a, m);
    make_tree(x->ch[1], x, a+m+1, tot-m-1);
    push_up(x);
}
void init(int n, int a[]) {
    null_node = mem;
    memset(null_node, 0, sizeof(Node));
    null_node->pre = null_node->ch[0] = null_node->ch[1] = null_node;
    mem_cnt = 1, top = -1;
    // 开头和末尾添加两个节点，避免特判，方便取区间，但是节点数变为 n+2
    root = new_node(-1);
    root->ch[1] = new_node(-1);
    root->ch[1]->pre = root;
    root->sz = 2; /* 可能要用 push_up */
    make_tree(keyTree, root->ch[1], a, n);
    push_up(root->ch[1]);
    push_up(root);
}
/* 将 [l,r] 区间转到 keyTree */
void ch_key_tree(int l, int r) {
    select(l, null_node);
    select(r+2, root);
}
void insert(int pos, int a[], int n) {
    Node *tmp_root = root, *tmp_root2;
    make_tree(root, null_node, a, n);
    tmp_root2 = root;
    root = tmp_root;
    ch_key_tree(pos+1, pos);
    keyTree = tmp_root2;
```

```
        keyTree->pre = root->ch[1]; /* 注意维护 pre */
        push_up(root->ch[1]);
        push_up(root);
    }
    void erase(Node * &x) {
        if (x == null_node) {
            return;
        }
        Node *f = x->pre, **fr = que, **ta = que;
        *ta++ = x;
        while (fr < ta) {
            x = *fr++;
            pool[++top] = x;
            if (x->ch[0] != null_node) *ta++ = x->ch[0];
            if (x->ch[1] != null_node) *ta++ = x->ch[1];
        }
        x = null_node;
        if (f != null_node) push_up(f);
    }
    void pre_ord(Node *x) { /* 中序遍历输出 */
        if (x == null_node) return;
        for (int i = 0; i < 2; ++i)
            if (x->ch[i] != null_node && x->ch[i]->pre != x) {
              printf("error pre %d(%p)<-%d(%p)\n", x->val, x, x->ch[i]->val,
x->ch[0]);
            }
        if (x->sz != 1 + x->ch[0]->sz + x->ch[1]->sz) {
            printf("error sz %d:%d  %p %d\n", x->sz, 1+x->ch[0]->sz+x->ch[1]-
>sz, x, x->val);
        }
        if (x->ch[0]) pre_ord(x->ch[0]);
        printf("%d ", x->val);
        if (x->ch[1]) pre_ord(x->ch[1]);
    }
    void travel() { /* 输出序列*/
        pre_ord(root);
        printf("\n");
    }
    void del(int pos, int tot) { /* 删除 pos 位置开始的 tot 个元素 */
        ch_key_tree(pos, pos+tot-1);
        erase(keyTree);
        push_up(root);
    }
    void make_same(int pos, int tot, int val) {
        ch_key_tree(pos, pos+tot-1);
        keyTree->same = 1;
        keyTree->val = val;
    }
    void reverse(int pos, int tot) {
        ch_key_tree(pos, pos+tot-1);
        keyTree->rev ^= 1;
    }
    int get_sum(int pos, int tot) {
        ch_key_tree(pos, pos+tot-1);
        push_down(keyTree);
        return keyTree->sum;
    }
    int max_sum() {
        ch_key_tree(1, size());
```

```
        push_down(keyTree);
        return keyTree->mmax;
    }
    };

    SplayTree spt;
    int a[MAXN];

    int main() {
    int n, m;
    while (EOF != scanf("%d%d", &n, &m)) {
        for (int i = 0; i < n; ++i) scanf("%d", &a[i]);
        spt.init(n, a);
        while (m--) {
            int pos, tot;
            char cmd[65];
            scanf("%s", cmd);
            // printf("%s\n", cmd);
            if (0 == strcmp(cmd, "INSERT")) {
                scanf("%d%d", &pos, &tot);
                for (int i = 0; i < tot; ++i) scanf("%d", &a[i]);
                spt.insert(pos, a, tot);
            } else if (0 == strcmp(cmd, "DELETE")) {
                scanf("%d%d", &pos, &tot);
                spt.del(pos, tot);
            } else if (0 == strcmp(cmd, "MAKE-SAME")) {
                int c;
                scanf("%d%d%d", &pos, &tot, &c);
                spt.make_same(pos, tot, c);
            } else if (0 == strcmp(cmd, "REVERSE")) {
                scanf("%d%d", &pos, &tot);
                spt.reverse(pos, tot);
            } else if (0 == strcmp(cmd, "GET-SUM")) {
                scanf("%d%d", &pos, &tot);
                printf("%d\n", spt.get_sum(pos, tot));
            } else if (0 == strcmp(cmd, "MAX-SUM")) {
                printf("%d\n", spt.max_sum());
            } else {
                printf("error cmd:%s\n", cmd);
            }
        }
    }
    return 0;
    }
```

1.9.5 和线段树的比较

用伸展树解决数列维护问题，可以支持两个线段树无法支持的操作：在某个位置插入一些数和删除一些连续的数。但是也带了更大的常数和更大的代码量。因此，在没有必要使用伸展树的时候，我们就不应该使用。不过有些问题看似线段树无法解决，然而对模型进行转化后，也能用线段树解决，所以做题的时候不要急着出算法，还是要分析一下问题的本质，选择最好最合适的方法解决。

1.9.6 伸展树例题

1.9.6.1 POJ 3468 A Simple Problem with Integers

题意：有N个整数的序列，值分别是$A_1, A_2, \ldots, A_N$，($1 \leqslant N \leqslant 100000$)。现在有Q个操作($1 \leqslant Q \leqslant 100000$)，操作有两种：

"C a b c" 表示给 Aa, Aa+1, ... , Ab 分别加上 c， -10000 ≤ c ≤ 10000。

"Q a b" 表示询问 Aa, Aa+1, ... , Ab 的和。

思路：

最好的方法是用线段树做，这里是为了练习伸展树而用伸展树解决此题。

每个节点再添加一个 sum 和 add，sum 表示以这个节点为根的树所有节点的和，add 表示这个子树所有节点加的值。对于 C a b c 的操作，先把[a,b]区间提取出来，然后在这个[a,b]子树的根节点的 add 加上 c 即可，在伸展的时候，再把这个节点的 add 值推向它的子节点；对于 Q a b 操作，先提取[a,b]区间，然后返回这个节点的 sum 值即可。可以直接套用 NOI 2005 维护数列一题的代码。

代码如下：

```
#include <cstdio>
#include <cstring>

typedef long long int64;
const int MAXN = 100005;
struct Node {
 Node *pre, *ch[2];
 int sz;
 int val, add;
 int64 sum;
};

struct SplayTree {
#define keyTree (root->ch[1]->ch[0])

 int cnt;
 Node mem[MAXN];
 Node *root, *null_leaf;

 void PushDown(Node *x) {
     if (x->add) {
         x->val += x->add;
         x->ch[0]->add += x->add;
         x->ch[1]->add += x->add;
         x->ch[0]->sum += (int64)(x->ch[0]->sz) * x->add;
         x->ch[1]->sum += (int64)(x->ch[1]->sz) * x->add;
         x->add = 0;
     }
 }

 void PushUp(Node *x) {
     if (x == null_leaf) puts("error null_leaf");
     x->sz = 1 + x->ch[0]->sz + x->ch[1]->sz;
     x->sum = x->add + x->val + x->ch[0]->sum + x->ch[1]->sum;
 }

 /* 旋转操作，c=0 表示左旋，逆时针，c=1 表示右旋，顺时针
  * */
 void Rotate(Node *x, int c) {
     Node *y = x->pre;
     PushDown(y);
     PushDown(x);
     y->ch[!c] = x->ch[c];
     if (x->ch[c] != NULL) x->ch[c]->pre = y;
     x->pre = y->pre;
     if (y->pre != NULL) y->pre->ch[ y->pre->ch[1] == y ] = x;
     x->ch[c] = y;
     y->pre = x;
     PushUp(y);
```

```
}

/* 将 x 转到 f 下，特别的 x->pre==NULL 表示的是根
 * */
void Splay(Node *x, Node *f) {
    PushDown(x);
    while (x->pre != f) {
        if (x->pre->pre == f) {
            Rotate(x, x->pre->ch[0] == x);
        } else {
            Node *y = x->pre, *z = y->pre;
            int c = (z->ch[0] == y);
            if (y->ch[c] == x) {
                Rotate(x, !c), Rotate(x, c); // 之字形旋转
            } else {
                Rotate(y, c), Rotate(x, c); // 一字形旋转
            }
        }
    }
    PushUp(x);
    //if (f == NULL) root = x;
    if (f == null_leaf) root = x;
}
/* 选择第 k 个节点，并转到 f 下
 * */
void Select(int k, Node *f) {
    Node *x = root;
    while (x->ch[0]->sz + 1 != k) {
        PushDown(x);
        int sz = x->ch[0]->sz;
        if (k <= sz) {
            x = x->ch[0];
        } else {
            k -= sz + 1;
            x = x->ch[1];
        }
    }
    Splay(x, f);
}

Node* newNode(int val) {
    Node *x = mem + cnt++;
    x->pre = x->ch[0] = x->ch[1] = null_leaf;
    //x->pre = NULL;
    x->sz = 1;
    x->add = 0;
    x->sum = x->val = val;
    return x;
}

void MakeTree(Node* &x, int l, int r, Node *f, int a[]) {
    if (l > r) return;
    int m = (l + r) >> 1;
    x = newNode(a[m]);
    x->pre = f;
    MakeTree(x->ch[0], l, m-1, x, a);
    MakeTree(x->ch[1], m+1, r, x, a);
    PushUp(x);
}
void init(int n, int a[]) {
    // null_leaf 是特殊的叶节点，表示无，避免特判,因为有些不能有个 NULL 节点
    null_leaf = mem;
    memset(null_leaf, 0, sizeof(*null_leaf));
    null_leaf->pre = null_leaf;
    cnt = 1;
    // 开头和末尾添加两个节点，避免特判，方便取区间，但是节点数变为 n+2
    root = newNode(-1);
    root->ch[1] = newNode(-1);
    root->ch[1]->pre = root;
```

```
        root->sz = 2;

        MakeTree(keyTree, 0, n-1, root->ch[1], a);
        PushUp(root->ch[1]);
        PushUp(root);
    }

    int64 Query(int a, int b) {
        Select(a, null_leaf);
        Select(b+2, root);
        return keyTree->sum;
    }

    void Add(int a, int b, int c) {
        Select(a, null_leaf);
        Select(b+2, root);
        keyTree->add += c;
        keyTree->sum += (int64) c * root->ch[1]->ch[0]->sz;
    }
   };

   SplayTree spt;
   int a[MAXN];

   int main() {
    int n, q;
    while (EOF != scanf("%d%d", &n, &q)) {
        for (int i = 0; i < n; ++i) scanf("%d", &a[i]);
        spt.init(n, a);
        while (q--) {
            char cmd[3];
            int a, b, c;
            scanf("%s%d%d", cmd, &a, &b);
            if (cmd[0] == 'Q') {
                printf("%lld\n", spt.Query(a, b));
            } else {
                scanf("%d", &c);
                spt.Add(a, b, c);
            }
        }
    }
    return 0;
    }
```

1.9.6.2 HDU 4441 Queue Sequence

Problem Description

There's a queue obeying the first in first out rule. Each time you can either push a number into the queue (+i), or pop a number out from the queue (-i). After a series of operation, you get a sequence (e.g. +1 -1 +2 +4 -2 -4). We call this sequence a queue sequence.

Now you are given a queue sequence and asked to perform several operations:

1. insert p

First you should find the smallest positive number (e.g. i) that does not appear in the current queue sequence, then you are asked to insert the +i at position p (position starts from 0). For -i, insert it into the right most position that result in a valid queue sequence (i.e. when encountered with element -x, the front of the queue should be exactly x).

For example, (+1 -1 +3 +4 -3 -4) would become (+1 +2 -1 +3 +4 -2 -3 -4) after operation 'insert 1'.

2. remove i

Remove +i and -i from the sequence.

For example, (+1 +2 -1 +3 +4 -2 -3 -4) would become (+1 +2 -1 +4 -2 -4) after operation 'remove 3'.

3. query i

Output the sum of elements between +i and -i. For example, the result of query 1, query 2, query 4 in sequence (+1 +2 -1 +4 -2 -4) is 2, 3(obtained by -1 + 4), -2 correspond.

Input

There are less than 25 test cases. Each case begins with a number indicating the number of operations n (1 ≤ n ≤ 100000). The following n lines with be 'insert p', 'remove i' or 'query i'(0 ≤ p ≤ length (current sequence), 1 ≤ i, i is granted to be in the sequence).

In each case, the sequence is empty initially.

The input is terminated by EOF.

Output

Before each case, print a line "Case #d:" indicating the id of the test case.

After each operation, output the sum of elements between +i and -i.

Sample Input

```
10
insert 0
insert 1
query 1
query 2
insert 2
query 2
remove 1
remove 2
insert 2
query 3
6
insert 0
insert 0
remove 2
query 1
insert 1
query 2
```

Sample Output

```
Case #1:
2
-1
2
0
Case #2:
0
-1
```

Source

2012 Asia Tianjin Regional Contest

思路：题目中的 remove 和 query 涉及区间的动态删除和询问，可以想到伸展树，也可以用其他的平衡树做，比如 SBT(Size Balance Tree)，这里用伸展树,直接用了 NOI2005 维护数列的模板。

下面分析各个操作的实现。

1.insert p

在 p 位置插入一个没在序列中的最小正整数，因为 n<=100000，可以用一个树状数组去找了，二分找到最小的正整数 a，具体应用看代码中的实现。

然后插入 a 到 p 位置，还要插入-a，因为是队列，所以如果 a 之前有 x 个正整数，那么我们的-a 就应该放在开始的 x 个负整数之后，要求-a 尽量往右，那么我们找到第 x+1 个负整数的位置 p_，然后插入到这个位置 p_，因为可能没有第 x+1 负整数，所以注意特判。

2.remove i

把 i 和-i 移除。我用一个数组记录 i 和-i 对应的节点指针，删除时，先把节点用 splay 操作旋转到根，然后就能知道节点所在的序列位置，删除即可。

3. query i

求 i 和-i 之间的数的和。跟 remove 一样，找到两数的位置后用 splay 操作提取区间求和即可。

```
#include <cstdio>
#include <cstring>
#include <cassert>
#include <algorithm>
using namespace std;

typedef long long int64;

const int MAXN = 200000 + 5;

struct TreeArray {
 int c[MAXN];
#define lowbit(x) ((x)&(-x))
 void add(int i, int v) {
     for (; i < MAXN; i += lowbit(i)) {
         c[i] += v;
     }
 }
 int sum(int i) {
     int s = 0;
     for (; i > 0; i -= lowbit(i)) {
         s += c[i];
     }
     return s;
 }
};

struct SplayTree {
 struct Node {
     Node *ch[2], *pre;
     int sz, psz, nsz;
     int val;
     int64 sum;
 };
#define keyTree (root->ch[1]->ch[0])
 Node *root, *null_node;
 Node mem[MAXN];
 Node *que[MAXN], *pool[MAXN];
 int mem_cnt, top;
 Node *vp[MAXN]; // 记录每个值对应的节点，所有数都加上 offset 偏移，避免负数
 int offset;

 void push_up(Node *x) {
     if (x == null_node) return;
     x->sz = 1;
     if (x->val > 0) {
         x->psz = 1, x->nsz = 0;
```

```
        } else if (x->val < 0) {
            x->psz = 0, x->nsz = 1;
        } else {
            x->psz = x->nsz = 0;
        }
        x->sum = x->val;
        for (int c = 0; c < 2; ++c) {
            if (x->ch[c] != null_node) {
                x->sz += x->ch[c]->sz;
                x->psz += x->ch[c]->psz;
                x->nsz += x->ch[c]->nsz;
                x->sum += x->ch[c]->sum;
            }
        }
    }
    /** 旋转操作，c=0 表示左旋，逆时针，c=1 表示右旋，顺时针 */
    void rotate(Node *x, int c) {
        Node *y = x->pre;
        y->ch[!c] = x->ch[c];
        if (x->ch[c] != null_node) x->ch[c]->pre = y;
        x->pre = y->pre;
        if (y->pre != null_node) y->pre->ch[ y == y->pre->ch[1] ] = x;
        x->ch[c] = y;
        y->pre = x;
        push_up(y);
    }
    /** 伸展的操作，将 x 节点转到 f 下，f 要在 x 之上，f 为 null_node 时表示 x 转为根 */
    void splay(Node *x, Node *f) {
        while (x->pre != f) {
            if (x->pre->pre == f) {
                rotate(x, x->pre->ch[0] == x);
            } else {
                Node *y = x->pre, *z = y->pre;
                bool c = (z->ch[0] == y);
                if (y->ch[c] == x) {
                    rotate(x, !c);
                    rotate(x, c);
                } else {
                    rotate(y, c);
                    rotate(x, c);
                }
            }
        }
        push_up(x);
        push_up(f);
        if (f == null_node) root = x;
    }
    bool select(int k, Node *f) {
        Node *x = root;
        for (;;) {
            int sz = x->ch[0]->sz;
            if (sz + 1 == k) break;
            if (k <= sz) {
                x = x->ch[0];
            } else {
                k -= sz + 1;
                x = x->ch[1];
            }
            if (x == null_node) {
                return false;
            }
        }
        splay(x, f);
        return true;
    }
    int size() const {
        return root->sz - 2;
    }
    Node* new_node(int val) {
```

```
        Node *x;
        if (top >= 0) x = pool[top--];
        else x = mem + mem_cnt++;
        memset(x, 0, sizeof(Node));
        x->pre = x->ch[0] = x->ch[1] = null_node;
        vp[val+offset] = x;
        x->sz = 1;
        if (val > 0) x->psz = 1;
        else if (val < 0) x->nsz = 1;
        x->val = x->sum = val;
        return x;
    }
    void init(int n) {
        memset(vp, 0, sizeof(vp));
        offset = n;
        null_node = mem;
        memset(null_node, 0, sizeof(Node));
        null_node->pre = null_node->ch[0] = null_node->ch[1] = null_node;
        mem_cnt = 1, top = -1;

        // 开头和末尾添加两个节点，避免特判，方便取区间，但是节点数变为 n+2
        root = new_node(0);
        root->ch[1] = new_node(-2*n-1);
        root->ch[1]->pre = root;
        push_up(root);

    }
    /* 将 实际 [l,r] 区间转到 keyTree */
    void ch_key_tree(int l, int r) {
        select(l, null_node);
        select(r+2, root);
    }
    void erase(Node * &x) {
        if (x == null_node) {
            return;
        }
        Node *f = x->pre, **fr = que, **ta = que;
        *ta++ = x;
        while (fr < ta) {
            Node *x = *fr++;
            vp[x->val + offset] = NULL;
            pool[++top] = x;
            if (x->ch[0] != null_node) *ta++ = x->ch[0];
            if (x->ch[1] != null_node) *ta++ = x->ch[1];
        }
        x = null_node;
        push_up(f);
    }
    void del(int pos, int tot) {
        ch_key_tree(pos, pos+tot-1);
        erase(keyTree);
        push_up(root);
    }
    int64 get_sum(int l, int r) {
        ch_key_tree(l, r);
        return keyTree->sum;
    }
    void pre_ord(Node *x) {
        if (x == null_node) return;
        for (int i = 0; i < 2; ++i)
            if (x->ch[i] != null_node && x->ch[i]->pre != x) {
                fprintf(stderr, "error pre %d(%p)<-%d(%p)\n", x->val, x, x->ch[i]->val,
x->ch[0]);
                exit(-1);
            }
        if (x->sz != 1 + x->ch[0]->sz + x->ch[1]->sz) {
            fprintf(stderr, "error sz %d:%d  %p %d\n", x->sz, 1+x->ch[0]->sz+x->ch[1]-
>sz, x, x->val);
            exit(-1);
```

```
        }
        if (x->ch[0]) pre_ord(x->ch[0]);
        printf("%d ", x->val);
        //printf("%d(%p)  ", x->val, x);
        if (x->ch[1]) pre_ord(x->ch[1]);
    }
    void travel() {
        printf("travel() ");
        pre_ord(root);
        printf("\n");
    }

    /******************** Problem Operation **********/
    int get_sz(int t, int v) {
        //printf("get_sz %d %d\n", v, v + offset);
        if (vp[v+offset] == NULL) {
            fprintf(stderr, "vp %d %d\n", v, v + offset);
            //while (1);
            assert(vp[v+offset]);
        }
        splay(vp[v+offset], null_node);
        if (t == 0) return root->ch[0]->sz;
        else if (t > 0) return root->ch[0]->psz + 1;
        else return root->ch[0]->nsz + 1;
    }
    int get_size(int t, int l, int r) {
        ch_key_tree(l, r);
        if (t == 0) return keyTree->sz;
        else if (t > 0) return keyTree->psz;
        else return keyTree->nsz;
    }
    void ins(int p, int v) {
        //printf("ins %d %d\n", p, v);
        ch_key_tree(p, p-1);
        keyTree = new_node(v);
        keyTree->pre = root->ch[1];
        push_up(root->ch[1]);
        push_up(root);
    }
    int loc_ne(int no) {
        int l = 1, r = size();
        while (l < r) {
            int m = (l + r) >> 1;
            int s = get_size(-1, 1, m);
            if (s < no) {
                l = m + 1;
            } else {
                r = m;
            }
        }
        if (get_size(-1, 1, l) < no) ++l;
        return l;
    }
    void rm(int v) {
        //if (vp[v+offset] == NULL) return;
        int l = get_sz(0, v);
        del(l, 1);
    }
    int64 query(int i) {
        if (vp[i+offset] == NULL || vp[-i+offset] == NULL) return -1;
        int l = get_sz(0, i), r = get_sz(0, -i);
        //printf("query %d %d\n", l, r);
        return get_sum(l, r);
    }
};

SplayTree spt;
TreeArray ta;
```

```
int get_minp() {
 int l = 1, r = MAXN - 1;
 while (l < r) {
     int m = (l + r) >> 1;
     if (ta.sum(m) >= 1) {
         r = m;
     } else {
         l = m + 1;
     }
 }
 return l;
}

int main() {
 int n, ncase = 0;
 while (EOF != scanf("%d", &n)) {
     ++ncase;
     printf("Case #%d:\n", ncase);
     memset(ta.c, 0, sizeof(ta.c));
     for (int i = 1; i <= n; ++i) ta.add(i, 1);
     spt.init(n);

     for (int i = 0; i < n; ++i) {
         char cmd[32];
         int d;
         scanf("%s%d", cmd, &d);
         //  fprintf(stderr, "%d:%s %d\n", i, cmd, d);
         if ('i' == cmd[0]) { // insert p
             ++d;
             int a = get_minp();
             ta.add(a, -1);
             spt.ins(d, a); // insert +a
             int as = spt.get_sz(1, a);
             int p_ = spt.loc_ne(as);
             //printf("loc %d => %d\n", as, p_);
             spt.ins(p_, -a);
         } else if ('r' == cmd[0]) { // remove d
             spt.rm(-d);
             spt.rm(d);
             ta.add(d, 1);
         } else if ('q' == cmd[0]) { // query d
             printf("%lld\n", spt.query(d));
         } else {
         }
         //      spt.travel(); puts("");
     }
 }
 return 0;
 }
```

其他例题：

HDU 4286 Data Handler 模拟题，可以直接使用双线链表进行模拟，但是需要一些处理技巧，也可以用伸展树模拟。

第2章 动态规划

2.1 递推

编写：周洲　　校核：黄李龙

2.1.1 递推原理

递推的概念和基本思想：给定一个数的序列 H0,H1,…,Hn,…若存在整数 n0，使当 n>n0 时,可以用等号(或大于号、小于号)将 Hn 与其前面的某些项 Hi(0<i<n)联系起来，这样的式子就叫做递推关系。

递推定义：递推算法是一种简单的算法，即通过已知条件，利用特定关系得出中间推论，直至得到结果的算法。递推算法分为顺推和逆推两种。

顺推法：从已知条件出发，逐步推算出要解决的问题的方法叫顺推。　　如斐波拉契数列，设它的函数为 f(n),已知 f(1)=1，f(2)=1;f(n)=f(n-2)+f(n-1)(n>=3,n∈N)。则我们通过顺推可以知道,f(3)=f(1)+f(2)=2,f(4)=f(2)+f(3)=3……直至我们要求的解。

逆推法：从已知问题的结果出发，用迭代表达式逐步推算出问题的开始的条件，即顺推法的逆过程，称为逆推。

2.1.2 一般的思路

首先，确认：能否容易的得到简单情况的解？

然后，假设：规模为 N-1 的情况已经得到解决。

最后，重点分析：当规模扩大到 N 时，如何枚举出所有的情况，并且要确保对于每一种子情况都能用已经得到的数据解决。

2.1.3 经典题目

2.1.3.1 HDU 1465 不容易系列之一

1．题目出处/来源：HDU 1465 不容易系列之一

2．题目描述：

某人写了 n 封信和 n 个信封，如果所有的信都装错了信封。求所有的信都装错信封,共有多少种不同情况。

3．分析:

当 N=1 和 2 时，易得解，假设 F(N-1)和 F(N-2)已经得到，重点分析下面的情况：

当有 N 封信的时候，前面 N-1 封信可以有 N-1 或者 N-2 封错装

前者，对于每种错装，可从 N-1 封信中任意取一封和第 N 封错装，故=F(N-1)*(N-1)

后者简单，只能是没装错的那封和第 N 封交换信封，没装错的那封可以是前面 N-1 封中的任意一个，故= F(N-2) * (N-1)

由此可得，f(n)=(n-1)*(f(n-1)+f(n-2))

4．代码:

```
#include<stdio.h>
int main()
{
   __int64 a[21]={0,0,1,2};
   int i,n;
   for(i=4;i<=20;i++)
```

```
        a[i]=(i-1)*(a[i-1]+a[i-2]);
    while(scanf("%d",&n)!=EOF)
        printf("%I64d\n",a[n]);
    return 0;
}
```

2.1.3.2 HDU 2045 不容易系列之(3)——LELE 的 RPG 难题

1．题目出处/来源：HDU 2045 不容易系列之(3)——LELE 的 RPG 难题

2．题目描述：

有排成一行的 n 个方格，用红(Red)、粉(Pink)、绿(Green)三色涂每个格子，每格涂一色，要求任何相邻的方格不能同色，且首尾两格也不同色．求全部的满足要求的涂法.

3．分析:

数组 F[i]保存 i 个方格有多少种填涂方法。

n 个方格可以由 n-1 个方格和 n-2 个方格填充得到。

比如，在一涂好的 n-1 个格子里最后再插入一个格子，就得到了 n 个格子了。

因为已经填好 n-1 的格子中，每两个格子的颜色都不相同。

所以只能插入一种颜色。而 n-1 个格子一共有 F[n-1]种填涂方法。所以从 n-1 格扩充到 n 格共有 F(n-1)种方法。

若前 n-1 不合法，而添加一个后变成合法，即前 n-2 个合法，而第 n-1 个与第 1 个相同。

这时候有两种填法。

所以

f[n] = f[n-1] + 2 * f[n-2];

f[1] = 3;

f[2] = 6;

f[3] = 6

4．代码:

```
#include <math.h>
#include <stdio.h>
int main()
{
    int i;
    __int64 d[51] = {0, 3, 6, 6};

    for (i = 4; i < 51; i++)
        d[i] = d[i-1] + 2*d[i-2];
    while (scanf("%d", &i) != EOF)
        printf("%I64d\n", d[i]);

    return 0;
}
```

2.2 背包问题

参考文献：

《背包九讲》

编写：周洲　　校核：黄李龙

2.2.1 背包的入门和进阶

一、01 背包问题

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即 f[i][v]表示前 i 件物品恰放入一个容量为 v 的背包可以获得的最大价值。则其状态转移方程便是：

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 i-1 件物品的问题。如果不放第 i 件物品，那么问题就转化为“前 i-1 件物 品放入容量为 v 的背包中”，价值为 f[i-1][v]；如果放第 i 件物品，那么问题就转化为“前 i-1 件物品放入剩下的容量为 v-c[i]的背包中”，此时能获得的最大价值就是 f[i-1][v-c[i]]再加上通过放入第 i 件物品获得的价值 w[i]。

优化空间复杂度

以上方法的时间和空间复杂度均为 O(VN)，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到 O。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环 i=1..N，每次算出来二维数组 f[i][0..V]的所有值。那么，如果只用一个数组 f[0..V]，能不能保证第 i 次循环结束后 f[v]中表示的就是我们定义的状态 f[i][v]呢？f[i][v]是由 f[i-1][v]和 f[i-1] [v-c[i]]两个子问题递推而来，能否保证在推 f[i][v]时（也即在第 i 次主循环中推 f[v]时）能够得到 f[i-1][v]和 f[i-1][v-c[i]]的值呢？事实上，这要求在每次主循环中我们以 v=V..0 的顺序推 f[v]，这样才能保证推 f[v]时 f[v-c[i]]保存的是状态 f[i-1][v-c[i]]的值。伪代码如下：

```
for i=1..N
    for v=V..0
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

其中的 f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程 f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因为现在的 f[v-c[i]]就相当于原来的 f[i-1][v-c[i]]。如果将 v 的循环顺序从上面的逆序改成顺序的话，那么则成了 f[i][v]由 f[i][v-c[i]]推知，与本题意不符，但它却是另一个重要的背包问题 P02 最简捷的解决方案，故学习只用一维数组解 01 背包问题是十分必要的。

事实上，使用一维数组解 01 背包的程序在后面会被多次用到，所以这里抽象出一个处理一件 01 背包中的物品过程，以后的代码中直接调用不加说明。

过程 ZeroOnePack，表示处理一件 01 背包中的物品，两个参数 cost、weight 分别表明这件物品的费用和价值。

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成 v=V..0 是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为 cost 的物品不会影响状态 f[0..cost-1]，这是显然的。

有了这个过程以后，01 背包问题的伪代码就可以这样写：

```
for i=1..N
    ZeroOnePack(c[i],w[i]);
```

初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 f[0]为 0 其它 f[1..V]均设为-∞，这样就可以保证最终得到的 f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 f[0..V]全部设为 0。

为什么呢？可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为 0 的背包可能被价值为 0 的 nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是-∞了。如果背包并非必须被装满，那么 任何容量的背包都有一个合法解“什么都不装”，这个解的价值为 0，所以初始时状态的值也就全部为 0 了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

一个常数优化

前面的伪代码中有 for v=V..1，可以将这个循环的下限进行改进。

由于只需要最后 f[v]的值，倒推前一个物品，其实只要知道 f[v-w[n]]即可。以此类推，对以第 j 个背包，其实只需要知道到 f[v-sum{w[j..n]}]即可，即代码中的

```
for i=1..N
    for v=V..0
```

可以改成

```
for i=1..n
    bound=max{V-sum{w[i..n]},c[i]}
    for v=V..bound
```

这对于 V 比较大时是有用的。

二、完全背包问题

有 N 种物品和一个容量为 V 的背包，每种物品都有无限件可用。第 i 种物品的费用是

c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这个问题非常类似于 01 背包问题，所 不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取 0 件、取 1 件、取 2 件……等很多种。如果仍然按照解 01 背包时的思路，令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k*c[i]<=v}

这跟 01 背包问题一样有 O(VN)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态 f[i][v]的时间是 O(v/c[i])，总的复杂度可以认为是 O(V*Σ(V/c[i]))，是比较大的。

将 01 背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明 01 背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i、j 满足 c[i]<=c[j]且 w[i]& gt;=w[j]，则将物品 j 去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得 j 换成物美价廉的 i，得到至少不会更差的方案。对于 随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的 O(N^2)地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于 V 的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以 O(V+N)地完成这个优化。这个不太重要的过程就不给出伪代码了，希望你能独立思考写 出伪代码或程序。

转化为 01 背包问题求解

既然 01 背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为 01 背包问题来解。最简单的想法是，考虑到第 i 种物品最多选 V/c[i]件，于是可以把第 i 种物品转化为 V/c[i]件费用及价值均不变的物品，然后求解这个 01 背包问题。这样完全没有改进基本 思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为 01 背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第 i 种物品拆成费用为 c[i]*2^k、价值为 w[i]*2^k 的若干件物品，其中 k 满足 c[i]*2^k<=V。这是二进制的思想，因为不管最优策略选几件第 i 种物品，总可以表示成若干个 2^k 件物品的和。这样把每种物品拆成 O(log V/c[i])件物品，是一个很大的改进。

但我们有更优的 O(VN)的算法。

O(VN)的算法

这个算法使用一维数组，先看伪代码：

```
for i=1..N
    for v=0..V
        f[v]=max{f[v],f[v-cost]+weight}
```

你会发现，这个伪代码与 P01 的伪代码只有 v 的循环次序不同而已。 为什么这样一改就可行呢？首先想想为什么 P01 中要按照 v=V..0 的逆序来循环。这是因为要保证第 i 次循环中的状态 f[i][v]是由状态 f[i-1] [v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第 i 件物品”这件策略时，依据的是一个绝无已经选入第 i 件物品的 子结果 f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第 i 种物品”这种策略时，却正需要一个可能已选入第 i 种物品的子结果 f[i][v-c[i]]，所以就可以并且必须采用 v=0..V 的顺序循环。这就是这个简单的程序为何成立的道理。

值得一提的是，上面的伪代码中两层 for 循环的次序可以颠倒。这个结论有可能会带来算法时间常数上的优化。

这个算法也可以以另外的思路得出。例如，将基本思路中求解 f[i][v-c[i]]的状态转移方程显式地写出来，代入原方程中，会发现该方程可以等价地变形成这种形式：

f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码：

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

二、多重背包问题

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 n[i]件可用，每件费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第 i 种物品有 n[i]+1 种策略：取 0 件，取 1 件……取 n[i]件。令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值，则有状态转移方程：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k<=n[i]}

复杂度是 O(V*Σn[i])。

转化为 01 背包问题

另一种好想好写的基本方法是转化为 01 背包求解：把第 i 种物品换成 n[i]件 01 背包中的物品，则得到了物品数为Σn[i]的 01 背包问题，直接求解，复杂度仍然是 O(V*Σn[i])。

但是我们期望将它转化为 01 背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第 i 种物品换成若干件物品，使得原问题中第 i 种物品可取的每种策略——取 0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过 n[i]件的策略必不能出现。

方法是：将第 i 种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为 1,2,4,...,2^(k-1),n[i]-2^k+1，且 k 是满足 n[i]-2^k+1>0 的最大整数。例如，如果 n[i]为 13，就将这种 物品分成系数分别为 1,2,4,6 的四件物品。

分成的这几件物品的系数和为 n[i]，表明不可能取多于 n[i]件的第 i 种物品。另外这种方法也能保证对于 0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分 0..2^k-1 和 2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第 i 种物品分成了 O(log n[i])种物品，将原问题转化为了复杂度为<math>O(V*Σlog n[i])的 01 背包问题，是很大的改进。

下面给出 O(log amount)时间处理一件多重背包中物品的过程，其中 amount 表示物品的数量：

```
procedure MultiplePack(cost,weight,amount)
    if cost*amount>=V
        CompletePack(cost,weight)
        return
    integer k=1
    while k<amount
        ZeroOnePack(k*cost,k*weight)
        amount=amount-k
        k=k*2
    ZeroOnePack(amount*cost,amount*weight)
```

希望你仔细体会这个伪代码，如果不太理解的话，不妨翻译成程序代码以后，单步执行几次，或者头脑加纸笔模拟一下，也许就会慢慢理解了。

2.2.2 经典题目

2.2.2.1 HDU 2602 Bone Collector

1．题目出处/来源: HDU 2602 Bone Collector

2．题目描述：知道了 N 块石头的体积和价值，有一个最多能装体积为 V 的石头的袋子，从 N 块石头里找出一些石头放在袋子里使得总价值最大，求出最大价值。

3．分析：符合 01 背包的定义，直接套用模版即可。

4．代码：

```
#include <stdio.h>
#include <string.h>

struct A
{
 int val;
```

```
 int v;
}E[1011];

int max(int a,int b)
{
 return a>b?a:b;
}
int main()
{
 int T;
 int dp[1011];
 int i,l;
 int N,V;

 scanf("%d",&T);
 while(T--)
 {
     scanf("%d%d",&N,&V);
     for(i=1;i<=N;i++)
         scanf("%d",&E[i].val);
     for(i=1;i<=N;i++)
         scanf("%d",&E[i].v);

     memset(dp,0,sizeof(dp));
     for(i=1;i<=N;i++)
     for(l=V;l>=E[i].v;l--)
         dp[l]=max(dp[l],dp[l-E[i].v]+E[i].val);

     printf("%d\n",dp[V]);
 }
 return 0;
}
```

2.2.2.2 HLG 1053 Warcraft III

1．题目出处/来源：HLG 1053 Warcraft III

2．题目描述：有 N 种作战的单位，每个单位都可以无限生产，但是生产都有一定的花费，每种单位都有自己的作战价值，现在知道了总的金钱数，问做多可以生产出的单位作战价值之和是多少。

3．分析：从完全背包的定义能够看出该题属于完全背包的题型，直接套用模版即可。

4．代码：

```
#include<stdio.h>
#include<string.h>
#define max(a,b)(a)>(b)?(a):(b)

int a[100000];
int b[100000];
int f[100000];
int main()
{
   int i,m,v,k,j,t;
   scanf("%d",&t);
   while(t--)
   {
       scanf("%d%d",&v,&m);
       memset(f,0,sizeof(f));

       for(i=0;i<m;i++)
           scanf("%d%d",&a[i],&b[i]);

       for(i=0;i<m;i++)
           for(j=b[i];j<=v;j++)
               f[j]=max(f[j],f[j-b[i]]+a[i]);

       printf("%d\n",f[v]);
```

```
    }
    return 0;
}
```

2.2.2.3 HD 2191 悼念 512

1．题目出处/来源：HD 2191 悼念 512

2．题目描述：

为了挽救灾区同胞的生命，心系灾区同胞的你准备自己采购一些粮食支援灾区，现在假设你一共有资金 n 元，而市场有 m 种大米，每种大米都是袋装产品，其价格不等，并且只能整袋购买。

请问：你用有限的资金最多能采购多少公斤粮食呢？

3．分析：多重背包，注意每种背包对应的情况。

4．代码：

```
#include<stdio.h>
int main()
{
    int f[200], p[200], d[200],s[200];
    int t, n, q, v;
    int i, j, k;
    scanf("%d",&t);
    while (t--)
    {
        scanf("%d %d",&v,&n);
        for (i=0; i<n; i++)
            scanf("%d %d %d",&d[i],&p[i],&s[i]);

        for (i=0; i<200; i++)
            f[i] = 0;

        for (i=0;i<n;i++)
        {
            if (d[i]*s[i]>=v)  // 使用完全背包
            {
                for(j=d[i];j<=v;j++)
                {
                    if(f[j]<f[j-d[i]]+p[i])
                        f[j]=f[j-d[i]]+p[i];
                    else f[j]=f[j];
                }
                continue;
            }
            k=1;
            while (k<s[i])    // 01 背包的变形，注意优化
            {
                for (j=v; j-k*d[i]>=0; j--)
                {
                    if (f[j]<f[j-k*d[i]]+k*p[i])
                        f[j]=f[j-k*d[i]]+k*p[i];
                    else f[j]=f[j];
                }
                s[i]=s[i]-k;
                k=k*2;
            }

            for (j=v;j-s[i]*d[i]>=0;j--)  //01 背包
            {
                if  (f[j]<f[j-s[i]*d[i]]+s[i]*p[i])
                    f[j]=f[j-s[i]*d[i]]+s[i]*p[i];
                else f[j]=f[j];
            }
```

```
            }
            printf("%d\n",f[v]);
        }
        return 0;
    }
```

2.3 区间动态规划

参考文献：

刘汝佳、黄亮《算法艺术与信息学竞赛》

郑州市第九中学 张旭祥《深入分析区间型动态规划》

编写：黄李龙　　校核：黄李龙

2.3.1 引子

区间动态规划不是一种方法，而是一种分类。某些动态规划的方程的状态是以**区间**作为状态，并在区间之间进行状态转移，获得最优解，所以区间动态规划需要以具体的题目去理解这种类别的动态规划题，需要自己多做题，总结这类题的特点，特别的要注意题目中最优子结构的证明，明白为什么这么做是正确。

下面以

NOIp2000 乘积最大

POJ 1141 Brackets Sequence

NOIp2006 能量项链

POJ 1191 棋盘分割

作为例题对区间动态规划做介绍。

2.3.2 NOIp2000 乘积最大

题目链接：Hrbustoj 1212

参考解答：http://blog.wledu.org/user1/yingqingjun/archives/2010/51274.html

题意：

设有一个长度 N 的数字串，要求选手使用 K 个乘号将它分成 K+1 个部分，找出一种分法，使得这 K+1 个部分的乘积能够为最大。

同时，为了帮助选手能够正确理解题意，主持人还举了如下的一个例子：

有一个数字串: 312，当 N=3，K=1 时会有以下两种分法：

1）3*12=36

2）31*2=62

这时，符合题目要求的结果是： 31*2=62

现在，请你设计一个程序，求得正确的答案。

输入

有多组测试数据，对于每组测试数据：

第一行共有 2 个自然数 N, K (6<=n<=40，1<=k<=6, K < N)

第二行是一个长度为 N 的数字串。

对于每组测试数据，输出一行，为所求得的最大乘积（一个自然数）。

样例:

输入
4 2
1231

输出
62

分析：

设字符串长度为 n，乘号数为 k，如果 n=50，k=1 时，有(n-1)=49 种不同的乘法，当 k=2 时，有 C(2，50-1)=1176 种乘法，即 C(k,n-1)种乘法，当 n、k 更大的时候，用穷举的方法就不行了。

设数字字符串为 a1a2…an

K=1 时：一个乘号可以插在 a1a2…an 中的 n-1 个位置，这样就得到 n-1 个子串的乘积：

a1*a2…an, a1a2*a3…an, …, a1a2…a n-1*an （ 这相当于是穷举的方法）

此时，最大值= max{a1*a2…an, a1a2*a3…an, …, a1a2…a n-1*an }

K=2 时，二个乘号可以插在 a1a2…an 中 n-1 个位置的任两个地方， 把这些乘积分个类，便于观察规律：

①最后一个数作为被乘数：

a1a2 ••• *a n-1*an , a1a2 ••• *a n-2 a n-1*an , a1*a2 ••• a n-3 a n-2 a n-1*an

设符号 F[n-1][1]为在前 n-1 个数中插入一个乘号的最大值,则的最大值为 F[n-1][1]*an

②最后 2 个数作为被乘数：

a1a2 ••• *a n-2*a n-1an , a1a2 …*a n-3 a n-2* an-1 an , a1*a2 ••• an-3 a n-2* an-1an

设符号 F[n-2][1]为在前 n-2 个数中插入一个乘号的最大值,则的最大值为 F[n-2][1]*an-1an

③最后 3 个数作为被乘数：

设符号 F[n-3][1]为在前 n-3 个数中插入一个乘号的最大值,则的最大值为 F[n-3][1]*an-2an-1an

••••••

a3~an 作为被乘数：F[2][1]*a3…an-2an-1an

此时的最大乘积为：

F[n][k]=max{F[n-1][1]*an，F[n-2][1]*an-1an，F[n-3][1]*an-2an-1an，•••，F[2][1]*a3…an-2an-1an}

设 F[i][j]表示在 i 个数中插入 j 个乘号的最大值，val[i][j]表示从 ai 到 aj 的子串作为一个数字的值，则可得到动规方程：

F[i][j] = max{F[i-1][j-1]*val[i][i], F[i-2][j-1]*val[i-1][i], F[i-3][j-1]*val[i-2][i], ••• , F[j][j-1]*val[j+1][i} (1<=i<=n, 1<=i<=k)

边界： F[i][0] = val[1][i] （数列本身）

阶段：子问题是在子串中插入 j-1,j-2……1,0 个乘号，因此乘号个数作为阶段的划分（j 个阶段）

状态：每个阶段随着被乘数数列的变化划分状态。

决策：在每个阶段的每种状态中做出决策。

注意问题：

（1）输入的字符需要进行数值转换。

（2）由于乘积可能很大，C\C++需要使用大整数乘法，也可以使用 Java 的 BigInteger

大整数类。

Java 代码：

```
import java.math.*;
import java.util.*;
import java.io.*;

public class Main {

 Scanner cin = new Scanner(new BufferedInputStream(System.in));
 BigInteger f[][] = new BigInteger[41][7];
 BigInteger val[][] = new BigInteger[41][41];

 public void solve() {
     while(cin.hasNext()) {
         int n, k;
         String s;
         n = cin.nextInt();
         k = cin.nextInt();
         s = cin.next();

         for(int i = 0; i < n; i++) {
             val[i][i] = BigInteger.valueOf(s.charAt(i)-'0');
             for(int j = i+1; j < n; j++) {
                 val[i][j] = val[i][j-
1].multiply(BigInteger.valueOf(10)).add(BigInteger.valueOf(s.charAt(j)-'0'));
             }
         }
         for(int i = 0; i < n; ++i) {
             f[i][0] = val[0][i];
             for(int j = 1; j <= i && j <= k; ++j) {
                 f[i][j] = BigInteger.ZERO;
                 for(int l = j-1; l < i; ++l) {
                     f[i][j] = f[i][j].max(f[l][j-1].multiply(val[l+1][i]));
                 }
             }
         }
         System.out.println(f[n-1][k]);
     }

 }
 public static void main(String[] args) {
     Main test = new Main();
     test.solve();
 }
 }
```

2.3.3 POJ 1141 Brackets Sequence

题意：我们定义一个合法的括号序列：

1.空序列是一个合法的括号序列。

2.如果 S 是一个合法的括号序列，那么(S)活着[S]也是一个合法的括号序列。

3.如果 A 和 B 是分别是合法的括号序列，那么 AB 也是一个合法的括号序列。

现在给出一个仅包含'(', ')', '[', 和 ']'字符的括号序列，要你添加尽量少的括号，使得整个序列成为合法的括号序列。

输入有多组测试数据，每组测试数据一行，为一个仅包含'(', ')', '[', 和 ']'字符的括号序列。

对于每组测试数据，输出添加括号后最短的合法括号序列。如果有多个答案，输出任意一个即可。

样例：

输入

```
([(]
```

输出

```
()[()]
```

分析：可以采用递归的方法来解决问题。设序列 SiSi+1 •••• Sj 最少需要添加 dp[i,j]个括号，根据不同情况，可以采用不同的方式转化成子问题。

- S 形如(S`)或者[S`]，此时只要把 S`变成合法序列，S 就是合法序列。
- S 形如(S`，先把 S`变成合法的序列，再在最后添加一个)即可。
- S 形如[S`、S`)和 S`]，和上一种情况一样。
- 只要序列的长度大于 1，都可以把 S 分成两部分 Si ••• Sk，Sk+1 ••• Sj，分别变成合法的规则序列后连在一起就变成了合法序列。

请思考一下，为什么这里的最优子结构能够成立，并获得最优解？

如果直接采用递归的方式解决，将会有很多重复的计算，可以用一个数组把这些已经计算过的子问题保存下来，这种方法叫做记忆化。也可以改成非递归的，此时就要先把子问题求出来，那么就是最小的问题一直推到大的问题，下面的代码是非递归形式实现的。题目还要求把答案输出，还要记录 dp[i,j]在获得最优值时的决策，我们用 pos[i,j]表示。如果 pos[i,j]=-1，则表示 Si 和 Sj 是一堆匹配的括号，如果不是则 pos[i,j]记录的是从哪个位置把 Si ••• Sj 分成两部分，输出答案的时候采用递归输出。最简单直接的做法是直接记录 dp[i,j]获得最优值时形成的序列，当然这种方法空间和时间花费更多。

代码：

```
#include <stdio.h>
#include <string.h>

const int MAXN = 256;
char br[MAXN];
int dp[MAXN][MAXN], pos[MAXN][MAXN];
int len_br;

void print_br(int i, int j)
{
    if (i > j)
        return;
    if (i == j){
        if (br[i]=='(' || br[i]==')')
            printf("()");
        else
            printf("[]");
    }
    else if (pos[i][j] == -1){
        printf("%c", br[i]);
        print_br(i+1, j-1);
        printf("%c", br[j]);
    }
    else {
        print_br(i, pos[i][j]);
        print_br(pos[i][j]+1, j);
    }
}
int main()
{
    int i, j, k, mid, t;

    while (NULL != gets(br)){

        len_br = strlen(br);
```

```
        memset(dp, 0, sizeof(dp));
        for (i = 0; i < len_br; i++)
            dp[i][i] = 1;
        for (k = 1; k < len_br; k++){
            for (i = 0; i + k < len_br; i++){
                j = i + k;
                dp[i][j] = 0x7fffffff;
                if (('('==br[i]&&br[j]==')') || ('['==br[i]&&br[j]==']')){
                        dp[i][j] = dp[i+1][j-1], pos[i][j] = -1;
                }
                for (mid = i; mid < j; mid++){
                    if (dp[i][j] > (t=dp[i][mid]+dp[mid+1][j])){
                        dp[i][j] = t, pos[i][j] = mid;
                    }
                }
            }
        }
        print_br(0, len_br-1);
        printf("\n");
    }
    return 0;
}
```

2.3.4 NOIp2006 能量项链

题目链接：Hrbustoj 1376

题目描述：

在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有 N 颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为 m，尾标记为 r，后一颗能量珠的头标记为 r，尾标记为 n，则聚合后释放的能量为 m*r*n（Mars 单位），新产生的珠子的头标记为 m，尾标记为 n。

需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

例如：设 N=4，4 颗珠子的头标记与尾标记依次为(2，3) (3，5) (5，10) (10，2)。我们用记号 ⊕ 表示两颗珠子的聚合操作，(j ⊕ k)表示第 j，k 两颗珠子聚合后所释放的能量。则第 4、1 两颗珠子聚合后释放的能量为：

(4 ⊕ 1)=10*2*3=60。

这一串项链可以得到最优值的一个聚合顺序所释放的总能量为

((4 ⊕ 1) ⊕ 2) ⊕ 3）=10*2*3+10*3*5+10*5*10=710。

输入

有多组测试数据。

对于每组测试数据，输入的第一行是一个正整数 N（4≤N≤100），表示项链上珠子的个数。第二行是 N 个用空格隔开的正整数，所有的数均不超过 1000。第 i 个数为第 i 颗珠子的头标记（1≤i≤N），当 i<N 时，第 i 颗珠子的尾标记应该等于第 i+1 颗珠子的头标记。第 N 颗珠子的尾标记应该等于第 1 颗珠子的头标记。

至于珠子的顺序，你可以这样确定：将项链放到桌面上，不要出现交叉，随意指定第一颗珠子，然后按顺时针方向确定其他珠子的顺序。

处理到文件结束。

输出

对于每组测试数据，输出只有一行，是一个正整数 E（E≤2.1*109），为一个最优聚合顺序所释放的总能量。

样例

输入：

4

2 3 5 10

输出：

710

分析：记 a[i]为第 i 颗能量珠的首标记，b[i]为第 i 颗能量珠的尾标记。记录 f[i,j]为从以 a[i]为首标记的能量珠开始顺时针数到以 b[j]为尾标记的能量珠为止所有能量珠组成的串合并后放出的最大能量。那么对于 f[i,j]，若先合并从以 a[i]为首标记的能量珠开始顺时针数到以 b[k]为尾标记的能量珠为止的串，再合并以 a[k]为首标记的能量珠开始顺时针数到以 b[j]为尾标记的能量珠为止的串，然后将这两颗合并后的能量珠合并，放出的能量为 f[i,k]+f[k,j]+a[i]*b[k]*b[j]。也就是说，状态转移方程为 f[i,j] = Max{f[i,k]+f[k,j]+a[i] *b[k]*b[j]} (i<=k<=j 或者 j<=k<=i)。

计算 f[i,j]时，只会用到含有能量珠数目比它少的串在数组中的值，所以可以以串的能量珠数目 x 为顺序规划，即按照从 i 到(i+l) mod n=j 的顺序进行规划。

对于 b[i]，有 b[i] = a[i+1]，所以可以只记录每颗能量珠的首标记即可。

C++代码：

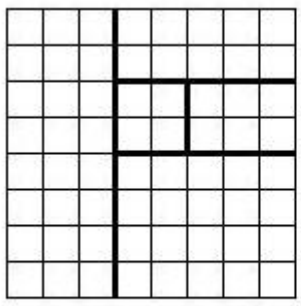
```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
const int NN = 105;
int f[NN][NN];
int a[NN];
int main()
{
 int n;
 while (EOF != scanf("%d", &n)) {
     for (int i = 0; i < n; ++i)     scanf("%d", &a[i]);
     memset(f, 0, sizeof(f));
     for (int l = 1; l < n; ++l) {
         for (int i = 0; i < n; ++i) {
             int j = (i + l) % n;
             for (int k = i; k != j; k = (k + 1) % n) {
                 f[i][j] = max(f[i][j], f[i][k] + f[(k + 1) % n][j] + a[i] *
a[(k+1)%n] * a[(j+1)%n]);
             }
         }
     }
     int ans = 0;
     for (int i = 0; i < n; ++i) {
         if (ans < f[i][(i-1 + n) % n]) {
             ans = f[i][(i-1 + n) % n];
         }
     }
     printf("%d\n", ans);
 }
 return 0;
}
```

2.3.5 NOI 2001 棋盘分割

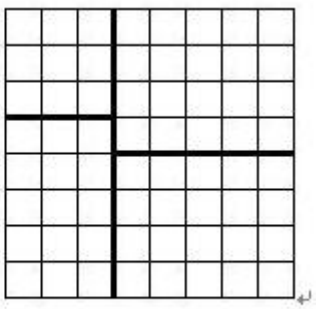
题目链接：POJ 1191 棋盘分割 http://poj.org/problem?id=1191

题目描述：

将一个 8*8 的棋盘进行如下分割：将原棋盘割下一块矩形棋盘并使剩下部分也是矩形，再将剩下的部分继续如此分割，这样割了(n-1)次后，连同最后剩下的矩形棋盘共有 n 块矩形棋盘。(每次切割都只能沿着棋盘格子的边进行)



允许的分割方案



不允许的分割方案

原棋盘上每一格有一个分值，一块矩形棋盘的总分为其所含各格分值之和。现在需要把棋盘按上述规则分割成 n 块矩形棋盘，并使各矩形棋盘总分的均方差最小。

均方差 $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$，其中平均值 $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$，xi 为第 i 块矩形棋盘的总分。

请编程对给出的棋盘及 n，求出 O'的最小值。

输入

第 1 行为一个整数 n(1 < n < 15)。

第 2 行至第 9 行每行为 8 个小于 100 的非负整数，表示棋盘上相应格子的分值。每行相邻两数之间用一个空格分隔。

输出

仅一个数，为 O'（四舍五入精确到小数点后三位）。

样例：

输入

```
3
1 1 1 1 1 1 1 3
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0
1 1 1 1 1 1 0 3
```

输出

1.633

题目来源
Noi 99

分析：参考《算法艺术与信息学竞赛》第 116 页。
均方差的公式比较复杂，先将其变形为

$$\sigma^2 = \frac{1}{n}(n(\bar{x})^2 + \sum_{i=1}^{n} x_i^{\,2} - 2\bar{x}\sum_{i=1}^{n} x_i) = \frac{1}{n}\sum_{i=1}^{n} x_i^{\,2} - (\bar{x})^2$$

由于平均值是一定的（它等于所有方格里的数的和除以 n），所以只需要让每个矩形的总分的平方和尽量小。考虑左上角坐标为(x1, y1)，右下角坐标为(x2,y2)的棋盘，设它的总和为 s[x1, y1, x2, y2]切割 k 次以后得到的 k+1 块矩形的总分的平方和最小值为 d[k, x1, y1, x2, y2]，则它可以沿着横线切，也可以沿着竖线切（这里用到了递归!）。故状态转移方程为：

d[k, x1, y1, x2, y2] = min {
min{d[k-1,x1,y1,a,y2]+s[a+1,y1,x2,y2],d[k-1,a+1,y1,x2,y2]+s[x1,y1,a,y2]}(x1<=a<x2)，
min{d[k-1,x1,y1,x2,b]+s[x1,b+1,x2,y2],d[k-1,x1,b+1,x2,y2]+s[x1,y1,x2,b]}(y1<=b<y2)，
}

设m为棋盘的边长，则状态数目为m^4n，决策数目为O(m)。预处理先用$O(m^2)$时间算出左上角为(1,1)的所有矩阵元素和，这样状态转移时间就是O(1)，故总的时间复杂度为$O(m^5n)$。由于m=8，n<=15，这个方法还是够快的。

C 代码：

```
#include <stdio.h>
#include <string.h>
#include <math.h>

int n;
long  dp[20][8][8][8][8];
long   s[8][8][8][8];
int map[8][8];


long l_min(long a, long b){return a<b?a:b;}

long l_sqr(long x){return x*x;}
double d_sqr(double x){return x*x;}

void init_status()
{
   memset(s, 0, sizeof(s));
   long sl, sl2;
   int x1, y1, x2, y2;

   for (x2 = 0; x2 < 8; x2++){
      sl = 0;
      for (y2 = 0; y2 < 8; y2++){
         sl += map[x2][y2];
         sl2 = 0;
         for (y1 = 0; y1 <= y2; y1++){
            for (x1 = 0; x1 <= x2; x1++){
               s[x1][y1][x2][y2] = sl - sl2;
```

```
                if (x2 > 0)
                    s[x1][y1][x2][y2] += s[x1][y1][x2-1][y2];
            }
            sl2 += map[x2][y1];
        }
    }
}
for (x1 = 0; x1 < 8; x1++){
    for (y1 = 0; y1 < 8; y1++){
        for (x2 = x1; x2 < 8; x2++){
            for (y2 = y1; y2 < 8; y2++){
                s[x1][y1][x2][y2] = l_sqr(s[x1][y1][x2][y2]);
                dp[0][x1][y1][x2][y2] = s[x1][y1][x2][y2];
            }
        }
    }
}

}

long solve()
{
    int x1, x2, y1, y2, k, a;
    long *p;

    for (k = 1; k < n; k++){
        for (x1 = 0; x1 < 8; x1++){
            for (y1 = 0; y1 < 8; y1++){
                for (x2 = x1; x2 < 8; x2++){
                    for (y2 = y1; y2 < 8; y2++){
                        p = &dp[k][x1][y1][x2][y2];
                        *p = 9999999;
                        for (a = x1; a < x2; a++){
                            *p = l_min(*p, dp[k-1][x1][y1][a][y2]+s[a+1][y1][x2][y2]);
                            *p = l_min(*p, dp[k-1][a+1][y1][x2][y2]+s[x1][y1][a][y2]);
                        }
                        for (a = y1; a < y2; a++){
                            *p = l_min(*p, dp[k-1][x1][y1][x2][a]+s[x1][a+1][x2][y2]);
                            *p = l_min(*p, dp[k-1][x1][a+1][x2][y2]+s[x1][y1][x2][a]);
                        }
                    }
                }
            }
        }
    }
    return dp[n-1][0][0][7][7];
}
int main()
{
    int i, j, x_;
    double _n;
    long ret;

    scanf("%d", &n);
    x_ = 0;
    for (i = 0; i < 8; i++){
        for (j = 0; j < 8; j++){
            scanf("%d", &map[i][j]);
            x_ += map[i][j];
        }
    }

    init_status();

    ret = solve();
    _n = 1.0 / n;

    printf("%.3f\n", sqrt(_n*ret - d_sqr((_n*x_))));
    return 0;
```

```
}
```

2.3.6 其他题目

NUAA 1086 最小代价字母树 [OIBH 模拟赛]
http://acm.nuaa.edu.cn/acmhome/problemdetail.do?method=showdetail&id=1086
NUAA 1007 加分二叉树[NOIp2003]
http://acm.nuaa.edu.cn/acmhome/problemdetail.do?method=showdetail&id=1007
HDU 4283 You Are the One
POJ 2176 Folding

更多题目可以用搜索引擎搜索 “区间 DP”。

2.4 状态压缩动态规划

编写：曹振海　　校核：黄李龙

2.4.1 状态压缩的原理

状态压缩 DP，其实所有的状态压缩 DP 基本都可以转换成普通的 DP，但是用状态压缩的目的，是为了减小内存消耗，因为利用状态压缩之后，所有的状态都可以用一个 32 位甚至是 16 位的整数（以 16 位居多，因为 32 位状态太多）表示出来，也可以利用位运算极快的运算速度来提高程序的速度，所以需要对几种位运算有一定的了解：

‘|’（按位或运算符），和逻辑中的或类似，按位或运算满足只有两个二进制位都为 0 时结果才为 0，其他情况皆为 1，举例来说 0000 0001|0000 0010 结果即为 0000 0011，按位或运算在状态 DP 中，常作为把一个状态集合加到另一个状态集合的方法，并且保证不会重复（因为重复得到的还是 1）。

‘&’（按位与运算符），同样和逻辑中的与运算类似，按位与运算满足两个二进制位都为 1 时结果为 1，其他情况皆为 0，0000 1010&0000 1101 结果为 0000 1000，按位与运算在状态 DP 中常作为判断状态集合中是否含有某种状态集合。

‘^’（按位异或运算符），按位异或运算符的运算规则是只有两个二进制位相反时结果为 1，相同则为 0，如 0000 1010^0000 1101 结果为 0000 0111,，‘^’运算在状态 DP 中常作为去掉某种状态集合的运算符，可先利用&判断是否含有这种状态，然后可以利用^运算从状态集合中去掉这种状态（这种方法在记忆化搜索中基本上是必须用到的，要熟练掌握）。

‘~’（按位取反运算）这是位运算中唯一一个单目运算符，这个运算符在状态 DP 中用的并不多，但是也要掌握，~是对相应的操作数按照二进制位按位取反，~0000 0011 结果为 1111 1100。

‘<<’（按位左移运算），是把一个数的二进制往左移位，相当于是这个数乘 2，对于有符号数来说，正数低位用 0 补齐，负数用 1 补齐，所以在用的时候要注意有可能会造成越界的问题，0000 0101<<2 结果为 0001 0100‘<<’运算符在状态 DP 中经常要和上面几种运算符结合起来使用实现状态的转换。

‘>>’(按位右移运算)，和左移相反，右移意味着这个数除 2，这个在二分里面用的比较多，但是在状态 DP 里的用处不多，0001 0100>>3 结果为 0000 0010。

上面是状态压缩 DP 里面要用到的一些基础的位运算的知识。状态压缩 DP 还有一个很重要的特点就是数据范围，一般情况下，状态压缩的数据范围都在 20 以内，要保证总状态数最多在百万的数量级的范围内才能保证 1 秒内能得出答案，一般会给出 16 以内的数据范围，这个时候就可以考虑利用状态压缩 DP 或者是状态枚举来解决了。有的时候题

目给出的数据范围并不一定满足条件，但是可以通过一定的转化把数据范围缩小到这样的范围，那么也是可以考虑利用状态 DP 的。

另外，状态中的每一个二进制位的 0 或者是 1 可以表示的意义有很多种，最直接的一种就是取不取，或者表示是不是全取。

和最基本的动态规划一样，状态 DP 一样可以分为两种，自顶向下的记忆化搜索和自下向上的递推式求解。在自顶向下的记忆化搜索中，经常要利用'^'运算去掉一些状态得到比较小的状态集合，而在自下向上的递推式求解中，又要经常利用'|'运算逐步加上一些状态以达到最终的状态。

2.4.2 一般的解题思路

当题中出现数据范围为 20 以内时，就应该尝试去利用状态 DP 解决了，或者可以通过一定的方式把数据范围减小到这个范围以内，也可以考虑状态 DP，想要用状态压缩 DP 解决一道题时，首先要搞明白，每个状态应该表示的意义是什么（或者说每一位的 1 或者 0 表示的意义分别是什么），然后根据表示的意义推导出状态转移方程，最后是确定如何用位运算进行优化。

2.4.3 经典题目

2.4.3.1 POJ1185-炮兵阵地

1．题目出处/来源

POJ1185-炮兵阵地

2．题目描述

司令部的将军们打算在 N*M 的网格地图上部署他们的炮兵部队。一个 N*M 的地图由 N 行 M 列组成，地图的每一格可能是山地（用"H" 表示），也可能是平原（用"P"表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

3．分析

这道题是最经典的一道状态 DP 的题，其中 N<=100,M<=10，从 M 的数据范围中可以看出这道题可以用状态 DP 来解决，我们分别先对每一行进行处理，每一行的状态范围为 0~(1<<M)-1，从这些状态中先得到一些在行上是合法的状态（即不会互相攻击），另外每一行能放的合法状态由其上面两行的状态共同决定，所以我们确定 f[i][j][k]表示的意义为

第 i-1 行的状态为 k 第 i 行的状态为 j 时所能得到的最优解，在不会与上两行的状态矛盾的情况下 f[i][j][k]=max(f[i][j][k],f[i-1][k][l]+sum[j]),其中 sum[j]表示 j 状态所能放的大炮的数量。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<stdio.h>
#include<string.h>
int st[61];//最多不会超过 61 种合法的状态
char a[110][15];//原图
int sum[61];//每个状态对应的大炮的数量
int surface[101];//记录每一行的山地状态
int f[101][61][61];
int cnt;//合法的状态的个数
int r,c;
int max(int a,int b)
{
    return a>b?a:b;
}
bool can(int x)//判断状态是否合法
{
    if(x&(x<<1))
    return false;
    if(x&(x<<2))
    return false;
    return true;
}
int getsum(int x)//求状态中 1 的数量，即为状态所表示的大炮的数量
{
    int num=0;
    while(x)
    {
        if(x&1)
        num++;
        x/=2;
    }
    return num;
}
void getst(int n)//对于单行来说，得到合法的状态
{
    int i;
    for(i=0;i<(1<<n);i++)
    {
        if(can(i))
        {
            st[cnt]=i;
            sum[cnt++]=getsum(i);
        }
    }
}
int main()
{
    int i,j,k;
    while(scanf("%d%d",&r,&c)!=EOF)
    {
        cnt=0;
        for(i=0;i<r;i++)
        scanf("%s",a[i]);
```

```
        getst(c);
        memset(surface,0,sizeof(surface));
        memset(f,0,sizeof(f));
        for(i=0;i<r;i++)//得到每一行的不能出现大炮的位置所表示的状态
        for(j=0;j<c;j++)
        {
            if(a[i][j]=='H')
            surface[i]+=1<<j;//这里+也可以改为|
        }
        for(i=0;i<cnt;i++)//预处理第一行
        {
            if(surface[0]&st[i])
            continue;
            f[0][i][0]=sum[i];
        }
        for(i=0;i<cnt;i++)//预处理第二行
        {
            if(surface[1]&st[i])
            continue;
            for(k=0;k<cnt;k++)
            {
                if(surface[0]&st[k])
                continue;
                if(st[k]&st[i])
                continue;//保证状态 i 不与第二行山地矛盾，不能与对应的第 1 行的状态矛盾
                f[1][i][k]=max(f[1][i][k],f[0][k][0]+sum[i]);
            }
        }
        for(i=2;i<r;i++)//求解后面的状态
        {
            for(j=0;j<cnt;j++)
            {
                if(surface[i]&st[j])
                continue;
                for(k=0;k<cnt;k++)
                {
                    if(surface[i-1]&st[k])
                    continue;
                    if(st[k]&st[j])
                    continue;
                    for(int l=0;l<cnt;l++)
                    {
                        if(st[l]&surface[i-2])
                        continue;
                        if((st[l]&st[k])||(st[l]&st[j]))
                        continue;//保证状态 j 不与上两行的状态矛盾，不与本行的山地矛盾
                        f[i][j][k]=max(f[i][j][k],f[i-1][k][l]+sum[j]);
                    }
                }
            }
        }
        int max=0;
        for(i=0;i<cnt;i++)//求解最优值
        for(j=0;j<cnt;j++)
        if(f[r-1][i][j]>max)
        max=f[r-1][i][j];
        printf("%d\n",max);
    }
    return 0;
```

```
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

2.4.3.2 Math Magic

1．题目出处/来源

ZOJ3662-2012 年第 37 届 ACM-ICPC 亚洲区预选赛长春站 C 题-Math Magic

2．题目描述

给出 K 个数的和 N 以及这 K 个数的最小公倍数 M，$1 \le N, M \le 1{,}000, 1 \le K \le 100$,要求出这 K 个数共有多少种不同的组合。

3．分析

从这道题所给出的数据范围我们是无法用状态 DP 来做的，但是我们看到有 K 个数的最小公倍数为 M，我们知道，K 个数的最小公倍数为 M，暨代表着在这 K 个数种每一个质因子所对应的最高的次数和 M 的质因子分解对应的次数是一样的，所以我们可以先对 M 质因子分解，1000 以内的数最多有四个不同的质因子（因为最小的五个质数 2*3*5*7*11 已经超过了 1000），这样就把这道题的数据范围缩小到了可以承受的范围之内，记录 M 的每一个质因子对应的次数，枚举 M 所有的因子，只有当因子的质因子分解的某一个质因子也达到了这个次数，才把这个因子所对应的状态中表示这一个质因子的二进制位置为 1.然后，DP[i][j][k]表示前 i 个数，状态为 i，和为 k 所对应的所有情况的数量。因为这道题中每个状态有可能多次出现，所以最好采用递推式的方法用'|'运算从下至上的运算，采用记忆化搜索自顶向下运算时，因为'^'运算本身的性质，会使得处理多次出现变的很麻烦。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define SUM 1005
#define STATE 20
#define N 105
#define MOD 1000000007
using namespace std;
int dp[N][STATE][SUM];
bool isprime[SUM];
int prime[200],cnt;
int dn,res[4];
int g[STATE][100],num[STATE];//g 存储的是每一种状态所对应的因子都有哪些，num 存储每种状态对应的因子的数量
void init()//打素数表
{
 int i,j;
 memset(isprime,true,sizeof(isprime));
 cnt=0;
 for(i=2;i<=1000;i++)
 {
     if(isprime[i])
     {
         prime[cnt++]=i;
         for(j=i*i;j<=1000;j+=i)
             isprime[j]=false;
     }
 }
}
int main()
{
 init();
 int n,m,k,i,j,temp,tn,pn;
 while(scanf("%d%d%d",&m,&n,&pn)!=EOF)//这里 m 表示和，n 表示最小公倍数，pn 表示数的个数
```

```
    {
        tn=n;
        dn=0;
        memset(dp,0,sizeof(dp));
        for(i=0;i<cnt;i++)//对 n 进行质因子分解
        {
            if(n%prime[i]==0)
            {
                temp=n;
                while(n%prime[i]==0)
                {
                    n/=prime[i];
                }
                res[dn]=temp/n;//res 存储的是每一个质因子对应的次数所表示因子
                dn++;
                if(n==1)
                    break;
            }
        }
        int sta;
        memset(num,0,sizeof(num));
        for(i=1;i<=tn;i++)//枚举 tn 的每一个因子，并确定所对应的状态
        {
            if(tn%i==0)
            {
                sta=0;
                for(j=0;j<dn;j++)
                {
                    if(i%res[j]==0)
                        sta|=(1<<j);//i 是 tn 的因子并且 i 达到了某一个质因子的最大的次数
                }
                g[sta][num[sta]++]=i;
            }
        }
        dp[0][0][0]=1;
        for(i=0;i<pn;i++)
            for(j=0;j<(1<<dn);j++)
                for(k=0;k<m;k++)
                {
                    if(dp[i][j][k])//当前面状态存在时，才进行下面的处理
                    {
                        for(int l=0;l<(1<<dn);l++)//这两层循环相当于是枚举每一个因子
                            for(int t=0;t<num[l];t++)
                            {
                                if(i+1<=pn&&k+g[l][t]<=m)
                                {
                                    dp[i+1][j|l][k+g[l][t]]+=dp[i][j][k];
                                    dp[i+1][j|l][k+g[l][t]]%=MOD;
                                }
                            }
                    }
                }
        printf("%d\n",dp[pn][(1<<dn)-1][m]);
    }
    return 0;
    }
```

2.4.3.3 HLG1473-教主的遗产

1．题目出处/来源

HLG1473-教主的遗产

2．题目描述

恭送教主！

教主在 2012 年 7 月 19 日上午 10:48，坐上前往北京的火车，从此开始了高富帅的生活。

在教主的的大学四年 ACM 生涯中，他用事实告诉我们，要想在比赛拿奖，除了平时的刻苦努力外，很大一部分还要依赖比赛时是

做题策略，简单来讲就是做题顺序，唯有想把能过的都过掉，然后再过难题，这样才能在比赛中拿奖。

我们将用这个做题顺序量化表示，即 AC 系数，AC 系数越大，拿奖可能性就越大。

在比赛中会给出 n 个题，不同做题顺序会有不同的 AC 系数，假如 A 先做，B 后做的话，B 对 A 的 AC 系数为 4， 反过来 B 先做，A 后做的话，A 对 B 的 AC 系数为 5，说明先做 B 后做 A 将得到更高的 AC 系数。

设 Sij 表示第 i 题在第 j 题做完后才做获得的 AC 系数，有三道题 a, b, c，做题顺序为 bac，则系数和为：Sum = Sab + Scb + Sca。

求一个做题顺序，使得 AC 系数和最大。

3．分析

这道题其实有点类似于欧几里得旅行商问题，做题有一个先后的顺序，根据题目的数据范围应该很容易想到用状态 DP 去做，DP[i]表述状态 i 所对应的最优值，这道题因为每道题只能选一次，所以可以用记忆化搜索的方法去做，而且最好先对原始的每两个题应该做的顺序做一个预处理，可以减少记忆化搜索的层数，实现更好的时间效率。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define N 16
using namespace std;
int dp[1<<N];
int map[N][N];//不同做题顺寻产生的 AC 系数的矩阵
int max(int a,int b)
{
    return a>b?a:b;
}
void init()
{
    memset(dp,-1,sizeof(dp));
}
int dfs(int state,int n)//记忆化搜索的部分，n 表示的是题数
{
    if(dp[state]!=-1)
    return dp[state];
    int i,j,temp,s;
    for(i=0;i<n;i++)
    {
        if(state&(1<<i))//对每一个二进制位都进行枚举，如果包含在状态里便去掉且计算最优值
        {
            temp=dfs(state^(1<<i),n);
            s=state^(1<<i);
            for(j=0;j<n;j++)
            {
                if(s&(1<<j))
                temp+=map[i][j];
            }
            dp[state]=max(dp[state],temp);//得到最优值
        }
    }
    return dp[state];
}
int main()
{
    int n,i,j;
    while(scanf("%d",&n)!=EOF)
    {
        init();
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
```

```
            scanf("%d",&map[i][j]);
            if(n==1)
            {
                printf("0\n");
                continue;
            }
            for(i=0;i<n-1;i++)
            for(j=i+1;j<n;j++)
            dp[1<<i|1<<j]=max(map[i][j],map[j][i]);//对只有两个二进制为 1 的状态预处理
            printf("%d\n",dfs((1<<n)-1,n));
        }
        return 0;
    }
```

2.4.4 扩展变型

POJ 1170 Shopping Offers
六进制的状态 DP
POJ 2817 WordStack

2.5 树形动态规划

编写：曹振海　　校核：黄李龙

2.5.1 树形动态规划介绍

树形动态规划，是在一个树形结构上进行的决策选择，因为是树形结构，所以决策就会受到边的连接关系的影响，树形动态规划能够处理的题目很多，也很繁杂，很多时候要和图论相互结合着用，不过很多树形动态规划的基本原理和一些背包问题很类似。

2.5.2 解题思路

树形动态规划属于一个难点，因为能处理的题型太多，所以很多时候不知道怎么用树形动态规划，状态转移方程也很不好写，不过有的时候也可以从题面中给出的一些数据中判断，一般用得到树形动态规划的题，所给的树的节点的数量都是 10^5 这样的数量级，因为这种数量级用 n^2 时间复杂度的算法就已经难以解决了，或者说题中有明显的出现求最优值这样的话，就可以试一下树形动态规划的思路。

2.5.3 经典题目

2.5.3.1 POJ1463-Strategic game

1．题目出处/来源

POJ1463-Strategic game

2．题目描述

Bob 很喜欢玩儿游戏，但是因为游戏有时候很难，不知道怎么玩儿会让他觉得不高兴，现在有一个问题，在一个城市中，街道的构成是树形的，他想知道最少要在多少个节点上驻守士兵能够监视到所有的道路。

3．分析

这道题是很简单的一个树形动态规划，和 01 背包特别像，因为每个点都有选和不选两种情况，所以定义 dp[i][0]表示以 i 为根节点的子树不选 i 点的最优值，dp[i][1]表示选 i 点的最优值，dp[i][0]= $\sum dp[j][1]$，其中 j 是 i 的子节点，dp[i][1]= $\sum \min(dp[j][0], dp[j][1])$(j 为 i 的子节点)。

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<stdio.h>
#include<string.h>
#define MAX 1501
int pre[MAX],child[MAX];
int dp[MAX][2];//dp[i][1]表示以 i 为根节点的子树选择 i 的最优值，dp[i][0]为不选的最优值
int min(int a,int b)
{
   return a<b?a:b;
}
int n;
int used[MAX];
void fun(int r)
{
   used[r]=1;
   int i;
   if(child[r]==0)
   {
      dp[r][1]=1;
      dp[r][0]=0;
      return;
   }
   int m=child[r];
   int dp0=0;
   int dp1=0;
   for(i=0;i<n;i++)
   {
      if(pre[i]==r)//对每一个子孙节点进行 DP
      {
         if(!used[i])
         {
            fun(i);
         }
         dp1+=min(dp[i][1],dp[i][0]);//状态转移方程
         dp0+=dp[i][1];
      }
   }
   dp[r][1]=dp1+1;
   dp[r][0]=dp0;
}
int main()
{
   int i,nroot,root,m,cld;
   while(scanf("%d",&n)!=EOF)
   {
      memset(pre,-1,sizeof(pre));
      memset(child,-1,sizeof(child));
      memset(dp,0,sizeof(dp));
      memset(used,0,sizeof(used));
      for(i=0;i<n;i++)
      {
         scanf("%d:(%d)",&nroot,&m);
         if(!i)
         root=nroot;
         child[nroot]=m;
         while(m--)
         {
            scanf("%d",&cld);
            pre[cld]=nroot;
            if(cld==root)
```

```
                root=nroot;
            }
        }
        fun(root);
        printf("%d\n",min(dp[root][1],dp[root][0]));//最后也要在根节点选和不选中选最优值
    }
    return 0;
}
```

5．思考与扩展：可以借鉴北大培训教材中做法。

2.5.3.2 POJ3140-Contestants Division

1．题目出处/来源

POJ3140-Contestants Division

2．题目描述

这道题是说给定一棵树，每个节点有自己的权值，求去掉一条边，使得分割成的两棵树的权值之差最小

3．分析

这道题的$\leq N \leq 100000, 1 \leq M \leq 1000000$，但是很明显 M 不会达到那么大，因为 n 个点的树边数是确定的，这道题中，树形搜索只是一部分，现在很多题的树形 DP 都会和图论结合起来，成为题里的一部分，这就使得想到用树形 DP 更难。要用搜索先得到以每一个点为根节点的树的所有节点的权值和 dp[i],那么去掉从根节点到 i 点的路径上直接连接 i 点的那条边所得到的两棵树的权值分别为 sum-dp[i]和 dp[i]差值为 abs(sum-2*dp[i])

4．代码（包含必要注释，采用最适宜阅读的 Courier New 字体，小五号，间距为固定值 12 磅）

```
#include<iostream>
#include<cstdio>
#include<cstring>
#define N 100005
#define M 1000005
using namespace std;
int head[N];//邻接表存储
struct edge
{
    int v;
    int next;
};
edge e[M];
int t;
long long v[N];//点的权值
long long dp[N];//以 i 为根节点的子树的权值的和
bool used[N];
void init()
{
    memset(head,-1,sizeof(head));
    memset(used,0,sizeof(used));
    memset(dp,0,sizeof(dp));
    t=0;
}
void add(int x,int y)//加边操作
{
    e[t].v=x;
    e[t].next=head[y];
    head[y]=t++;
    e[t].v=y;
    e[t].next=head[x];
    head[x]=t++;
}
void dfs(int u)
{
    used[u]=true;
```

```
        dp[u]=v[u];
        int i;
        for(i=head[u];i>=0;i=e[i].next)
        {
            if(!used[e[i].v])
            {
                dfs(e[i].v);
                dp[u]+=dp[e[i].v];
            }
        }
    }
    long long abs(long long x)
    {
        return x>=0?x:-x;
    }
    long long min(long long a,long long b)
    {
        return a<b?a:b;
    }
    int main()
    {
        int n,m;
        int i,j,k;
        int icase=1;
        while(scanf("%d%d",&n,&m)&&(m||n))
        {
            init();
            long long all=0;
            long long ans;
            for(i=1;i<=n;i++)
            {
                scanf("%lld",&v[i]);
                all+=v[i];
            }
            while(m--)
            {
                scanf("%d%d",&j,&k);
                add(j,k);
            }
            dfs(1);
            printf("Case %d: ",icase++);
            ans=abs(all-2*dp[1]);
            for(i=2;i<=n;i++)
            ans=min(ans,abs(all-2*dp[i]));
            printf("%lld\n",ans);
        }
        return 0;
     }
```

2.6 利用单调性质优化动态规划

参考文献：

JSOI2009 集训队论文 《用单调性优化动态规划》

IOI2004 国家集训队论文 周源 《浅谈数形结合思想在信息学竞赛中的应用》

单调队列+斜率优化DP：http://www.notonlysuccess.com/index.php/dp_optimize/

编写：黄李龙　　校核：黄李龙

2.6.1 利用单调性优化最长上升子序列

POJ 3903 Stock Exchange

题意：已知 L 天的股票价格，分别是 p1, p2 ••• pL，现在要求一个最长的上升子序列 pi1 < pi1 < ••• <pik，并且有 i1<i2< ••• <ik，输出其长度。

输入：

有多组测试数据，每组测试数据的第一行为一个整数 L（L<=100000）。第二行为 L 个整数，为连续 L 天的股票价格。

输出：

如题中描述的最长的长度。

分析：很经典的最长上升子序列问题，DP 方程 F[i] = MAX{F[j] + 1} (1<=j< i，p[j] < p[i])，否则 F[i] = 1。求解的关键在于能否优化查找最大的 F[j]，且满足 p[j] < p[i]。

假如F[i]的最优解为F[j]+1，那么有p[j] < p[i]，且F[j] < F[i]。再考虑另外一种情况，当a<b，且F[a]=F[b]时，有p[a] > p[b]，因为如果p[a]<p[b]，则F[b]=F[a]+1 会得到最优值，但是F[a]=F[b]，所以有p[a] > p[b]。假设当前处理到第i天的股票，我们用H[c]表示，序列长度为c时，第c天的的股票价格最小是多少，可以看出H[]数组是单调递增的。并且，我们求得当前F[i]的最优解时，有H[F[i]] = p[i]，且为最小（证明可以根据之前的提示得出）。那么我们可以对H[]数组进行二分查找的操作，寻找一个最大的长度l，且满足H[l]<p[i]。时间复杂度为$O(N\log_2 N)$。

C++代码：

```
#include <cstdio>

int h[100005];

int main() {
 int n, a;
 while (EOF != scanf("%d", &n)) {
     scanf("%d", &a);
     int ans = 1;
     h[1] = a;
     for (int i = 1; i < n; ++i) {
         scanf("%d", &a);
         int l = 1, r = ans;
         while (l <= r) {
             int m = (l + r) / 2;
             if (h[m] < a) {
                 l = m + 1;
             } else {
                 r = m - 1;
             }
         }
         h[l] = a;
         if (ans < l) ans = l;
     }
     printf("%d\n", ans);
 }
 return 0;
}
```

类似的题目有 POJ 1631 Bridging signals 和 Hrbustoj 1427 Leyni 的情人节，其中 Leyni 的情人节这题需要做一些转换才能利用最长下降子序列的方法做。

2.6.2 单调队列

什么是单调（双端）队列

单调队列，顾名思义，就是一个元素单调的队列，那么就能保证队首的元素是最小（最大）的，从而满足动态规划的最优性问题的需求。

单调队列，又名双端队列。双端队列，就是说它不同于一般的队列只能在队首删除、队尾插入，它能够在队首、队尾同时进行删除。

单调队列的性质

一般，在动态规划的过程中，单调队列中每个元素一般存储的是两个值：

在原数列中的位置（下标）

他在动态规划中的状态值

而单调队列则保证这两个值同时单调。

单调队列有什么用

我们来看这样一个问题：一个含有 n 项的数列(n<=2000000)，求出每一项前面的第 m 个数到它这个区间内的最小值。

这道题目，我们很容易想到线段树、或者 st 算法之类的 RMQ 问题的解法。但庞大的数据范围让这些对数级的算法没有生存的空间。我们先尝试用动态规划的方法。用 $f(i)$ 代表第 i 个数对应的答案，$a[i]$ 表示第 i 个数，很容易写出状态转移方程：

$$f(i) = \min_{j=i-m+1}^{i} (a[j])$$

这个方程，直接求解的复杂度是 O(nm)的，甚至比线段树还差。这时候，单调队列就发挥了他的作用：

我们维护这样一个队列：队列中的每个元素有两个域{position,value}，分别代表他在原队列中的位置和 $a[i]$，我们随时保持这个队列中的元素两个域都单调递增。

那计算 $f(i)$ 的时候，只要在队首不断删除，直到队首的 position 大于等于 $i-m+1$，那此时队首的 value 必定是 $f(i)$ 的不二人选，因为队列是单调的！

我们看看怎样将 $a[i]$ 插入到队列中供别人决策：首先，要保证 position 单调递增，由于我们动态规划的过程总是由小到大（反之亦然），所以肯定在队尾插入。又因为要保证队列的 value 单调递增，所以将队尾元素不断删除，直到队尾元素小于 $a[i]$。

时间效率分析

很明显的一点，由于每个元素最多出队一次、进队一次，所以时间复杂度是 O(n)。用单调队列完美的解决了这一题。

为什么要这么做

我们来分析为什么要这样在队尾插入：为什么前面那些比 $a[i]$ 大的数就这样无情的被枪毙了？我们来反问自己：他们活着有什么意义？！由于 $i-m+1$ 是随着 i 单调递增的，所以对于 $\forall j < i, a[j] > a[i]$，在计算任意一个状态 $f(x), x >= i$ 的时候，j 都不会比 i 优，所以 j 被枪毙是“罪有应得”。

我们再来分析为什么能够在队首不断删除，一句话：$i-m+1$ 是随着 i 单调递增的！

小结

对于这样一类动态规划问题，我们可以运用单调队列来解决：

$$f(x) = \operatorname{opt}_{i=bound[x]}^{x-1} (const[i])$$

其中 bound[x]随着 x 单调不降，而 const[i]则是可以根据 i 在常数时间内确定的唯一的常数。这类问题，一般用单调队列在很优美的时间内解决。

2.6.3 直接利用单调队列解题

2.6.3.1 Hrbustoj 1522 子序列的和

题目描述：

输入一个长度为 n 的整数序列（A1,A2,……,An），从中找出一段连续的长度不超过 m 的子序列，使得这个子序列的和最大。

输入：

有多组测试数据，不超过 20 组测试数据。

对于每组测试的第一行，包含两个整数 n 和 m(n,m<=10^5)，表示有 n 个数，子序列长度限制为 m，表示这个序列的长度，第二行为 n 个数，每个数的范围为[-1000, 1000]。

输出：

对于每组测试数据，输出最大的子序列和，并换行。

样例

输入：

3 1

1 2 3

3 2

-1000 1000 1

输出

3

1001

分析：

F(i)为以 Ai 结尾长度不超过 M 的最大子序和

$$F(i) = \max\{\sum_{j=i-k+1}^{i} A_j \mid k = 1..m\}$$

对于每个 F(i)，从 1 到 m 枚举 k 的值，完成 Aj 的累加和取最大值。

该算法的时间复杂度为$O(N^2)$

简化方程

令$S(i) = \sum_{j=1}^{i} A_j$

$$\begin{aligned} F(i) &= \max\{\sum_{j=i-k+1}^{i} A_j \mid k = 1..m\} \\ &= \max\{S(i) - S(i-k) \mid k = 1..m\} \\ &= S(i) - \min\{S(i-k) \mid k = 1..m\} \end{aligned}$$

单调队列优化

在算法中，考虑用队列来维护决策值 S(i-k)。每次只需要在队首删掉 S(i-m-1)，在队尾添加 S(i-1) 。但是取最小值操作还是需要 O(n)时间复杂度的扫描。

考察在添加 S(i-1)的时候，设现在队尾的元素是 S(k)，由于 k<i-1，所以 S(k)必然比 S(i-1)先出队。若此时 S(i-1)<=S(k)，则 S(k)这个决策永远不会在以后用到，可以将 S(k)从队尾删除掉（此时队列的尾部形成了一个类似栈的结构）

同理，若队列中两个元素 S(i)和 S(j)，若 i<j 且 S(i)>=S(j)，则我们可以删掉 S(i)（因为 S(i)永远不会被用到）。此时的队列中的元素构成了一个单调递增的序列，即：

S1<S2<S3<……<Sk

我们来整理在求 F(i)的时候，用队列维护 S(i-k)所需要的操作：

☆若当前队首元素 S(x)，有 x<i-m，则 S(x)出队；直到队首元素 S(x)有 x>=i-m 为止。

☆若当前队尾元素 S(k)>=S(i-1)，则 S(k)出队；直到 S(k)<S(i-1)为止。

☆在队尾插入 S(i-1)。

☆取出队列中的最小值，即队首元素。

由于对于求每个 F(i)的时候，进队和出队的元素不止一个。

但是我们可以通过分摊分析得知，每一个元素 S(i)只进队一次、出队一次，所以队列维护的时间复杂度是 O(n)。而每次求 F(i)的时候取最小值操作的复杂度是 O(1)，所以这一步的总复杂度也是 O(n)。

综上所述，该算法的总复杂度是 O(n)

C++代码：

```
#include <cstdio>
#include <cstring>

const int INF = 999999999;
const int MaxN = 1000005;

struct Node {
 int i, s;
 Node() {}
 Node(int ti, int ts) : i(ti), s(ts) {}
} que[MaxN];

int s[MaxN];

int main() {
 int n, m;
 while (EOF != scanf("%d%d", &n, &m)) {
     s[0] = 0;
     int ans = -INF;
     for (int i = 1; i <= n; ++i) {
         scanf("%d", &s[i]);
         s[i] += s[i-1];
     }
     Node *fr = que, *ta = que;
     que[0] = Node(0, 0);
     ans = s[1];
     for (int i = 2; i <= n; ++i) {
         while (fr <= ta && fr->i < i - m) ++fr;
         while (fr <= ta && ta->s >= s[i-1]) --ta;
         *++ta = Node(i-1, s[i-1]);
         int t = s[i] - fr->s;
         if (ans < t) ans = t;
     }
     printf("%d\n", ans);
 }
 return 0;
 }
```

2.6.3.2 其他直接应用单调队列的题目

FZU 1894 志愿者选拔

HDU 3415 Max Sum of Max-K-sub-sequence

POJ 2823 Sliding Window

2.6.4 单调队列优化动态规划

2.6.4.1 HDU 3530 Subsequence

题意：

给一个长度为 n 的数列，要求一个子区间，使得区间的最大值与最小值的差 s 满足,m<=s<=k，求满足条件的最长子区间的长度。

输入：

有多组测试数据，对于每组测试数据：

第一行是 n m k，分别表示题目中的变量。

第二行有 n 个整数，每个整数的的范围是[0, 1000000]。

输出：

最长的长度。

分析：先考虑朴素的算法，枚举区间的右边界 i，再枚举左边界 j，然后找出区间[j,i]的最大和最小值，然后判断是否满足，更新答案。很明显会超时。

考虑如何优化。如果某次枚举的左边界 j 使得区间[j,i]满足条件，那么就不用继续枚举 j 了，因为再枚举也不会比当前更优。那么问题就转化成对于每一个右边界 i 找到一个最小的左边界 j 使得区间[j,i]满足最大最小值之差在给定范围内。

一个贪心的想法就是维护[1,i]这个区间的最值，因为 1 最小，但是这个区间可能并不满足，也就是说，j 还可能从 1 继续往右枚举。

看一个结论，如果对于一个右边界 i，其最优区间的左边界是 j，那么对于以后所有的 i`>i 都有 i`的最优区间左边界 j'>=j。这个结论用反证法证明。

枚举 i 的时候，j 都是单调不递减的，这提示我们可以使用单调队列来维护了，于是有如下方法：

维护两个单调队列，一个单调不递减，记录最小值，一个单调不递增，记录最大值；

每次枚举一个 i，先将 i 加入到两个单调队列的队尾，注意维护各自队列的单调性；

检查两个单调队列的队首元素，如果两个元素之差大于题目所给出的上界 k 则将队首元素序号较小的那个队列的队首元素出队，并且更新左边界 j。（出队是为了使两个队首元素之差在规定范围内。想一想，为何要将序号较小的出队？）

如果两个队首元素之差大于等于题目所给出的下界 m 则更新答案。

注意左边界 j 初始化为 0。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 100005;

int a[MAXN];
int q1[MAXN], q2[MAXN];

int main() {
    int n, low, up;
    while (EOF != scanf("%d%d%d", &n, &low, &up)) {
        int ans = 0, h1 = 0, t1 = 0, h2 = 0, t2 = 0, j = 0;
        for (int i = 0; i < n; ++i) {
            scanf("%d", &a[i]);
```

```
            while (h1 < t1 && a[q1[t1-1]] < a[i]) --t1;
            q1[t1++] = i;
            while (h2 < t2 && a[q2[t2-1]] > a[i]) --t2;
            q2[t2++] = i;
            while (a[q1[h1]] - a[q2[h2]] > up) {
                if (q1[h1] < q2[h2]) {
                    j = q1[h1] + 1;
                    ++h1;
                } else {
                    j = q2[h2] + 1;
                    ++h2;
                }
            }
            int t = a[q1[h1]] - a[q2[h2]];
            if (low <= t && t <= up) {
                t = i - j + 1;
                if (ans < t) ans = t;
            }
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

2.6.4.2 HDU 3401 Trade

题意：

给出 T 天股票的信息，每天的信息分别可以表示为：APi , BPi , ASi , BSi，分别表示第 i 天时股票的买入单价、卖出单价、第 i 天最多可以买入的股票数、最多可以卖出的股票

数，并且还有一个约束条件，那就是两个交易日相距的天数要大于W天，即如果是在第 i 天交易了，下次的交易时间只能从 i+W+1 开始，每天最多能持有 MaxP 的股票数量。求最大获利。

输入：

有多组测试数据，对于每组测试数据：

第一行为 T MaxP W，含义如题目描述。

接下来是 T 行，第 i+1 行为第 i 天的信息：APi BPi ASi BSi，含义如题目描述。

输出：

最大获利。

分析：

参考： http://blog.csdn.net/ivan_zjj/article/details/7559985

这题的 DP 方程很容易列，用 f[i][j]表示第 i 天持有 j 股的最大拥有的钱数，转移方程为：

不做交易：

f[i][j] = MAX(f[i][j] , f[i-1][j])

买入：

f[i][j] = MAX(f[pre][k] - (j - k) * AP[i])

我们把不变的量 j * AP[i] 提出来，则：

f[i][j] + j * AP[i] = MAX(f[pre][k] + k *AP[i])

令 Fa(k) = f[pre][k] + k * AP[i] ，则：f[i][j] = MAX(FF(k)) - j * AP[i]

这样就转化为经典的单调队列优化的 DP 了，我们只需要用一个单调队列来存放决策点 k 就可以了。

卖出：

f[i][j] = MAX(f[pre][k] + (k - j)*BP[i])

我们把不变的量 j * AP[i] 提出来，则：

f[i][j] + j * AP[i] = MAX(f[pre][k] + k *BP[i])

令 Fb(k) = f[pre][k] + k * BP[i] ，则：f[i][j] = MAX(Fb(k)) - j *AP[i]

情况和上面的类似。

这样每次求 f[i][j] 的时候，我们可以得出以下的一个重要结论：在第 i 天的时候，最优的 pre

应该是：pre = i - W - 1。其实证明这个结论很简单， 假设 pre1 < pre2，我们假设 f[pre1][k] >

f[pre2][k] ，但是这种情况是不可能出现的，因为我们可以这样考虑，我现在从 pre1 天到

pre2 天这中间都不交易，那么我就可以得到 f[pre2][k] >= f[pre1][k]，所以说上面的假设是

不可能成立的，因此我们每次只要将 pre 赋值为 i - W - 1 即可。

剩下的工作就是每次求 f[i][j] 的时候分别用两个队列分别维护买入和卖出就可以了。

C++代码：

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 2005, INF = 0x3fffffff;

int ap[MAXN], bp[MAXN], as[MAXN], bs[MAXN];
int f[MAXN][MAXN];
int q[MAXN];

template<class T>
void check_max(T &a, const T &b) {
   if (a < b) a = b;
}

int main() {
   int runs;
   scanf("%d", &runs);
   while (runs--) {
      int n, mp, w;
      scanf("%d%d%d", &n, &mp, &w);
      for (int i = 1; i <= n; ++i) {
         scanf("%d%d%d%d", &ap[i], &bp[i], &as[i], &bs[i]);
      }
      for (int i = 1; i <= n; ++i) {
         for (int j = 0; j <= as[i]; ++j) f[i][j] = -j * ap[i];
         for (int j = as[i] + 1; j <= mp; ++j) f[i][j] = -INF;
      }
      for (int i = 2; i <= n; ++i) {
         for (int j = 0; j <= mp; ++j) check_max(f[i][j], f[i-1][j]);
         if (i - w - 1 <= 0) continue;
         int fr = 0, ta = 1, pr = i - w - 1;
         q[0] = 0;
         for (int j = 1; j <= mp; ++j) {
            while (fr < ta && j - q[fr] > as[i]) ++fr;
            if (fr < ta) check_max(f[i][j], f[pr][q[fr]] - ap[i] * (j - q[fr]));
            while (fr < ta && f[pr][q[ta-1]] + ap[i] * q[ta-1] <= f[pr][j] + ap[i]
* j) --ta;
            q[ta++] = j;
         }
```

```
            fr = 0, ta = 1;
            q[0] = mp;
            for (int j = mp-1; j >= 0; --j) {
               while (fr < ta && q[fr] - j > bs[i]) ++fr;
               if (fr < ta) check_max(f[i][j], f[pr][q[fr]] + bp[i] * (q[fr] - j));
               while (fr < ta && f[pr][q[ta-1]] + bp[i] * q[ta-1] <= f[pr][j] + bp[i]
* j) --ta;
               q[ta++] = j;
            }
         }
         int ans = -INF;
         for (int j = 0; j <= mp; ++j) {
            if (ans < f[n][j]) ans = f[n][j];
         }
         printf("%d\n", ans);
      }
      return 0;
   }
```

2.6.4.3 POJ 3017 Cut the Sequence

以下分析来着：JSOI2009 集训队论文 《用单调性优化动态规划》 。

问题描述

给定一个有 n 个非负整数的数列 a，要求将其划分为若干个部分，使得每部分的和不超过给定的常数 m，并且所有部分的最大值的和最小。其中 n<=105。

例：n=8, m=17，8 个数分别为 2 2 2 | 8 1 8 |1 2，答案为 12，分割方案如图所示。

• 解法分析

刚开始拿到这道题目，首先要读好题：最大值的和最小。

首先设计出一个动态规划的方法：

$f(i)=\underset{j=b[x]}{\overset{i-1}{Max}}(f[j]+Maxnumber[j+1,i])$，其中 $f(i)$ 代表把前 i 个数分割开来的最小代价。$b[i]=Min(j \mid sum[j+1,i]<=m)$，$b[i]$可以用二分查找来实现。

直接求解复杂度最坏情况下（M 超大）是 $O(n^2)$ 的，优化势在必行。

通过仔细观察，可以发现以下几点性质：

在计算状态 f(x)的时候，如果一个决策 k 作为该状态的决策，那么可以发现第 k 个元素和第 x 个元素是不分在一组的。

b[x]随着 x 单调不降的，用这一点，可以想到什么？可以想到前面单调队列的一个限制条件。

来看一个最重要的性质：如果一个决策 k 能够成为状态 f(x)的最优决策，当且仅当 $a[k]>\forall a[j], j\in[k+1,x]$。为什么呢？其实证明非常非常容易（用到性质 1），交给读者自己考虑。

到此为止，我们可以这样做：由于性质三，每计算一个状态 f(x)，它的有

效决策集肯定是一个元素值单调递减的序列，我们可以像单调队列那样每次在队首删除元素，直到队首在数列中的位置小于等于 $b[x]$，然后将 a[x]插入队尾，保持队列的元素单调性。

这时候问题来了，队首元素一定是最佳决策点吗？我们只保证了他的元素值最大……如果扫一遍队列，只是常数上的优化，一个递减序足以将它否决。

我们观察整个操作，将队列不断插入、不断删除。对于除了队尾的元素之外，每个队列中的元素供当前要计算的状态的“值”是 $f(q[x].position)+a[q[x+1].position]$，其中,q[x]代表第 x 个队列元素，position 这代表他在原来数组中的位置，我们不妨把这个值记为 t。那每一次在队首、队尾的删除就相当于删除 t，每一次删除完毕之后又要插入一个新的 t，

然后需要求出队列中的 t 的最小值。

我们发现，完成上述一系列工作的最佳选择就是平衡树，这样每个元素都插入、删除、查找各一遍，复杂度为 O(logn),最后的时间复杂度是 $O(n\log n)$ 。

有一个细节：$b[x]$ 这一个单独的决策点是不能够被省掉的（仍然留给读者思考），而上述队列的方法有可能将其删除，所以要通过特判来完成。

以上就是《用单调性优化动态规划》中的分析内容，。因为此题所给的数据量小，分析中所说的求数队列中的 t 的最小值可以直接枚举得到。当然，也可以使用 C++ STL 中的 multiset 去维护当前最小值。下面给出两份代码，分别是直接枚举获得队列最小值和用 multiset 维护队列最小值的两种方法。

```
// 直接枚举获得队列最小值的方法
#include <cstdio>
#include <string>

typedef long long int64;

const int MAXN = 100005;

int64 a[MAXN];
int64 f[MAXN];
int d[MAXN];
int q[MAXN];

template <class T>
void check_min(T &a, const T b) {
 if (a > b) a = b;
}

int main() {
 int n;
 int64 m;
 while (EOF != scanf("%d%lld", &n, &m)) {
     a[0] = 0;
     bool can = true;
     for (int i = 1, t; i <= n; ++i) {
         scanf("%d", &t);
         a[i] = t;
         if (t > m) can = false;
     }
     if (!can) {
         puts("-1");
         continue;
     }
     int fr = 0, re = 0, low = 0;
     f[0] = 0;
     int64 sum = 0;
     for (int i = 1; i <= n; ++i) {
         sum += a[i];
         while (sum > m) {
             sum -= a[++low];
         }
         while (fr < re && q[fr] < low) {
             ++fr;
         }
         while (fr < re && a[q[re-1]] <= a[i]) {
             --re;
         }
         if (fr < re) {
             d[re] = q[re-1];
         } else {
             d[re] = low;
         }
```

```
            q[re++] = i;
            if (d[fr] < low) {
                d[fr] = low;
            }
            f[i] = f[d[fr]] + a[q[fr]];
            for (int j = fr + 1; j < re; ++j) {
                check_min(f[i], f[d[j]] + a[q[j]]);
            }
        }
        printf("%lld\n", f[n]);
    }
    return 0;
}
```

// 用 multiset 维护队列最小值

```
#include <cstdio>
#include <string>
#include <set>
using namespace std;

typedef long long int64;

const int MAXN = 100005;

int64 a[MAXN];
int64 f[MAXN];
int d[MAXN];
int q[MAXN];

template <class T>
void check_min(T &a, const T b) {
    if (a > b) a = b;
}

int main() {
    int n;
    int64 m;
    while (EOF != scanf("%d%lld", &n, &m)) {
        a[0] = 0;
        bool can = true;
        for (int i = 1, t; i <= n; ++i) {
            scanf("%d", &t);
            a[i] = t;
            if (t > m) can = false;
        }
        if (!can) {
            puts("-1");
            continue;
        }
        int fr = 0, re = 0, low = 0;
        f[0] = 0;
        int64 sum = 0;
        multiset<int64> ms;
        for (int i = 1; i <= n; ++i) {
            sum += a[i];
            while (sum > m) {
                sum -= a[++low];
            }
            while (fr < re && q[fr] < low) {
                ms.erase(f[d[fr]] + a[q[fr]]);
                ++fr;
            }
            while (fr < re && a[q[re-1]] <= a[i]) {
                ms.erase(f[d[re-1]] + a[q[re-1]]);
                --re;
            }
            if (fr < re) {
                d[re] = q[re-1];
            } else {
```

```
                d[re] = low;
            }
            q[re++] = i;
            ms.insert(f[d[re-1]] + a[q[re-1]]);
            if (d[fr] < low) {
                ms.erase(f[d[fr]] + a[q[fr]]);
                d[fr] = low;
                ms.insert(f[d[fr]] + a[q[fr]]);
            }
            /*
            f[i] = f[d[fr]] + a[q[fr]];
            for (int j = erasefr + 1; j < re; ++j) {
                check_min(f[i], f[d[j]] + a[q[j]]);
            }
            */
            f[i] = *ms.begin();
        }
        printf("%lld\n", f[n]);
    }
    return 0;
}
```

2.6.4.4 其他应用单调队列或者单调性题目

POJ 3245 Sequence Partitioning

POJ 1742 Coins　单调队列优化多重背包

HDU 3474 Necklace

2.6.5 利用斜率的单调性

基础入门：MAX　Average Problem

HDU 2993 MAX　Average Problem

题目描述：给一个长度为 N，仅包含正整数的序列，序列为 a1,a2 ••• an，还给出一个不超过 N 的正整数 K，定义 AVE(i,j)为 ai ••• aj 的平均值，求出最大的 AVE(i,j)，并且 1<=i<=j-K+1<=N。

输入：

有多组测试数据，对于每组测试数据：

第一行为 N　K。

第二行为 N 个整数，a1 a2 ••• an。变量含义在题目描述中有说明。

输出：

最大的平均值，精确到小数点后两位。

分析：（一下分析来自 IOI2004 国家集训队论文 周源 《浅谈数形结合思想在信息学竞赛中的应用》）

简单的枚举算法可以这样描述：每次枚举一对满足条件的(a, b)，即 a≤b-F+1，检查 ave(a, b)，并更新当前最大值。

然而这题中 N 很大，N2 的枚举算法显然不能使用，但是能不能优化一下这个效率不高的算法呢？答案是肯定的。

目标图形化

首先一定会设序列 ai 的部分和：Si=a1+a2+…+ai,，特别的定义 S0=0。

这样可以很简洁的表示出目标函数：

$$ave(i,j)=\frac{S_j-S_{i-1}}{j-(i-1)}$$

如果将 S 函数绘在平面直角坐标系内，这就是过点 Sj 和点 Si-1 直线的斜率！于是问题转化为：平面上已知 N+1 个点，Pi(i, Si)，0≤i≤N，求横向距离大于等于 F 的任意两点连线的最大斜率。

构造下凸折线

有序化一下，规定对 i<j，只检查 Pj 向 Pi 的连线，对 Pi 不检查与 Pj 的连线。也就是说对任意一点，仅检查该点与在其前方的点的斜率。于是我们定义点 Pi 的检查集合为

Gi = {Pj, 0≤j≤i-F}

特别的，当 i<F 时，Gi 为空集。

其明确的物理意义为：在平方级算法中，若要检查 ave(a, b)，那么一定有 Pa∈Gb；因此平方级的算法也可以这样描述，首先依次枚举 Pb 点，再枚举 Pa∈Gb，同时检查 k(PaPb)。

若将 Pi 和 Gi 同时列出，则不妨称 Pi 为检查点，Gi 中的元素都是 Pi 的被检查点。

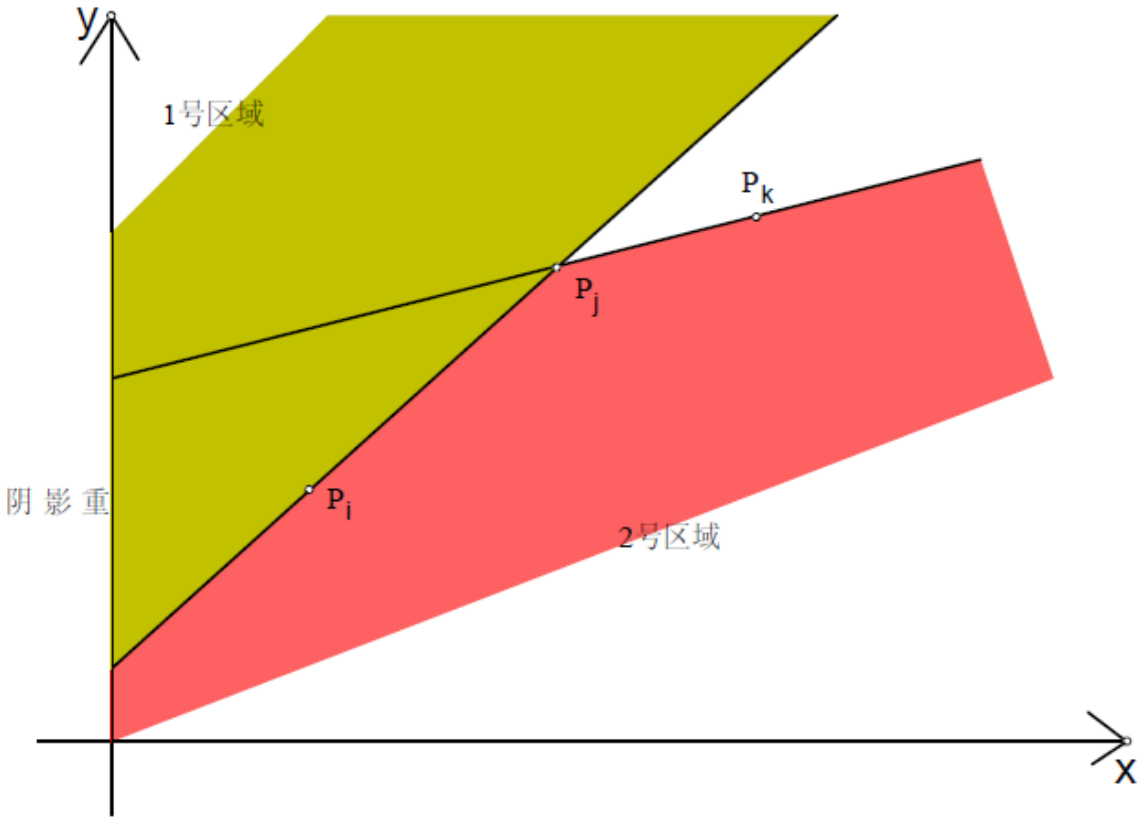
当我们考察一个点 Pt 时，朴素的平方级算法依次选取 Gt 中的每一个被检查点 p，考察直线 pPt 的斜率。但仔细观察，若集合内存在三个点 Pi, Pj, Pk，且 i<j<k，三个点形成如下图所示的的关系，即 Pj 点在直线 PiPk 的上凸部分：k(Pi, Pj)>k(Pj,Pk)，就很容易可以证明 Pj 点是多余的。

以上就是论文中的分析，我认为写得很好，写得很直观，把题目转换一下，利用了斜率的单调性去解决，推荐好好看一遍这篇论文。

C++代码：

```
#include <cstdio>
typedef long long int64;

const int MAXN = 100005;

int q[MAXN];
int64 s[MAXN];

int64 cross(int64 x1, int64 y1, int64 x2, int64 y2) {
    return x1 * y2 - x2 * y1;
}

int main() {
    int n, k;
    while (EOF != scanf("%d%d", &n, &k)) {
        s[0] = 0;
        for (int i = 1; i <= n; ++i) {
            int t;
            scanf("%d", &t);
            s[i] = s[i-1] + t;
        }
        int fr = 0, ta = 0;
        double ans = 0.0;
        for (int i = k; i <= n; ++i) { // 单调队列里维护的应该是可以选择的决策点，虽然不一定是最优点
            int j = i - k;
            while (fr<ta-1 && cross(q[ta-1]-q[ta-2], s[q[ta-1]]-s[q[ta-2]], j-q[ta-2], s[j]-s[q[ta-2]]) <= 0) {
                --ta;
            }
            q[ta++] = j;
            while (fr+1<ta && (s[i]-s[q[fr+1]])*(i-q[fr]) >= (s[i]-s[q[fr]])*(i-q[fr+1])) {
                ++fr;
```

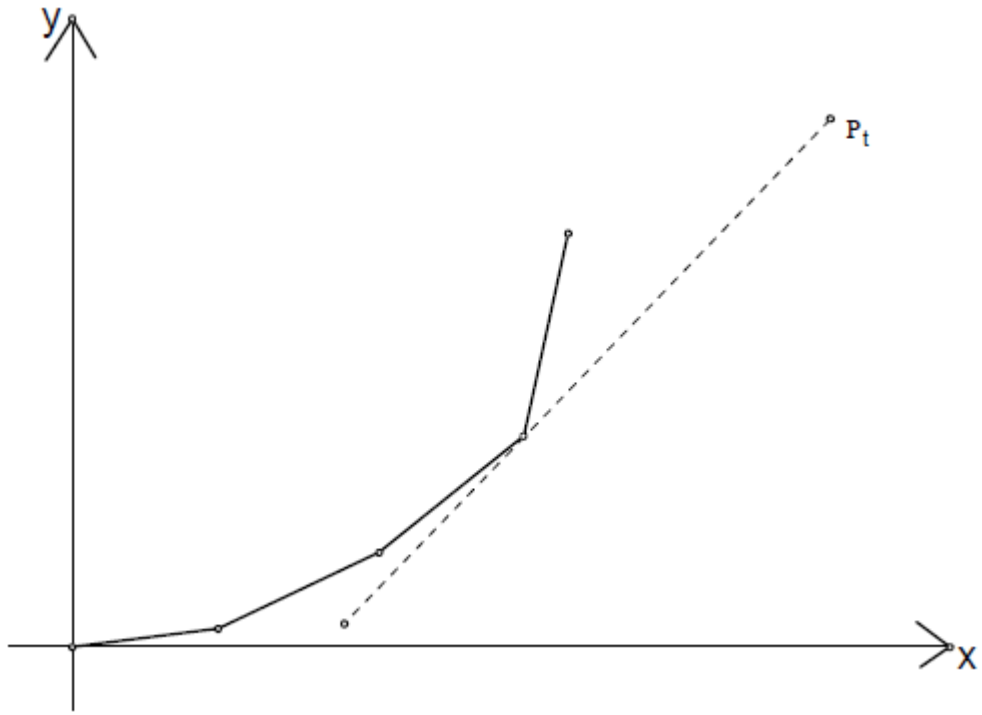
```
            }
            double t = (double)(s[i] - s[q[fr]]) / (i - q[fr]);
            if (ans < t) ans = t;
        }
        printf("%.2lf\n", ans);
    }
    return 0;
}
```



若 k(Pt, Pj) > k(Pt, Pi)，那么可以看出，Pt 点一定要在直线 PiPj 的上方，即阴影所示的 1 号区域。同理若 k(Pt, Pj) > k(Pt, Pk)，那么 Pt 点一定要在直线 PjPk 的下方，即阴影所示的 2 号区域。

综合上述两种情况，若 PtPj 的斜率同时大于 PtPi 和 PtPk 的，Pt 点一定要落在两阴影的重叠部分，但这部分显然不满足开始时 t>j 的假设。于是，Pt 落在任何一个合法的位置时，PtPj 的斜率要么小于 PtPi，要么小于 PtPk，即不可能成为最大值，因此 Pj 点多余，完全可以从检查集合中删去。

这个结论告诉我们，任何一个点 Pt 的检查集合中，不可能存在一个对最优结果有贡献的上凸点，因此我们可以删去每一个上凸点，剩下的则是一个下凸折线。最后需要在这个下凸折线上找一点与 Pt 点构成的直线斜率最大——显然这条直线是在与折线相切时斜率最大，如下图所示。



维护下凸折线

这一小节中，我们的目标是：用尽可能少的时间得到每一个检查点的下凸折线。算法首先从PF开始执行：它是检查集合非空的最左边的一个点，集合内仅有一个元素P0，而这显然满足下凸折线的要求，接着向右不停的检查新的点：P_{F+1},P_{F+2}, …, P_N。

检查的过程中，维护这个下凸折线：每检查一个新的点 Pt，就可以向折线最右端加入一个新的点 Pt-F，同时新点的加入可能会导致折线右端的一些点变成上凸点，我们用一个类似于构造凸包的过程依次删去这些上凸点，从而保证折线的下凸性。由于每个点仅被加入和删除一次，所以每次维护下凸折线的平摊复杂度为 O(1)，即我们用 O(N)的时间得到了每个检查集合的下凸折线。

最后的优化：利用图形的单调性

最后一个问题就是如何求过 Pt 点，且与折线相切的直线了。一种直接的方法就是二分，每次查找的复杂度是 O(log2N)。但是从图形的性质上很容易得到另一种更简便更迅速的方法：由于折线上过每一个点切线的斜率都是一定的 3，而且根据下凸函数斜率的单调性，如果在检查点 Pt 时找到了折线上的已知一个切点 A，那么 A 以前的所有点都可以删除了：过这些点的切线斜率一定小于已知最优解，不会做出更大的贡献了。

于是另外保留一个指针不回溯的向后移动以寻找切线斜率即可，平摊复杂度为为 O(1)。

至此，此题算法时空复杂度均为 O(N)，得到了圆满的解决。

小结

回顾本题的解题过程，一开始就确立了以平面几何为思考工具的正确路线，很快就发现了检查集合中对最优解有贡献的点构成一个下凸函数这个重要结论，之后借助计算几何中求凸包的方法维护一个下凸折线，最后还利用下凸函数斜率的单调性发现了找切线简单方法。题解围绕平面几何这个中心，以斜率为主线，整个解题过程一气呵成，又避免了令人头晕的代数式变换，堪称以形助数的经典例题。

这里仅仅对斜率优化的方法做一些入门的介绍，可以参看其他利用斜率优化的题目：

HDU 2829 Lawrence

HDU 3507 Print Article

2.6.6 扩展推荐

利用单调栈解题：POJ2082 POJ2559 POJ 2796。

四边形不等式优化:在 JSOI2009 集训队论文 《用单调性优化动态规划》和 IOI2004 国家集训队论文 周源 《浅谈数形结合思想在信息学竞赛中的应用》都有介绍，推荐阅读。