一、数学建模（20 分）

请对下面的生产计划问题进行分析，然后建立该问题的数学模型。

生产计划问题：某工厂生产甲、乙两种产品，甲产品每生产一件需消耗黄铜 2kg、3 个工作日、两个外协件，每件可获利润 60 元；乙产品每生产一件需消耗黄铜 4kg、1 个工作日、不需要外协件，每件可获利润 30 元，该厂每月可供生产用的黄铜 320kg，总工作日 180 个，外协件 100 个。问怎样安排生产才能使工厂的利润最高？

答案：

用 $\mathbf{x_1}, \mathbf{x_2}$ 分别表示甲、乙两种产品生产的件数，该厂追求的目标是获取最高利润，用数学表达式表示为：

$$\max f = 60x_1 + 30x_2$$

由于生产甲、乙产品的件数要受到生产能力的约束，即

黄铜约束：$2x_1 + 4x_2 \le 320$,

工作日约束：$3x_1 + x_2 \le 180$,

外协件约束：$2x_1 \le 100$,

非负约束：　$x_1, x_2 \ge 0$.

这样，该生产计划问题就归结为如下数学模型：

$$\max f = 60x_1 + 30x_2, \quad （4 分）$$

$$\begin{cases} 2x_1 + 4x_2 \le 320, & （4 分） \\ 3x_1 + x_2 \le 180, & （4 分） \\ 2x_1 \le 100, & （4 分） \\ x_1, x_2 \ge 0. & （4 分） \end{cases}$$

二、算法填空

两个空格，第一个给 5 分，第二给 7 分

```
// ?1  return(S);
// ?2  S:=S*(m1*2n+(m1+m2+m3)*2n/2+m3);
```

时间代价给 8 分

设 X 和 Y 都是 n 位的二进制整数，现在要计算它们的乘积 XY。我们可以用小学所学的方法来设计一个计算乘积 XY 的算法，但是这样做计算步骤太多，显得效率较低。如果将每 2 个 1 位数的乘法或加法看作一步运算，那么这种方法要作 $O(n^2)$步运算才能求出乘积 XY。下面我们用分治法来设计一个更有效的大整数乘积算法。

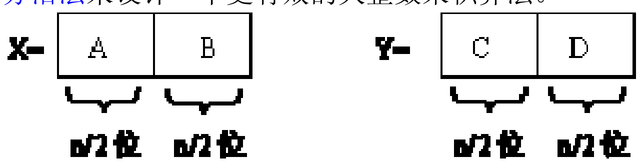


图　大整数 X 和 Y 的分段

我们将 n 位的二进制整数 X 和 Y 各分为 2 段，每段的长为 n/2 位(为简单起见，假设 n 是 2 的幂)，如图 6-3 所示。

由此，$X=A2^{n/2}+B$ ，$Y=C2^{n/2}+D$。这样，X 和 Y 的乘积为：

$$XY=(A2^{n/2}+B)(C2^{n/2}+D)=AC2^{n}+(AD+CB)2^{n/2}+BD \qquad (1)$$

如果按式(1)计算 XY，则我们必须进行 4 次 n/2 位整数的乘法(AC，AD，BC 和 BD)，以及 3 次不超过 n 位的整数加法(分别对应于式(1)中的加号)，此外还要做 2 次移位(分别对应于式(1)中乘 2^n 和乘 $2^{n/2}$)。所有这些加法和移位共用 $O(n)$步运算。设 T(n)是 2 个 n 位整数相乘所需的运算总数，则由式(1)，我们有:

$$\begin{cases} T(1)=1 \\ T(n)=4T(n/2)+O(n) \end{cases} \qquad (2)$$

由此可得 $T(n)=O(n^2)$。因此，用(1)式来计算 X 和 Y 的乘积并不比小学生的方法更有效。要想改进算法的计算复杂性，必须减少乘法次数。为此我们把 XY 写成另一种形式:

$$XY=AC2^n+[(A-B)(D-C)+AC+BD]2^{n/2}+BD \qquad (3)$$

虽然，式(3)看起来比式(1)复杂些，但它仅需做 3 次 n/2 位整数的乘法(AC，BD 和(A-B)(D-C))，6 次加、减法和 2 次移位。由此可得:

$$\begin{cases} T(1)=1 \\ T(n)=3T(n/2)+cn \end{cases} \qquad (4)$$

用解递归方程的套用公式法马上可得其解为 $T(n)=O(n^{\log 3})=O(n^{1.59})$。

三、算法辨析（25 分）

二叉搜索堆（Binary Search Heap,简称 BSH）是一颗二叉树，它的内部结点由一个标签 label 和权值 priority 组成，并且同时满足二叉搜索树和堆的性质，即（1）每个结点的左子树中所有结点的 label 小于该结点的 label，而右子树中所有结点的 label 大于该结点的 label；(2) 每个结点的权值都小于其父结点的权值。

请判断下述二叉搜索堆的构建算法是否正确。

（1） 如果正确，请通过演示该算法运行步骤，给出由 7 个标签权值对 a/3 b/6 c/4 d/7 e/2 f/5 g/1 一步步构建成的二叉搜索堆的状态变化过程。

（2） 如果不正确，请在原题中指出错误之处，并给出改正后的结果。然后给出由 7 个标签权值对 a/3 b/6 c/4 d/7 e/2 f/5 g/1 一步步构建成的二叉搜索堆的状态变化过程。请不要撇开原算法，自己重写一套。

```
struct Node        //二叉搜索堆结点定义
{
      string label;      //记录该结点的标号
      int priority;       //该结点的权
      Node *left;         //指向该结点的左子女
      Node *right;       //指向该结点的右子女
      Node * parent;    //指向该结点的父结点
};
Node * buildTreap( Node *list, int n) //构建 n 个结点的 BSH。list 为存放 n 个结点的数组，且已经按 label 从小到大排完序。函数最后返回 BSH 的根结点指针 root
{
      Node * root = &list[0]; //list 第一个结点地址赋给根结点
      Node * pNew;              //pNew 指向每次要插入的新结点
      Node * p;                  //p 指向树在中序周游下的最后一个结点
      p = root; //刚开始时，p 和 root 指向同一个结点，即 list[0]
      for(int i = 1; i <= n; i ++)
```

```
    {
        pNew = & list[i];                    //指向要插入的结点
        if( root ->priority < pNew->priority )//新插入结点权值比根结点权值大
        {
            pNew->right = root;
            root->parent = pNew;
            root = pNew;
        }
        else
        {
            while( p->priority > pNew->priority ) //从树的最右下角开始往上寻找第一个比要插入结点(pNew)权大的结点
                p = p->parent;
            pNew->left = p->right;//p 的右子树成为新结点的左子树
            if(p->right != NULL )
                p->right->parent = pNew;
            p->left = pNew;//新结点作为 p 的右子女
            pNew->parent = p;
        }
        p = pNew;
    }
    return root; //返回根结点指针
}
```

其中，上述程序段中往一个二叉检索堆插入新结点的方法是：(1)如果新插入的结点的权大于根结点的权，就将新结点作为新树的根，并将原树作为新树根结点的左子树。(2)否则，就从树的最右下的结点（即中序周游下的最后一个结点，也是上一次插入的结点）往上搜第一个比新结点权大的结点 x。将 x 的右子树设为新结点的左子树，将新结点作为 x 的右子女。

答案：

判断出错误给 1 分

上述着重标记的四行分别有错误，应改正为：

for(int i = 1; i < n; i ++) 即将<=改为<

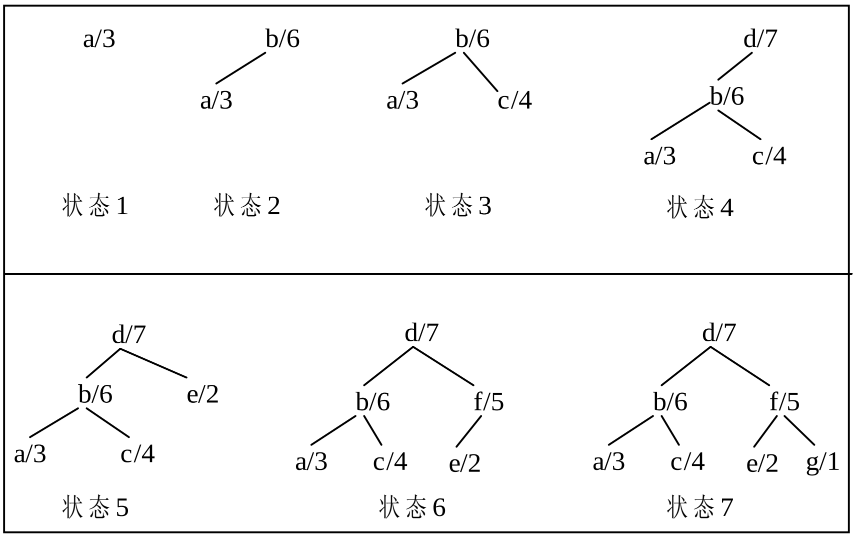
pNew->left = root; 即将 right 改为 left

while(p->priority < pNew->priority) 即将>改为<

p->right = pNew; 即将 left 改为 right

四个错误每改对一个给 2.5 分（如果指出错误点，没改对也给 1 分），共 10 分。

二叉搜索的状态变化过程如下：



每个状态给对给 2 分，共 14 分

四、算法设计

1、 背包问题参考答案。

算法思想：

回溯法，逐个枚举物品放在哪一个包中。

程序代码段：

```
int cap[MAX_M];//每个包的容量，MAX_M 为包的最大值上限。
int weight[MAX_N];//每个物品的质量，MAX_N 为物品数上限。
int m, n;//m 为包数，n 为物品数。
int put[MAX_N];//put[i]表示第 i 个物品放在哪个包。
int Search(int k)
/*表示前 k（0 至 k-1)个物品已经放好，现在要放编号为 k 的物品的处理
返回 1 表示可以接受，继续处理 k+1;
返回 0 表示不可以接受，需要回溯.*/
{
	if(k==n)
		return 1;
	else
	{
		int i;
		for(i=0;i<m;i++)//枚举每个物品，直至找到解。
		{
			if(cap[i]>=weight[k])
			{
				cap[i]-=weight[k];//i 包可以放第 k 个物品
				if(Search(k+1)==1)//处理 k+1，如果成功，返回 1，否则 i++，再处理
				{
					put[k]=i;
```

```
                return 1;
            }
            cap[i]+=weight[k];//第 k+1 个物品处理不成功，回溯 k
        }
    }//end of for
}//end of else
}
```

程序说明：

以上程序是一个递归程序，其中初始化时 cap[0..m-1]为背包初始容量，调用 Search(0)根据返回值可判断是否有解。

若有解，结果保存在全局变量 put 数组里。

2、计算围棋目问题参考答案

算法思想：

利用“*”所代表的棋子做边界，填充所有外围棋子（包含“*”棋子），并计数，则用总棋子数减去被填充数即为所求。

程序代码段：

```
int board[MAX+2][MAX+2];/*count[i][j]值表示 i 行 j 列是否有棋子，1 表示有棋子，0 表示无棋子，当程序调用 FloodFill 填充完毕时，0 值的个数即为围棋的目。*/
int n;//棋盘为 n*n 大小
int dir[4][2]={ {0,-1}, {0,1}, {-1,0}, {1,0} };//四个方向向量，相对原棋子所在位置的偏移量。
int count=(n+2)*(n+2);
int FloodFill(int x, int y)//x,y 表示填充的坐标
{
    board[x][y]=1;
    count--;
    int d;
    for(d=0;d<4;d++)
    {
        int _x, _y;
        _x=x+dir[d][0];
        _y=y+dir[d][1];

        if( _x>=0&& _x<=n+1 && _y>=0 && _y<=n+1)//test _x,_y is on board.
        {
            if(board[_x][_y]==0)
                FloodFill(_x, _y);
            else
                count--;
        }//end if on board
    }//end for
}
```

程序说明：

棋盘存在 board[1..n][1..n]里，board[0][0..n+1]、board[n+1][0..n+1]、board[0..n+1][0]和 board[0..n+1][n+1]是为防止棋盘边界上放棋子的情况所添加的一圈无棋子盘。
程序调用 FloodFill(0, 0)，然后输出 count 值即为围棋的目。

五、程序设计

1. 魔术师穿墙

注意不需要循环输入。
【解法分析】
此题的贪心做法：从左到右扫描所有的列，
1) 记录在该列出现的墙，并统计总宽度 wid；
2) 若 wid > k，则删除右边界最靠后的 wid - k 面墙。
在扫描之前先将所有的墙对右边界排序，可提高第 2 步的效率。
算法的时间复杂度为 O(n * logn + n * k)。
【参考程序】

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

using namespace std;

// 记录墙的数据结果，left 和 right 分别记录墙左右两端的 x 坐标
struct Wall {
    int left, right;
} wall[100];

// 比较两段墙右端点的函数，在排序里使用
bool cmp(const Wall& a, const Wall& b) {
    return a.right < b.right;
}

int main() {
    int t, i, j;
    int strength;           // 魔术师能够穿越 strength 堵墙
    int wallNum;            // 墙的数目
    int ans;                // 问题答案
    bool isDeleted[100];    // 记录某堵墙是否被删除

    cin >> t;
    for (; t > 0; t--) {
        cin >> wallNum >> strength;
        for (i = 0; i < wallNum; i++) {
```

```
            cin >> wall[i].left >> j;
            cin >> wall[i].right >> j;
            if (wall[i].left > wall[i].right)
                swap(wall[i].left, wall[i].right);
        }
        sort(wall, wall + wallNum, cmp);    // 按墙的右端点的 x 坐标对墙进行排序
        for (i = 0; i < wallNum; i++) isDeleted[i] = false;
        ans = 0;
        for (i = 0; i <= 100; i++) {
            int check[100];// 记录还没有被删除，而且满足 left <= i <= right 的墙
            int checkNum;   // check 中存储的墙的数目
            checkNum = 0;
            for (j = wallNum - 1; j >= 0; j--)
                if (!isDeleted[j] && wall[j].left <= i && wall[j].right >= i) {
                    check[checkNum++] = j;
                }
            for (j = 0; j < checkNum - strength; j++) {
                isDeleted[check[j]] = true;     // 删除魔术师不能穿越的墙
                ans++;                          // 答案加一
            }
        }
        cout << ans << endl;
    }
    return 0;
}
```

2. 最大的全 1 子矩阵

一个 n*n 的矩阵 A，A 中任意一个元素只能取值 0 或 1。
求出最大的 m*m 全 1 子矩阵（子矩阵中元素全部为 1），m<=n。
1 考虑矩阵 A 中 A[i,j]点元素向左的最长的全 1 的序列个数，记为 Sx[i,j]。
  若 A[i,j]位于某行最左边第一个，则 Sx[i,1] = A[i,1];
  若 A[i,j] ==0 ,则 Sx[i,j] = 0;
  否则 Sx[i,j] = Sx[i,j-1] + 1.

0	1	1	0	1
1	1	1	1	1
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

则有递推公式：

$$Sx[i,j] = \begin{cases} A[i,j] & , \quad j==1 \\ 0 & , A[i,j] == 0 \ \&\& \ j > 1 \\ Sx[i,j-1]+1, & A[i,j] ==1 \ \&\& \ j>1 \end{cases}$$

考虑矩阵 A 中 A[i,j]点元素向上的最长的全 1 的序列个数，记为 Sy[i,j]。
则有递推公式：

Sy[i,j] = A[i,j] , i = =1
0 , A[i,j] == 0 && i > 1
Sy[i-1,j]+1, A[i,j] ==1 && i>1

由于计算 A[i,j]对应的 Sx[i,j], Sy[i,j], M[i,j]时每步都是常数时间，所以总的时间复杂度是 O(n^2)，空间复杂度是 O(n^2)。M 矩阵中元素的最大值就是要求的 m 值。

举例：如下的矩阵 A：

0	1	1	0	1
1	1	1	1	1
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

对应求得 Sx

0	1	2	0	1
1	2	3	4	5
1	0	1	2	3
1	2	3	4	5
1	0	0	1	0

对应求得 Sy

0	1	1	0	1
1	2	2	1	2
2	0	3	2	3
3	1	4	3	4
4	0	0	4	0

对应求得 M

0	1	1	0	1
1	1	2	1	1
1	0	1	2	2
1	1	1	2	3
1	0	0	1	0

```
int FindMaxSubMatrix(int** A, int n)
```

```
{
    int m = 0;
    for(i = 0; i <= n; i++)
        for(j = 0; j <=n; j++)
        {
            if (i == 0 || j == 0)
            {
                Sx[i,j] = A[i,j];
                Sy[i,j] = A[i,j];
                M[i,j] = A[i,j];
                if(A[i,j] == 1)
                    m = 1;
            }
            else if (A[i,j] == 0)
            {
                Sx[i,j] = 0;
                Sy[i,j] = 0;
                M[i,j] = 0;
            }
            else
            {
                Sx[i,j] = Sx[i,j-1] + 1;
                Sy[i,j] = Sy[i-1,j] + 1;
                M[i,j] = MIN(M[i-1,j-1]+1, Sx[i,j], Sy[i,j]);
                if (M[i,j] > m)
                {
                    m = M[i,j];
                }
            }
        }
    return m;

}
```