# 第二题：一个工厂制造的产品形状都是长方体

**输入输出样例，共有两组有效输入：**第一组表示有4 个3\*3 的产品和一个6\*6 的产品，此时4 个3\*3 的产品占用一个箱子，另外一个6\*6 的产品占用1 个箱子，所以箱子数是2；

第二组表示有7 个1\*1 的产品，5 个2\*2的产品和1 个3\*3 的产品，我们可以把他们统统放在一个箱子中，所以输出是1。

**分析6个型号具体情况如下：**

* 6\*6 的产品每个会占用一个完整的箱子，并且没有空余空间；
* 5\*5 的产品每个占用一个新的箱子，并且留下11 个可以盛放1\*1的产品的空余空间；
* 4\*4 的产品每个占用一个新的箱子，并且留下5 个可以盛放2\*2 的产品的空余空间；
* 3\*3 的产品情况比较复杂，首先3\*3 的产品不能放在原来盛有5\*5 或者4\*4 的箱子中，那么必须为3\*3 的产品另开新的箱子，新开的箱子数目等于3\*3 的产品的数目除以4 向上取整；同时我们需要讨论为3\*3 的产品新开箱子时，剩余的空间可以盛放多少2\*2 和1\*1 的产品（这里如果有空间可以盛放2\*2 的产品，我们就将它计入2\*2 的空余空间，等到2\*2 的产品全部装完，如果还有2\*2 的空间剩余，再将它们转换成1\*1 的剩余空间）。

我们可以分情况讨论为3\*3 的产品打开的新箱子中剩余的空位，共为四种情况：

第一种，3\*3 的产品的数目正好是4 的倍数，所以没有空余空间；

第二种，3\*3 的产品数目是4 的倍数加1，这时还剩5 个2\*2 的空位和7 个1\*1 的空位；

第三种，3\*3 的产品数目是4 的倍数加2，这时还剩3 个2\*2 的空位和6 个1\*1 的空位；

第四种，3\*3 的产品数目是4 的倍数加3，这时还剩1 个2\*2 的空位和5 个1\*1 的空位；

* 处理完3\*3 的产品，就可以比较一下剩余的2\*2的空位和2\*2 产品的数目，如果产品数目多，就将2\*2 的空位全部填满，再为2\*2 的产品打开新箱子，同时计算新箱子中1\*1 的空位，如果剩余空位多，就将2\*2 的产品全部填入2\*2的空位，再将剩余的2\*2 的空位转换成1\*1 的空位；
* 最后处理1\*1 的产品，比较一下1\*1的空位与1\*1 的产品数目，如果空位多，将1\*1 的产品全部填入空位，否则，先将1\*1 的空位填满，然后再为1\*1 的产品打开新的箱子。

**参考程序：**

1. #include <stdio.h>
2. **void** main()
3. {
4. **int** N, a, b, c, d, e, f, y, x；//N 用来存储需要的箱子数目，y 用来存储2\*2 的空位数目
5. // x 用来存储1\*1 的空位数目。
6. **int** u[4]={0, 5, 3, 1};
7. //数组u 表示3\*3 的产品数目分别是4 的倍数，4 的倍数+1, 4 的倍数+2, 4 的倍数+3
8. //时，为3\*3 的产品打开的新箱子中剩余的2\*2 的空位的个数
9. **while**(1){
10. scanf("%d%d%d%d%d%d", &a, &b, &c, &d, &e, &f);
11. **if** (a == 0 && b == 0 && c == 0 && d == 0 && e == 0 && f == 0) **break**;
12. N = f + e + d + (c + 3) / 4;
13. //这里有一个小技巧 - (c+3)/4 正好等于c 除以4 向上取整的结果,下同
14. y = 5 \* d + u[c % 4];
15. **if**(b > y) N += (b - y + 8 ) / 9;
16. x = 36 \* N - 36 \* f - 25 \* e - 16 \* d - 9 \* c - 4 \* b;
17. **if**(a > x) N += ( a - x + 35 ) / 36;
18. printf("%d\n", N);
19. }
20. }

# 第三题：滑雪问题

**分析及代码实现：**

这个问题应该来说是个简单的，很容易想到用动态规划去做的题目。这个问题满足最有子结构是比较容易看出来。

非常容易建立如下递归式：  
如果从i,j可以顺着某侧滑的话：  
dis\_sk[i][j] = max{dis\_sk[i-1][j],dis\_sk[i][j-1],dis\_sk[i+1][j],dis\_sk[i][j+1]}+1

那么我们很容易写出递归的：  
int dis(int i,int j){  
  for(i,j上侧，下侧，左侧，右侧)  
      if(该位置没有越界){  
         if(顺着该侧可以往下滑)  
            如果该侧位置可以滑行的距离(递归调用dis函数)大于dis\_sk[i][j]，则把dis\_sk[i][j]改成该距离+1  
      }  
}  
把这个递归改成动态规划很容易，只要在开始判断一下  
if(dis\_sk[i][j]) return dis\_sk[i][j]; //dis\_sk[i][j]开始为0  
这样基本上就可以很顺畅的写出代码了：  
定义的变量如下：

cpp 代码

1. int h[101][101];//输入的高度值
2. int dis\_sk[101][101];//记录了每个点可以滑行的最大距离
3. int dx[]={-1,1,0,0};//为了方便上下左右侧的滑行的最大距离而使用的方便数组
4. int dy[]={0,0,-1,1};
5. int r,c;//输入的行和列

 一个用来判断是否越界的辅助函数：

cpp 代码

1. bool in\_bound(int i,int j){
2. **return** i >= 0 && i < r && j >= 0 && j < c;
3. }

下边就是以lookup方式写的动态规划实现的从i,j下滑最大距离：

cpp 代码

1. int dis(int i,int j){
2. int temp;
3. **if**(dis\_sk[i][j])//如果已经求出来了，直接返回
4. **return** dis\_sk[i][j];
5. **for**(int k = 0; k < 4; k++){
6. **if**(in\_bound(i+dx[k],j+dy[k])){//如果没有越界
7. **if**(h[i][j] > h[ i+dx[k] ][ j+dy[k] ]){//如果顺着该侧可以滑
8. temp = dis(i+dx[k],j+dy[k]);//递归求dis(i+dx[k],j+dy[k])，并保存在临时变量temp中
9. dis\_sk[i][j] = dis\_sk[i][j] > temp ? dis\_sk[i][j] : temp + 1;//如果dis\_sk[i][j]比temp小，则取temp+1
10. }
11. }
12. }
13. **return** dis\_sk[i][j];
14. }

最后是main函数，取dis(i,j)的最大者就是所求的最长区域的长度：

1. **int** main(){
2. **int** max\_dis = 0;
3. **int** temp;
4. **int** i,j;
5. cin >> r >> c;
6. **for**(i = 0; i < r; i++)
7. **for**(j = 0; j < c; j++){
8. cin >> h[i][j];
9. dis\_sk[i][j] = 0;
10. }
12. **for**(i = 0; i < r; i++)
13. **for**(j = 0; j < c; j++){
14. temp = dis(i,j);
15. max\_dis = max\_dis > temp ? max\_dis : temp;
16. }
18. cout << max\_dis + 1 << endl;
19. **return** 0;
20. }