****优势策略均衡（）**也称**占优策略均衡**,是指不管其他人采取什么策略，每个博弈者都会找到对自己最有利的策略所构成的一个[策略组合](http://wiki.mbalib.com/wiki/%E7%AD%96%E7%95%A5%E7%BB%84%E5%90%88" \o "策略组合)。其实质为：不管其他参与者如何行动，每个参与者都有一个对自己来说最好的策略。例子：囚徒困境。 **纳什均衡**，又称为**非合作博弈均衡，**需给定其他参与者的行动，每个参与者才可以选择一个对自己来说是最好的策略。其核心思想是“换位思考”。例子：智猪博弈**

**优势策略均衡和纳什均衡的区别在于：**

**在优势策略均衡中，我所做的是不管你做什么，我所能做的是最好的（大多数博弈情况下，优势策略均衡是不存在的）。**

**在纳什均衡中，我所做的是给定你所做的前提下，我所能做的是最好的，你所做的是在给定我所做的前提下你所能做的是最好的，从二者的关系可以看出，优势策略均衡是纳什均衡的一个特例，一个优势策略均衡首先是一个纳什均衡。**

**优势策略均衡一定是纳什均衡，但纳什均衡不一定是优势策略均衡。**

**分治（Divide and Conquer）**

* **设*X*[0:n-1]和*Y*[0:n-1]为两个数组，每个数组中含有*n*个已排好序的数。试设计一个O(log*n*)时间的分治算法，找出*X*和*Y*的2*n*个数的中位数，并证明算法的时间复杂性为O(log*n*)**
  + **个数为奇数,则处于最中间位置的数**
  + **个数为偶数,则中间两个数据的平均数**

**Answer:**

**(1). Algorithm thought：Take the divide and conquer algorithm, mark the median of X and Y as resultMedian, the procedure is as follows:**

**Step1: Find the median of array X[0, n-1], and mark it as x0;**

**Find the median of array Y[0, n-1], and mark it as y0;**

**Step2: Compare x and y:**

**If (x0 = y0) resultMedian = x0;**

**If (x0 < y0) resultMedian must be in X[x0, n-1] or Y[0, y0];**

**If (x0 > y0) resultMedian must be in X[0, x0] or Y[y0, n-1].**

**Step3: Loop step1 ande step2 for the subarrays of array X and Y**

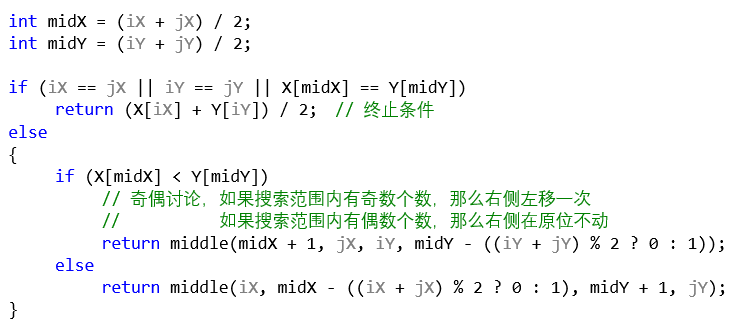
**until finding the resultMedian.**

**(2). Algorithm time complexity analysis**

**Assuming that after a times division, the count of array elements reach 1, then we have , so .**

**So the time complexity is O(logn).**

**伪代码：**

****

* **有一实数序列，若 且 ，则构成了一个逆序对，请使用分治方法求整个序列中逆序对个数，并分析算法的时间复杂性。**
  + **例如：序列(4,3,2)逆序对有(4,3)，(4,2)，(3,2)共3个**

**Answer:**

**(1). Algorithm thought：Take the divide and conquer algorithm, just like the merge sort algorithm, the procedure is as follows:**

**Step1: Let array A = [], and divide the A into 2 sub arrays from the middle: A1[, A2[, for A1,A2, do the same division, until there is only one element in the subarrays.**

**Step2: Comquer: for each subarray, compare and sort, and count the number of reverse order pair.**

**(2). Algorithm time complexity analysis**

**According to Master theorem, so**

* **给定一个已按从小到大排序的数列，数列长度为N。数列中只有一个数只出现过一次，其余每个数均出现两次，请设计一个算法，找出那个只出现了一次的那个数。**

**Answer:**

**思路:类似于二分查找，采用分治法，不断递归，每次将中间数与其前后的数进 行比较，直到找到左右都不等于它的那个数。**

**伪代码：**

**if (length == 1)**

* **return Part[0];**
* **int mid = (length - 1) / 2;**
* **if (mid % 2 == 0)**
* **{**
* **if (Part[mid] == Part[mid - 1])**
* **{**
* **int \*sub = &Part[0];**
* **return once(sub, mid);**
* **}**
* **else if (Part[mid] == Part[mid + 1])**
* **{**
* **int \*sub = &Part[mid + 1];**
* **return once(sub, mid);**
* **}**
* **else**
* **return Part[mid];**
* **}**
* **else**
* **{**
* **if (Part[mid] == Part[mid - 1])**
* **{**
* **int \*sub = &Part[mid + 1];**
* **return once(sub, mid);**
* **}**
* **else if (Part[mid] == Part[mid + 1])**
* **{**
* **int \*sub = &Part[0];**
* **return once(sub, mid);**
* **}**
* **else**
* **return Part[mid];**
* **轮廓线**

**这种算法的思路是：要求n个矩形的轮廓，先将这n个矩形分成两个大小相等的部分，分别求其轮廓，然后再将这两个轮廓合并。**

**规模为1的问题可以直接解决。具体来说，如果这个矩形的三元组表示为(L,H,R)，那么其轮廓为(L,H,R,0)。**

**对于规模为k的问题，假设得到了两个规模为k/2的轮廓，分别为A和B，我们如何得到合并后的轮廓C？首先，容易证明轮廓C的每一个横坐标，都来源于轮廓A和B的横坐标，而不会产生新的坐标值。因此，我们只需计算A和B中所有涉及到的横坐标在C中的高度。由于轮廓C中的横坐标值要求有序，我们可以仿照归并排序的方法，用两个指针扫描轮廓A和B。具体的方法是，设指针i指向轮廓A的当前横坐标，指针j指向轮廓B的当前横坐标。如果指针i指向的横坐标较小，那么将这一横坐标加入到C中，且在C中的高度为A中第i个横坐标对应的高度与B中第j-1个横坐标对应的高度的最大值，然后将指针i向后移一位；指针j指向的横坐标较小的情况则类似处理。如果两个指针指向的横坐标相同，此时只需将这一横坐标加入到C中一次，且高度为两指针指向高度的最大值，然后将两指针同时向后移一位。最后，需要扫描一遍轮廓C，将相邻的高度相同的横坐标合并。**

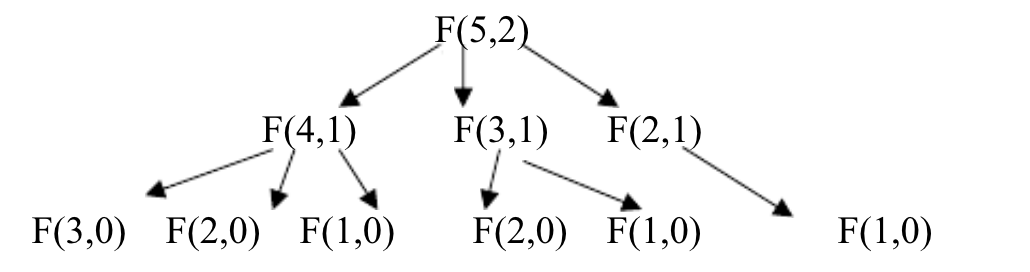
**分析时间复杂度，设T(n)表示解决规模为n的问题需要的时间，那么有。解此递归方程，得到T(n)=O(nlogn)。空间方面，由于递归的层数为O(logn)，每一层需要保存O(n)的空间，所以总的空间复杂度为O(nlogn)。**

* **#define max(a,b) (a >= b? a : b) ;**
* **const int BuildingNumber = 8;**
* **typedef struct line**
* **{**
* **int start = -1;**
* **int high = -1;**
* **}\*couple;**
* **typedef struct triple**
* **{**
* **int left;**
* **int right;**
* **int high;**
* **}\*triplepointer;**
* **couple triple2line(triple T)//将元组转化成轮廓线**
* **couple mergeline(couple A, couple B)//合并轮廓线**
* **{**
* **int i = 1, j = 1, k = 1;**
* **couple C;**
* **C = (couple)malloc(sizeof(line)\* (2 \* BuildingNumber + 1));**
* **memset(C, -1, sizeof(line)\* (2 \* BuildingNumber + 1));**
* **C[0].start = 0;**
* **C[0].high = 0;**
* **while ((A[i].start!=-1 )&& (B[j].start != -1))**
* **{**
* **if (A[i].start < B[j].start)**
* **{**
* **C[k].start = A[i].start;**
* **C[k].high = max(A[i].high, B[j - 1].high);**
* **i++;**
* **k++;**
* **}**
* **else if (A[i].start > B[j].start)**
* **{**
* **C[k].start = B[j].start;**
* **C[k].high = max(B[j].high, A[i - 1].high);**
* **j++;**
* **k++;**
* **}**
* **else //if (A[i].start = B[j].start)**
* **{**
* **C[k].start = A[i].start;**
* **C[k].high = max(A[i].high, B[j].high);**
* **i++;**
* **j++;**
* **k++;**
* **}**
* **if (C[k-1].high == C[k - 2].high)**
* **{**
* **k--;**
* **}**
* **}**
* **if (A[i].start == -1)**
* **{**
* **while (B[j].start != -1)**
* **{**
* **C[k].start = B[j].start;**
* **C[k].high = B[j].high;**
* **j++;**
* **k++;**
* **if (C[k-1].high == C[k - 2].high)**
* **k--;**
* **}**
* **}**
* **if (B[j].start == -1)**
* **{**
* **while (A[i].start != -1)**
* **{**
* **C[k].start = A[i].start;**
* **C[k].high = A[i].high;**
* **i++;**
* **k++;**
* **if (C[k-1].high == C[k - 2].high)**
* **k--;**
* **}**
* **}**
* **couple Result;**
* **Result = (couple)malloc(sizeof(line)\*(k+1));**
* **for (int kk = 0; kk < k; kk++)**
* **{**
* **Result[kk].start = C[kk].start;**
* **Result[kk].high = C[kk].high;**
* **}**
* **Result[k].start = -1;**
* **Result[k].high = -1;**
* **return Result;**
* **}**
* **couple Calculate(triplepointer buildings, int buildings\_length)//分治计算**
* **{**
* **if (buildings\_length == 1)**
* **{**
* **couple temp = triple2line(buildings[0]);**
* **return temp;**
* **}**
* **else if (buildings\_length == 2)**
* **{**
* **couple temp1, temp2;**
* **temp1 = triple2line(buildings[0]);**
* **temp2 = triple2line(buildings[1]);**
* **return mergeline(temp1,temp2);**
* **}**
* **else**
* **{**
* **triplepointer A, B;**
* **A = &buildings[0];**
* **B = &buildings[buildings\_length / 2 + 1];**
* **return mergeline(Calculate(A, buildings\_length / 2 +1), Calculate(B, buildings\_length - int(buildings\_length / 2) - 1));**
* **}**
* **}**

**动态规划（Dynamic Programming）**

* **给出*N*个1-9的数字 (*v*1,*v*2,…,*vN*)，不改变它们的相对位置，在中间加入*K*个乘号和*N-K*-1个加号，（括号随便加）使最终结果尽量大。因为乘号和加号一共就是*N*-1个了，所以恰好每两个相邻数字之间都有一个符号。并说明其具有优化子结构性质及子问题重叠性质。**
  + **例如： *N*=5, *K*=2，5个数字分别为1、2、3、4、5，可以加成：**
  + **1\*2\*(3+4+5)=24**
  + **1\*(2+3)\*(4+5)=45**
  + **(1\*2+3)\*(4+5)=45**

**子问题重叠性质举例：**

****

**算法伪代码如下：**

**sum[0..n] //定义一个新数组，用来记录数组v第一个元素到第i个元素的和，1≤i≤n**

**Sum[0]=0;**

**for i=0 to n**

**for j=0 to K**

**f[i][j]=0;**

**for i=1 to n**

**sum[i]=sum[i-1]+v[i]**

**for i=1 to n**

**f[i][0]=sum[i]**

**for i=2 to n**

**t=min(i-1,K)**

**for j=1 to t**

**for h=2 to i**

**s=sum[i]-sum[k-1];**

**f[i][j]=max(f[k-1][j-1]\*s,f[i][j]);**

**return f[n][K]**

**Answers:**

1. **Algorithm thought:**

**Take the dynamic programming algorithm:**

**f (N, K): the optimal solution of N numbers with K multiple sign;**

**Assuming that m is the position of the first multiple sign, m is between the number vm and vm+1. Then the array is divided into two parts, left and right. The result of left is**

**The result of right is f (N-m, k -1).**

**So the state transformation equation is**

1. **Optimal substructure property**

**Subproblem: array vm+1, vm+2, …, vN, K-1 multiple signs, to find the maximum equation value.**

**According to the state transformation equation, we can know that the optimal solution of the main problem includes the optimal results of subproblems.**

1. **Subproblems overlapping property**

**画图。**

**According to the above graph, we can know that the results of subproblems can be used many times.**

* **给定一长度为N的整数序列(a1,*a*2,…,*aN*) ，将其划分成多个子序列（此问题中子序列是连续的一段整数），满足每个子序列中整数的和不大于一个数B，设计一种划分方法，最小化所有子序列中最大值的和。说明其具有优化子结构及子问题重叠性质**

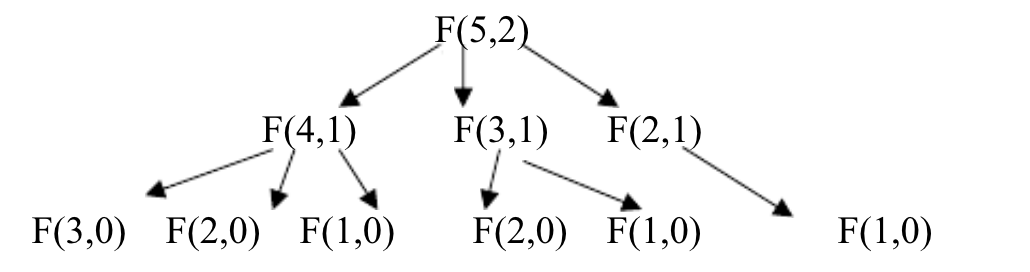
**例如： 序列长度为8的整数序列(2,2,2,8,1,8,2,1)，B=17，可将其划分成三个子序列(2,2,2)，(8,1,8)以及(2,1)，则可满足每个子序列中整数和不大于17，所有子序列中最大值的和12为最终结果**

**解：设S[i]表示从a[1]到a[i]这个整数序列的所有子序列最大值的最小化之和，则有相应的递推式：**

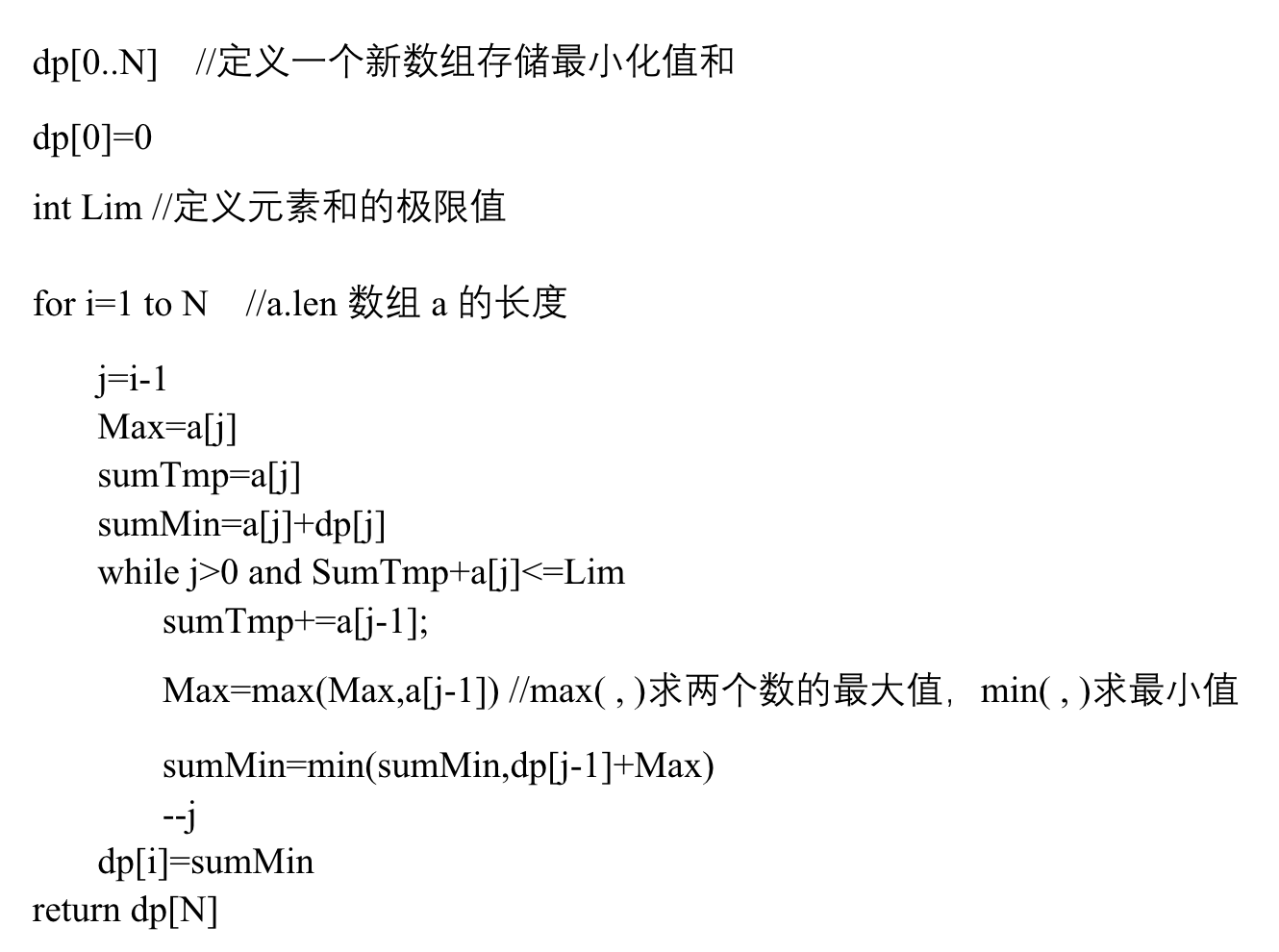
**S[i]=min{S[i−1]+a[i], S[i−j−1]+max{a[i−j−1],…,a[i]}} 其中 a[i-j-1]+a[i-j]+…+a[i]<B，j<i-1，**

**优化子结构性质：若S[i−j−1]不是最小值，则存在S′[i −j−1]<S[i−j−1]，使得存在S′[i]<S[i] S[i]不再是最小化和，与条件相矛盾，因此存在优化子结构性质**

**子问题重叠性质举例：**

****

**算法伪代码如下：**

****

* **基因相似性 X[0:m-1] Y[0:n-1]**

**本题为典型的动态规划，关键找出序列比对的3个不同情况，即子问题**

**设d[i][j]为取s1第i个字符，s2第j个字符时的最大分值**

**则决定p为最优的情况有三种 p数组为分数矩阵**

**1、  s1取第i个字母，s2取“ - ”： d[i-1][j]+p[ s1[i-1] ]['-'];**

**2、  s1取“ - ”，s2取第j个字母：d[i][j-1]+p['-'][ s2[j-1] ];**

**3、  s1取第i个字母，s2取第j个字母：d[i-1][j-1]+p[ s1[i-1] ][ s2[j-1] ];**

**即dp[i][j]为上述三种情况的最大值**

1. **int max3(int x, int y, int z){**
2. **int a = x>y?x:y;**
3. **return a>z?a:z;**
4. **}**
6. **int main(){**
7. **int p[maxlen][maxlen], d[maxlen][maxlen];**
8. **char s1[maxlen], s2[maxlen];**
9. **int len1,len2,T,i,j;**
11. **cin>>T;**
12. **while(T--){**
13. **cin>>len1>>s1>>len2>>s2;**
14. **memset(d, 0, sizeof(d));**
15. **for(i = 1; i <= len1; i++){**
16. **d[i][0] = d[i-1][0] + p[s1[i-1]]['-'];**
17. **}**
18. **for(i = 1; i < len2; i++){**
19. **d[0][i] = d[0][i-1] + p['-'][s2[i-1]];**
20. **}**
21. **for(i = 1; i <= len1; i++){**
22. **for(j = 1;j <= len2; j++){**
23. **d[i][j] = max3(d[i-1][j-1] + p[s1[i-1]][s2[j-1]],**
24. **d[i][j-1] + p['-'][s2[j-1]],**
25. **d[i-1][j] + p[s1[i-1]]['-']);**
26. **}**
27. **}**
28. **cout<<d[len1][len2]<<endl;**
29. **}**
30. **return 0;**
31. **}**

**贪心(Greedy Algorithm)**

* **（1）给定*n*个物品，物品价值分别为*P*1，*P*2，…，*Pn*，物品重量分别*W*1，*W*2, …, *Wn*，背包容量为*M*。每种物品可部分装入到背包中。输出*X*1，*X*2，…，*Xn*，0≤*Xi*≤1, 使得最大，且 ≤M。试设计一个算法求解该问题，分析算法的正确性。**

**Answer:**

1. **Algorithm thought: Take the greedy algorithm. First, calculate the cost performance ratio (Pi/Wi) of all objects, and sort items according to their cost performance from high to low. Each time, place the object with highest cost performance ration to the backpack.**
2. **Optical substructure(最优子结构)**
3. **Greedy Selection（贪心选择性）**

**//找出vector中的最大值下标**

**int maxPos(vector<float> temp)**

**{**

**int maxPos = 0;**

**for (int i = 1; i < temp.size(); i ++)**

**{**

**if(temp[i] > temp[maxPos])**

**maxPos = i;**

**}**

**return maxPos;**

**}**

**int main()**

**{**

**int n = 5; //n个物品**

**int m = 14;**

**vector<float> p = {3.0, 5.0, 7.0, 9.0, 10.0};//n个物品的价值**

**vector<float> w = {2.0, 4.0, 5.0, 2.0, 6.0}; //n个物品的重量**

**vector<float> v; //n个物品的单位价值**

**vector<float> x = {0, 0, 0, 0, 0}; //装入份量**

**for(int i = 0; i < n; i ++)**

**{**

**v.push\_back(p[i] / w[i]);**

**}**

**int max = 0;**

**float value = 0;**

**float weight = 0;**

**for (int i = 0; i < n && weight <= m; i ++)**

**{**

**max = maxPos(v);**

**value += p[max];**

**x[max] = 1; //将单位价值最大的物品装入分量设置为1**

**weight += w[max] \* x[max];**

**v[max] = 0; //加入背包中的v清零，不参与下一次的求最大值**

**}**

**weight -= w[max];**

**x[max] = (m -weight) / w[max];// 设置最后一个装入物品的装入份量**

**value -= (p[max] / w[max]) \* (1 - x[max]);**

**for (int i = 0; i < n; i ++)**

**{**

**cout<<x[i]<<" ";**

**}**

**cout<<endl;**

**cout<<"最大价值为："<<value<<endl;**

**return 0;**

**}**

* **（2）海面上有一些船需要与陆地进行通信，需要在海岸线上布置一些基站。现将问题抽象为，在x轴上方，给出*N*条船的坐标，，，在x轴上安放的基站可以覆盖半径为d的区域内的所有点，问在x轴上至少要安放几个点才可以将x轴上方的点都覆盖起来。试设计一个算法求解该问题，并分析算法的正确性。**

**Answer:**

1. **算法思想:分别以每个渔船为中心，以 d 为半径画圆，记录与 x 轴交的左 右坐标。每次选择右坐标中最小的值 min，如果某个点的左坐标比 min 小， 则砍掉该点（去掉这个船）。依次执行下去，即可找到最少的基站**

**//找出vector中的最小值下标**

**int minPos(vector<float> temp)**

**{**

**int minPos = 0;**

**for (int i = 1; i < temp.size(); i ++)**

**{**

**if(temp[i] < temp[minPos])**

**minPos = i;**

**}**

**return minPos;**

**}**

**int main()**

**{**

**int n = 6; //n条船**

**int d = 5; //基站能覆盖船的半径**

**vector<float> x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0}; //船的横坐标**

**vector<float> y = {1.0, 2.0, 3.0, 4.0, 4.0, 2.0}; //船的纵坐标**

**vector<float> minX = {0};**

**vector<float> maxX = {0};**

**vector<float> station;**

**//找出所有点与X轴左交点的坐标**

**for (int i = 0; i < n; i ++)**

**{**

**minX[i] = x[i] - sqrt(d \* d - y[i] \* y[i]);**

**}**

**//找出所有点与X轴右交点的坐标**

**for (int i = 0; i < n; i ++)**

**{**

**maxX[i] = x[i] + sqrt(d \* d - y[i] \* y[i]);**

**}**

**for (int i = 0; i < n; i ++)**

**{**

**cout<<minX[i]<<" ";**

**}**

**cout<<endl;**

**for (int i = 0; i < n; i ++)**

**{**

**cout<<maxX[i]<<" ";**

**}**

**int min;//记录右坐标中的最小值**

**for (int i = 0; i < n; i ++)**

**{**

**min = minPos(maxX);**

**for (int j = i + 1; j < n; j ++)**

**{**

**if (minX[j] <= maxX[min]) //如果该点的左坐标小于min点的右坐标，则砍掉该点**

**{**

**minX[j] = -1000.0;//（-1000，1000）表示该点被砍掉**

**maxX[j] = 1000.0;**

**}**

**}**

**}**

**for (int i = 0; i < n; i ++)**

**{**

**if (maxX[i] != 1000)**

**{**

**station.push\_back(maxX[i]);//将maxX中未砍掉的点装入station中**

**}**

**}**

**}**