1、有个大小不同的杯子和与之匹配的*n*个杯盖, 你可以尝试一个杯子和一个杯盖知否匹配, 尝试结果有三种：（1）杯子太大；（2）匹配成功；（3）杯盖太大. 请设计一个分治算法完成所有杯子和杯盖的匹配, 算法的时间复杂性用匹配尝试的次数来衡量。

（1）叙述算法设计思路. (请叙述以下方面)

我们令杯子为a[1], a[2], …, a[n], 杯盖为b[1], b[2], …, b[n]。最左边的左指针记为l，右指针记为r。

边界条件：

标志位(mark)杯子左()右()都没有需要再匹配的杯盖。即l=mark=r。

Divide：

1. 选取一个杯子(例如a[l]),作为参照。首先将a[1]与b[1], b[2], …, b[n]进行匹配，记录匹配成功的位置为mark。在匹配过程中，比a[1]小的杯盖数量记录为count。那么，真实情况下实际a[1]应该是第1+count小的杯子（因为只比count个杯盖大）。所以我们将a[1]和a[1+count]进行调换。接下来我们将b[mark]与b[1+count]交换，这一操作的目的是让第1+count小的杯盖的b[mark]到达其位置上。
2. 此时a[1+count]和b[1+count]是相匹配的。将mark赋值为1+count。我们把比a[mark]小的杯盖移动到b[mark]左边，把比a[mark]大的杯盖移动到b[mark]右边。同理对a中杯子进行同样操作，从而实现类似快速排序的方法。
3. 划分}, }

Conquer：

如果非空，则对于其进行匹配操作；如果非空，同理对于其进行匹配操作。

Merge：

没有merge操作，匹配过程中自动求好了。

（2）写出算法伪代码.

伪代码描述：



C++描述：

#include <bits/stdc++.h>   
   
using namespace std;   
   
int x[6] = {3, 7, 2, 5, 4, 6};   
int y[6] = {2, 5, 4, 6, 7, 3};   
   
void print() {   
 for (int i = 0; i < 6; i++) printf("%d,", x[i]);   
 printf("\n");   
 for (int i = 0; i < 6; i++) printf("%d,", y[i]);   
}   
   
void match(int a[], int b[], int l, int r) {   
 int t = a[l], mark = -1, count = 0;   
 for (int i = l; i <= r; i++) {   
 if (t == b[i]) mark = i;   
 else if (t > b[i]) count++;   
 }   
 swap(a[l + count], a[l]);   
 swap(b[l + count], b[mark]);   
 mark = l + count;   
   
 int i = l, j = r;   
 while (i < mark && j > mark) {   
 while (i < j && a[i] < b[mark]) i++;   
 while (i < j && a[j] > b[mark]) j--;   
 if (i < j) swap(a[i++], a[j--]);   
 }   
   
 i = l;   
 j = r;   
 while (i < mark && j > mark) {   
 while (i < j && b[i] < a[mark]) i++;   
 while (i < j && b[j] > a[mark]) j--;   
 if (i < j) swap(b[i++], b[j--]);   
 }   
   
 if (l < mark) match(a, b, l, mark - 1);   
 if (r > mark) match(a, b, mark + 1, r);   
   
}   
   
int main() {   
 match(x, y, 0, 5);   
 print();   
 return 0;   
}

（3）分析算法的时间复杂性.

时间复杂度和快速排序一样，平均情况下为O(nlogn)，为了减少数据的不确定性，可以每次选择t的时候不用a[l]而是随便选择一个下标作为参考。具体推导如下：

首先，找到mark对应的杯子杯盖只需要扫描一遍数组，复杂度为O(n)。交换、赋值都是常数时间的操作，为O(1)。while循环中执行的操作,i和j的增加至多总共n次，因此while循环内的操作至多执行n次退出，为O(n)，因此除了分治之外，总时间复杂度为O(n)。因此我们有

对于不同情况下的mark，我们计算T(n)的期望为：

因此，我们有

令, 通过累加我们有

和为两个数组，每个数组中的个均已经排好序，试设计一个的分治算法，找出*X*和*Y*中个数的中位数。

（1）叙述算法设计思路.

边界条件：如果X数组和Y数组只有一个元素，直接返回他们俩的平均值。

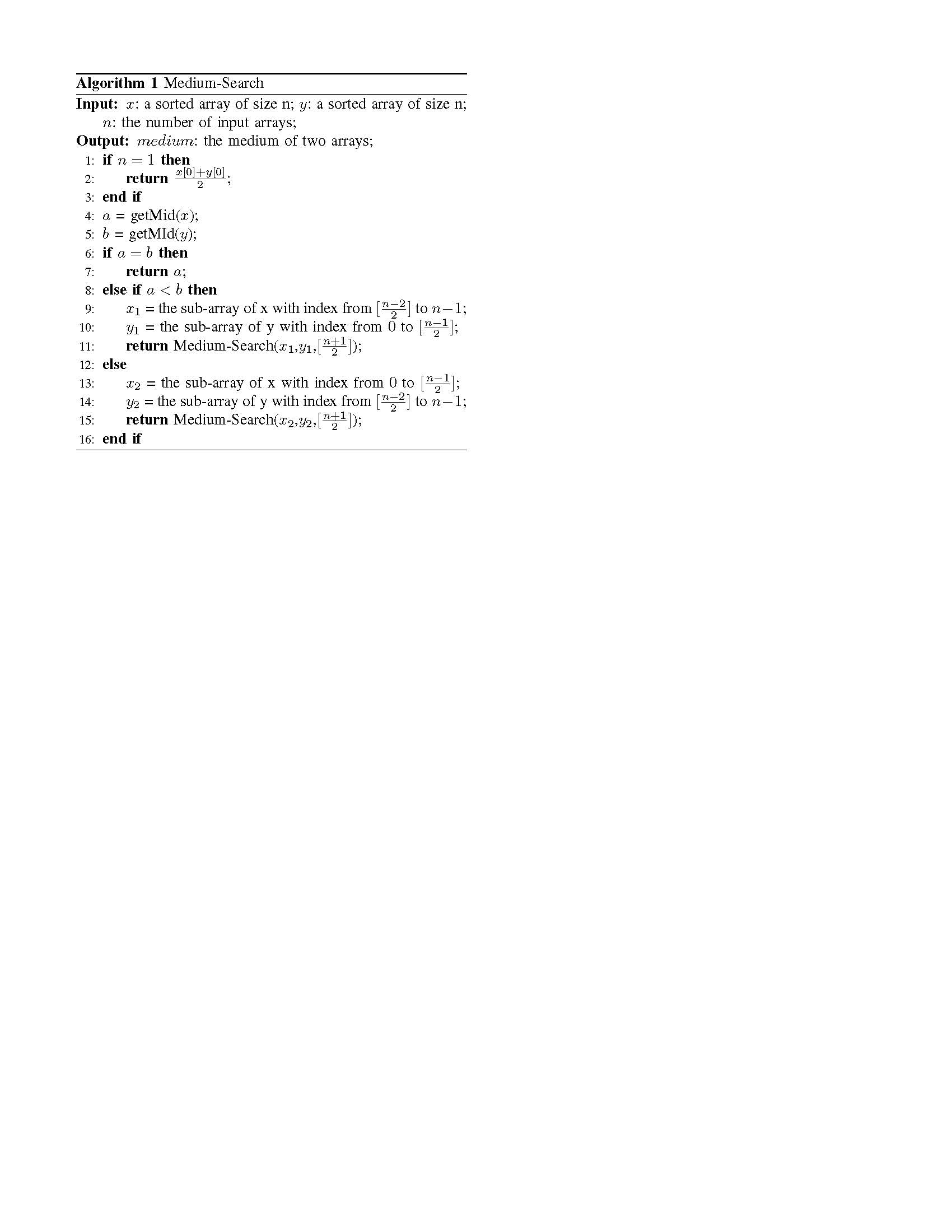
1. 对于X数组和Y数组，我们找出他们各自的中位数a, b。

2. 倘若a=b，那么a之前在X数组的中间，且a在Y数组的中间，进行合并后，a左边的元素为现在a左边元素的两倍，右边同理。因此a或者b即为最终数组的中位数，返回a即可。

3. 倘若a>b，设合并后中位数为mid，如果mid>=a则合并后mid右侧的数<=a在X中右边的数+Y贡献的元素个数<a在X中右边的数+b在Y中右边的数，因此n不为中位数，矛盾！同理可证，mid<=b。于是我们把中位数的搜索范围缩小为了a[0:mid], b[mid:n-1]

4. 倘若a<b，类似3得到缩小的搜索范围为a[mid:n-1], b[0:mid]

（2）写出算法伪代码.



C++描述：

* #include <bits/stdc++.h>   
     
  using namespace std;   
     
  int x[10] = {1, 3, 5, 7, 9, 13, 15, 16, 18, 20};   
  int y[10] = {2, 4, 6, 8, 10, 12, 13, 16, 18, 21};   
     
  double getMedium(int \*a, int n) {   
   if (n % 2 == 0) return (a[n / 2] + a[n / 2 + 1]) / 2.0;   
   return a[n / 2];   
  }   
     
     
  double medium(int \*a, int \*b, int n) {   
  // cout << a[0] << ' ' << b[0] << endl;   
   if (n == 1) return (a[0] + b[0]) / 2.0;   
   double a\_mid = getMedium(a, n);   
   double b\_mid = getMedium(b, n);   
   if (a\_mid == b\_mid) return a\_mid;   
   else if (a\_mid < b\_mid) return medium(a + n / 2, b, (n + 1) / 2);   
   else return medium(a, b + n / 2, (n + 1) / 2);   
     
  }   
     
  int main() {   
   cout << medium(x, y, 10);   
   return 0;   
  }

（3）分析算法的时间复杂性.

由于算法复杂度等同于二分查找，因此时间复杂度为O(logn)。具体推导如下，首先由于数组有序，所以中位数操作是常数时间内完成为O(1). 接着由分治策略得，每次操作之后划归为对于(n+1)/2规模的数组进行中位数操作，因此

由Master定理，T(n) = logn

3、对于两个二维数据元素和，如果(1) , 或者 (2) , ，则支配，记为。二维数据集合的定义如下：。本部分内容计算二维数据集合的。设计基于分治的二维数据Skyline求解算法。

（1）叙述算法设计思路. (请叙述以下方面)

边界条件：当集合为空时，停止操作。

Divide：创建集合left和right，便于划分集合。对于x坐标进行排序，得到x坐标有序的点的序列(x1,y1),(x2,y2),…,(xn,yn)。我们取中间位置

我们对于left和right递归地重复操作。

Conquer：对于left和right我们找到其划分之后剩下地集合l和r，然后对这个两个集合进行merge操作。

Merge：我们扫描right得到最大的y，记录为,与left中的y进行比较，如果left中的元素的y比小，那么这个x一定被的这个点所支配(因为x和y都比这个对应的点所支配)。我们从left删除掉这些点之后，把剩下的点全部加入right中。

（2）写出算法伪代码.



C++代码描述如下:

# include<bits/stdc++.h>   
   
using namespace std;   
   
bool cmpx(pair<int, int> a, pair<int, int> b) {   
 return a.first <= b.first;   
}   
   
bool cmpy(pair<int, int> a, pair<int, int> b) {   
 return a.second <= b.second;   
}   
   
void print(set<pair<int, int>> D) {   
 for (auto i:D) cout << i.first << ',' << i.second << "\n";   
}   
   
int k = 0;   
   
set<pair<int, int>> getSkyline(set<pair<int, int>> D) {   
 if (D.size() <= 1) return D;   
   
 vector<pair<int, int>> elements;   
 set<pair<int, int>> left, right;   
   
 for (auto i:D) {   
 elements.push\_back(i);   
 }   
 //sort   
 sort(elements.begin(), elements.end(), cmpx);   
 int mid = D.size() / 2;   
   
 //left and right   
 for (int i = 0; i < mid; i++) {   
 left.insert(elements[i]);   
 }   
   
 for (int i = mid; i < D.size(); i++) {   
 right.insert(elements[i]);   
 }   
   
 //divide   
 left = getSkyline(left);   
 right = getSkyline(right);   
   
 pair<int, int> p = {-1, -1};   
 // find maxy in the right   
 for (auto i:right) {   
 if (cmpy(p, i)) p = i;   
 }   
 // earse some of the left   
 for (auto i:left) {   
 if (cmpy(i, p)) left.erase(i);   
 }   
 D.clear();   
 set\_union(left.begin(), left.end(), right.begin(), right.end(), inserter(D, D.begin()));   
   
 return D;   
}   
   
int main() {   
 set<pair<int, int>> D = {{1, 2},   
 {2, 3},   
 {4, 2},   
 {3, 5},   
 {5, 4}};   
 D = getSkyline(D);   
 print(D);   
 return 0;   
}

（3）分析算法的时间复杂性.

对于x坐标的排序算法，复杂度为O(nlogn)。对于分治算法，合并集合的复杂度为O(n).因此我们有如下的公式：

根据master定理，我们得到T(n)=O(nlogn)

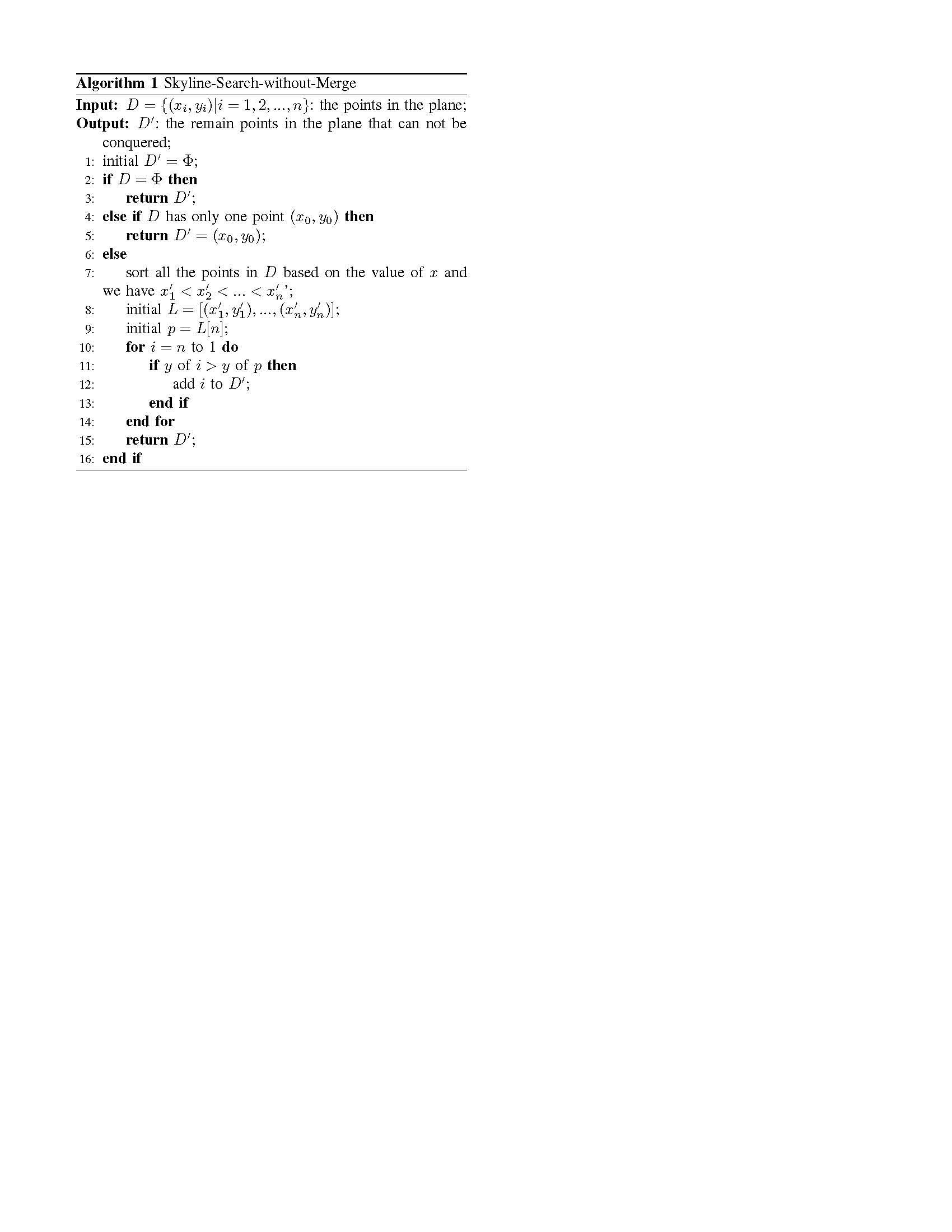
PS:

一开始我考虑的方法没有用到分治策略，算法大致描述如下：

1. 首先类似上面对x进行排序。
2. 接着我们按照x从大到小进行扫描，令x最大的点为p。如果扫描的点的y值大于p的y值(当前的)，那么这个点一定属于最终的集合，因为在它之上只有p比这个点x大，但y却比p大。反之这个点不属于最终的集合，不需要加入。我们记录保留下来的点的新的y值，然后每次遇到一个新的点，如果y值比它小就不需要加入这个点，反之加入这个点并且更新值

由于扫描一遍只需要O(n)，另外排序是O(nlogn),因此总体复杂度为O(nlogn)。

**伪代码描述：**



C++描述：

# include<bits/stdc++.h>   
   
using namespace std;   
#define N 1000007   
int n;   
typedef pair<int, int> point;   
point ans[N];   
point p[N];   
   
bool cmp(point a, point b) {   
 return a.first > b.first;   
}   
   
int main() {   
 int n;   
 scanf("%d", &n);   
 for (int i = 0; i < n; i++) {   
 scanf("%d%d", &p[i].first, &p[i].second);   
 }   
 int num = 0;   
 sort(p, p + n, cmp);   
 int max\_height = -1;   
 for (int i = 0; i < n; i++) {   
 if (p[i].second > max\_height) {   
 max\_height = p[i].second;   
 ans[num].first = p[i].first;   
 ans[num].second = p[i].second;   
 num++;   
 }   
 }   
 for (int i = num - 1; i >= 0; i--) {   
 printf("%d %d\n", ans[i].first, ans[i].second);   
 }   
 return 0;   
}