***2019***



**算法设计与分析实验报告**



|  |  |
| --- | --- |
| 专 业： | 计算机科学与技术 |
| 班 级： | 计算机科学与技术1707 |
| 学 号： | U201714807 |
| 姓 名： | 李俊欣 |
| 完成日期： | 2019.12.14 |

|  |
| --- |
| 一、原创性声明 |
| 本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。  特此声明！  **作者签字:** |

|  |
| --- |
| 二、对课程设计的学术评语（教师填写） |
|  |
| 三、对课程设计的评分（教师填写） |
| |  |  |  |  | | --- | --- | --- | --- | | 评分项目  （分值） | 报告撰写  （50分） | 课设过程  （50分） | 最终评定  （100分） | | 得分 |  |  |  | |
| **指导教师签字:** |

**目 录**

[**1.完成情况** 1](#_Toc27345689)

[**2.解题报告** 1](#_Toc27345690)

[2.1 poj3295 Tautology解题报告 1](#_Toc27345691)

[2.2 poj3714 Raid解题报告 4](#_Toc27345692)

[2.3 poj1723 Soldiers解题报告 8](#_Toc27345693)

[2.4 poj2228 Naptime解题报告 10](#_Toc27345694)

[**3.总结** 13](#_Toc27345695)

[3.1 已经完成的工作 13](#_Toc27345696)

[3.2 进一步需要完善的地方 13](#_Toc27345697)

[3.3 心得体会 13](#_Toc27345698)

# 

# **1.完成情况**

整个算法实验，八节课，八个课时，我一共完成了16道poj上的题，因为有多次试探提交答案，所以提交一共26次，AC16次,成功提交的题号和题数如图1-1所示。



图1-1 AC情况

# **2.解题报告**

整个实验AC16题，我将选择四道比较经典的题从目的分析、算法的设计、性能的分析和运行结果的展示四个方面进行详述。另外对没有AC的题目及原因在总结中分析说明。

## 2.1 poj3295 Tautology解题报告

（1）目的分析

Poj3295这道题的题目要求是输入一串有特定意义的合法字符串，根据一定的规则判断是否能成为永真式。

其中：\*输入由 p、q、r、s、t、K、A、N、C、E 共 10 个字母组成的逻辑表达式，

\*其中 p、q、r、s、t 的值为 1（true）或 0（false），即逻辑变量；

\*K、A、N、C、E 为逻辑运算符，

\*K --> and: x && y

\*A --> or: x || y

\*N --> not : ! x

\*C --> implies : (!x)||y

\*E --> equals : x==y

\*问这个逻辑表达式是否为永真式。

\*PS:输入格式保证是合法的

例如：Input：ApNp

ApNq

0

Output：tautology

not

输出分析：因为不论p的逻辑值为1或者0，ApNp所代表的p||（！p）都将是逻辑真1，所以该式是一个重言式；而对于ApNq，可以举出反例，当p为假，

q为真的时候，该式为假0，所以不是一个重言式。

所以可以很明显地判断出来，这道题重点和难点是对两种输入的判断处理，即运算符和逻辑元。

（2）算法的设计

根据上述目的分析，对题目算法进行设计。

根据题意，可以首先想到两种解题方法:递归和栈方法。

①递归（AC方法）

因为只有p、q、r、s、t是作为逻辑单元，所以可以将他们别人看成一位二进制数，有0、1两个状态，所以用一个五位的二进制数S来代替每一个字母，即tsrqp，例如00001代表只有p为1其他逻辑元都为0。因此要判断是否为一个重言式，只需要例举出所有的二进制数S可能的取值即可，如果都满足，则为重言式，只要有一个不满足，则直接判断为非重言式。

而将输入的字符串保存在str[101]的数组中，每层递归读取一个字符，，根据读出的字符作相应的逻辑处理。主要分两类：逻辑元和逻辑运算。逻辑元按照S移位取相应的0、1作为值即可，而逻辑运算为了得出运算元的值则做下一层的递归运算调用，直到得出值为止。例如：对于K，则是return process(str,k)&process(str,k)，其中对取出的字符为这一层的下一个。其他运算符相应有A: return process(str, k)|process(str, k)、N：return !process(str, k)、C: return (!process(str, k))|process(str, k)、E：return process(str, k) = = process(str, k)，逻辑如图1-2所示。



图1-2 递归调用举例逻辑图

值得注意的是，我们在例举32中情况的中间，只要出现了不满足的情况就是违背了重言式的条件，所以这时直接判错。关键代码如下：

bool judge(char str[101])

{

for (int k = 0; k < 32; k++) {

n = -1;

if (!process(str, k))

return false;

}

return true;

}

②队列方法

队列是一种很特别的数据结构，因为有着先进先出的逻辑所以常常用来处

理字符串匹配和算式处理。根据题意，我们需要处理的字符可以分为两类，即逻辑元和运算符。由于不同的运算符需要取出不同数量的逻辑元运算，所以用两个栈来对它们进行分别保存是比较理想的做法。

我们设队列Q用来存储逻辑元和运算符。每当输入任意一个逻辑元（p、q、r、s、t）或者任意一个运算符（K、A、N、C、E），我们将其入栈Stack。根据运算规则我们知道，可以从运算式的最右边进行运算，每次取出一个运算符，根据运算符的类型来进一步取逻辑元的个数。例如当从Q中取出一个逻辑元后，又取出了N（非运算），则组成了一个新的逻辑元，保存到一个特定的bool变量1中，再取出下一个栈内容，如果为逻辑元，则保存到另一个特定的bool变量2中，当再出栈的一个为运算符的时候，例如K（或），则将bool变量1和2中的逻辑元作与运算，结果保存到bool变量1中，继续做相应操作。逻辑如图1-3所示。

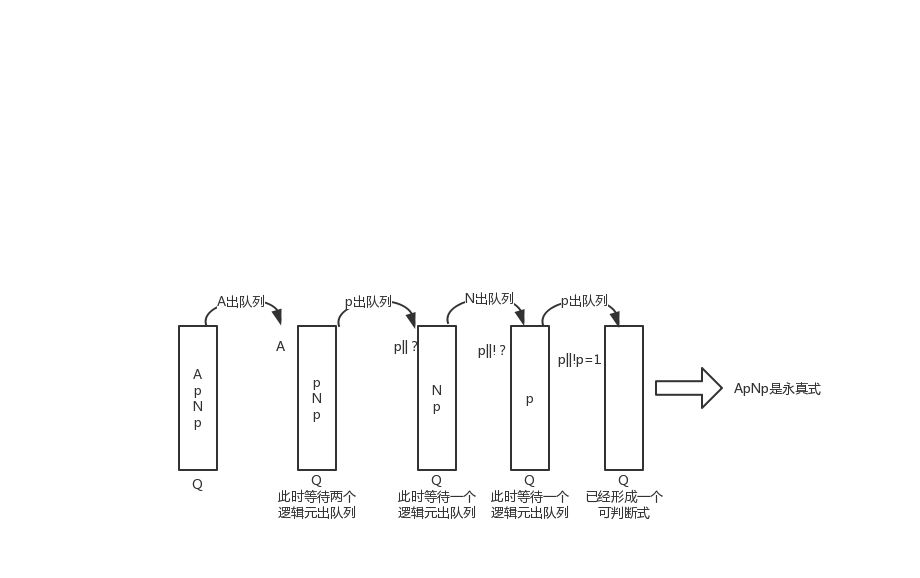


图1-3 队列思想逻辑图

（3）性能的分析

由于，AC的代码是上述分析的第一种递归方法，在这里只做递归方法的性能分析。具体用时和空间使用如图1-4所示。

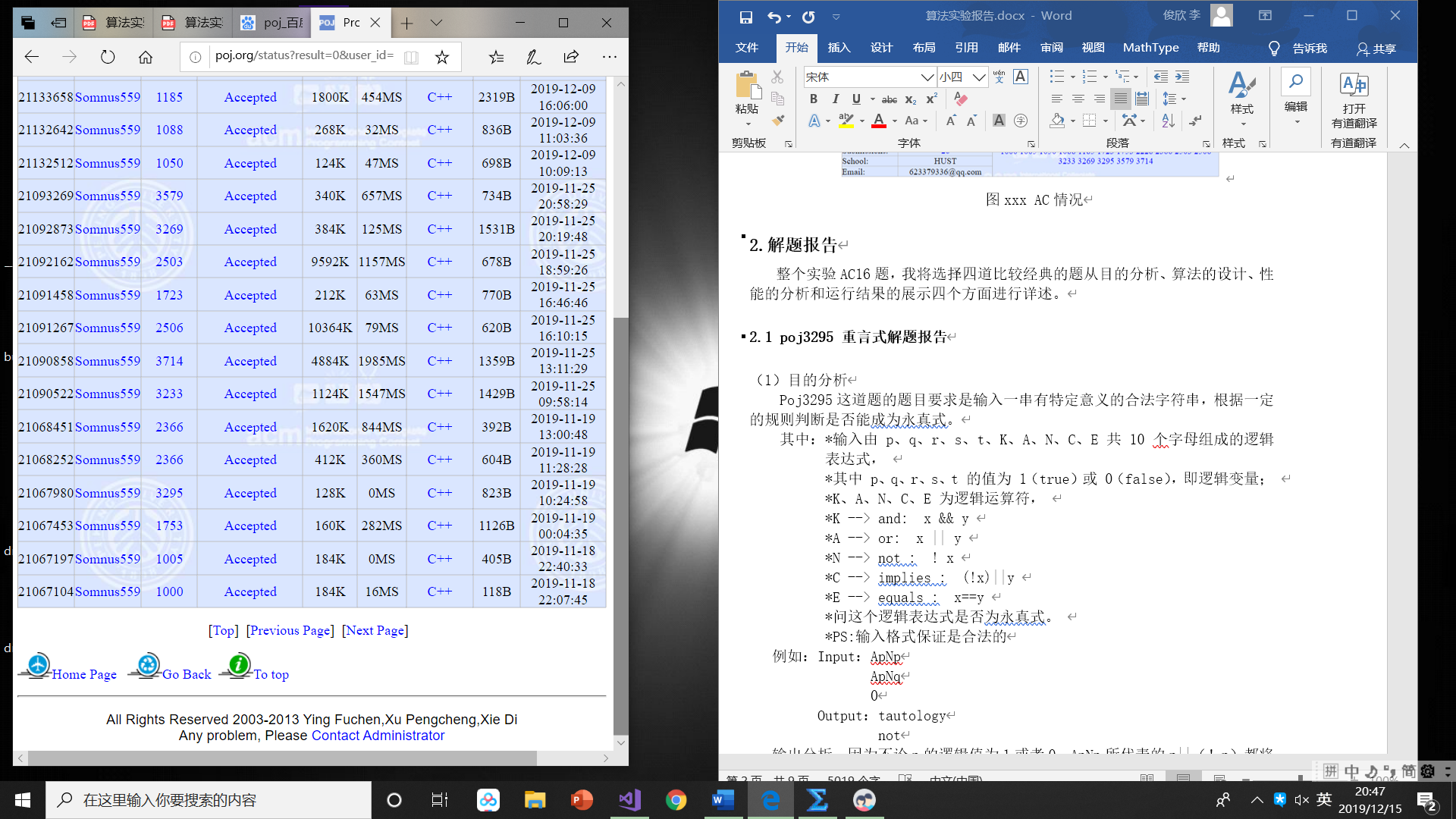


图1-4 3295AC时间和空间使用情况

可以看见，虽然使用了递归，但是空间使用只有128K还是比较小的，而时间更占优势，没有达到1Ms，所以这里没有表示出来。下面来具体分析。

空间复杂度：外部使用一个char型数组str进行存储，由于运算符最多是双目运算，因此，在考虑到有重复N运算的情况，递归的深度最多为n，而一般情况下，递归的深度只有3，即一个双目运算符加两层逻辑元递归运算。空间复杂度也就只有O（n）。

时间复杂度：因为需要对输入字符串中每一个字符进行处理，每一层递归调用的时间为O（1），所以总体的时间复杂度只是O（n）。

根据上述分析可知，递归方法的时间复杂度和空间复杂度一样，但是相对于一般算法来说，虽然时间复杂度不高，但是递归的空间复杂度太高，由于这道题的特殊性，其递归深度不会很大，所以还好。但是一般算法题不建议使用递归，

容易造成溢出。

（4）运行结果的展示

根据上面的分析，写好代码后运行测试。如图1-5所示，跟样例输出一样，结果正确。

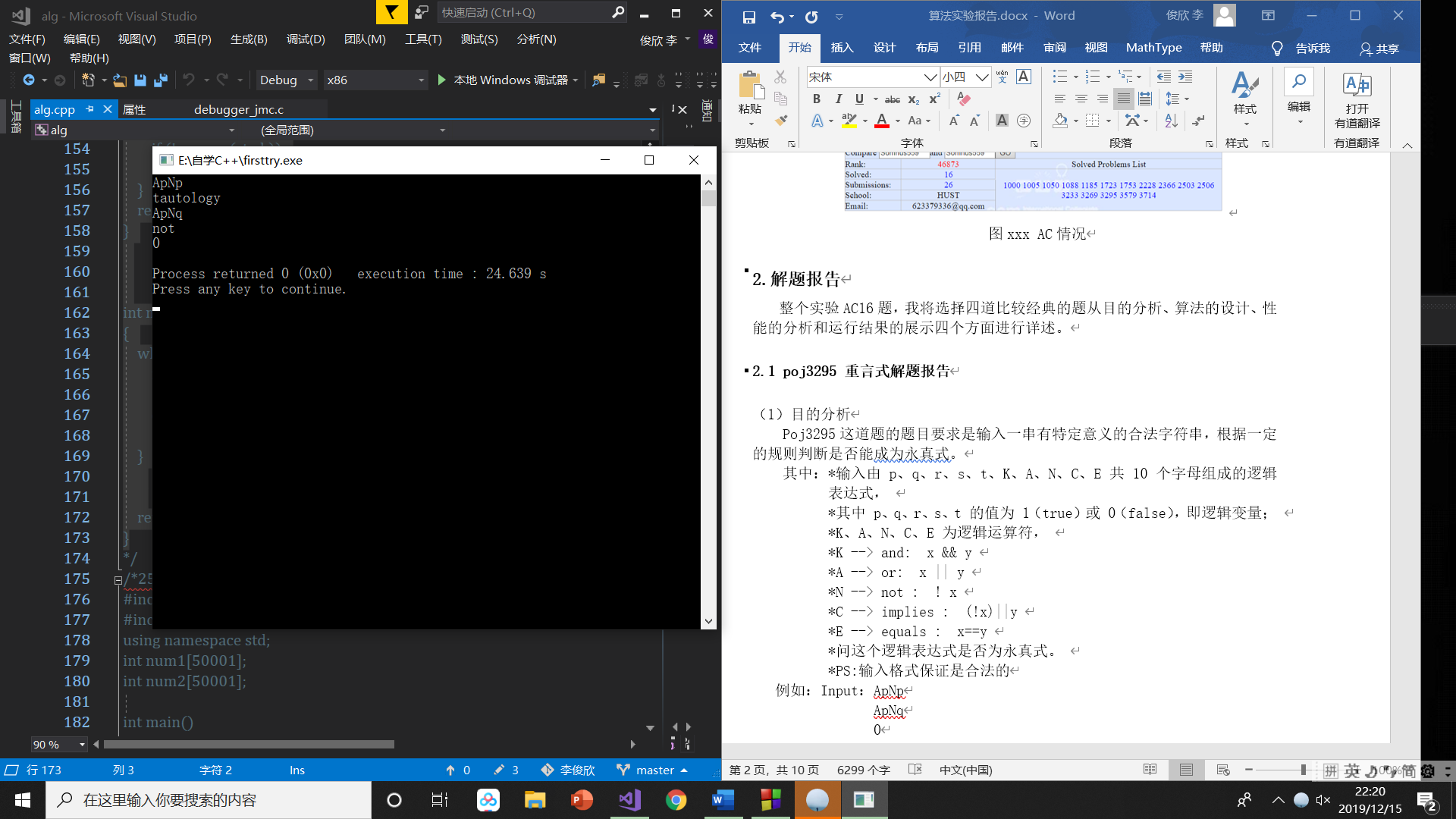


图1-5 测试结果展示

## 2.2 poj3714 Raid解题报告

（1）目的分析

Poj3714这道题的题目要求是在一组士兵和发电站中找到一组士兵和发电站之间的距离最短。

例如：Input： 4

0 0

0 1

1 0

1 1

2 2

2 3

3 2

3 3

Output：1.414

由于最短的士兵和发电站的两点是（1，1）和（2，2），所以欧氏距离就是根号二，保留三位小数为1.414。

所以弄清楚题意了之后我们可以发现，这道题实际上就是为了让我们能得到两个最优化点即可，难点就在于这些点是两个种类，需要分开处理。

（2）算法的设计

①遍历计算

根据上述的目的分析，我们很容易可以想到的是最简单的变量方法，将每一个士兵和每一个发电站一一组合进行求解，最后不断更新得到最优化的结果。这样的算法的确简单，但是让我们来看看可行性。当输入的士兵和发电站的数量分别是n的时候，我们需要运算次来遍历所有的情况，从而得出最近的欧式距离，所以时间复杂度为。在测试用例N= 100000的时候，其运算复

杂度是比较恐怖的，所以我们尽量在做算法题的时候不要采用直接遍历的这种方法。

②二分法加分治

在对遍历算法尽心分析之后，我们认为那样子计算很容易超时，所以我们提出另一种解题思路，即二分法加分治。

上面的遍历中我们将点的处理分成了两个部分，是因为那种方法只是基于代数计算的，只算必要的距离。但是在二分法加分治中，我们决定在几何空间中进行。我们如果将空间上的点分成两类，将会增加处理的难度，所以将是士兵或者发电站这个属性只在计算欧氏距离之前时给出加以区分。具体说来，就是将所有的点放在几何空间上，不区分士兵和发电站，当找到两个点满足一定条件被选中计算之前再判断筛除同类的点。

使用分治二分法计算，首先需要根据X的坐标对所有的点进行排序，为了二分的进行。然后我们将空间按照X轴分成两个部分，满足最优的两个点的分布只能是全在左边，全在右边以及跨越中间的左右两点，三种情况中最优的一组。左边和右边的最优点距比较简单，只用再次二分递归按照同样的方法计算即可。我们先标记d为左右情况中最近的两点d1、d2的距离。接下来计算横跨中间的情况，这里我们有一步处理来简化计算，即在左边和右边分别找一个点的时候，两个点到中线的距离不能超过d，这一点也很好理解，因为如果大于了很容易判断出来不是一个最优的距离。所以经过这样的筛选，在左右中分别选择点组合，算出距离，更新最优值即可。逻辑图如图1-6所示。

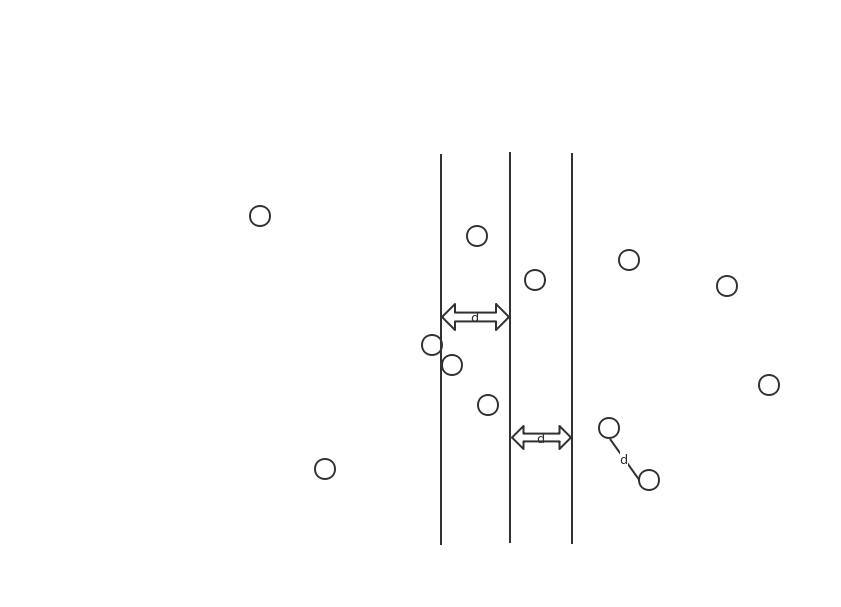


图1-6 分治、二分法处理逻辑图

对图1-6解释的关键代码如下：

double solve(const int& l, const int& r) {

if (l == r)

return INF;

int mid = (l + r) >> 1;

// 计算左边和右边的最近点对的距离

double a = solve(l, mid);

double b = solve(mid + 1, r);

// 取较小值d

double d = min(a, b);

// 计算横跨中间线的最近点对

for (int i = mid; i >= l; i--) {

if (no[mid].x - no[i].x > d)

break;

// 找到一个距离中间线小于d的左边的点，再遍历寻找右边的距离中间线小于d的点

for (int j = mid + 1; j <= r; ++j) {

// 如果右边的某个点距离中间大于d，则它更右边的点都不要了

if (no[j].x - no[i].x > d)

break;

// 如果id相同，跳过

if(no[j].id == no[i].id)

continue;

double tmp = dis(i, j);

if (no[i].id != no[j].id && tmp < d)

d = tmp;

}

}

return d;

}

总结一下我们计算时优化的筛选方法：1.左右点到中线的距离超过一组点都位于左边或者右边的情况d，舍弃；2.如果点的属性相同，我们舍弃。

（3）性能的分析

具体用时和空间使用如图1-7所示。

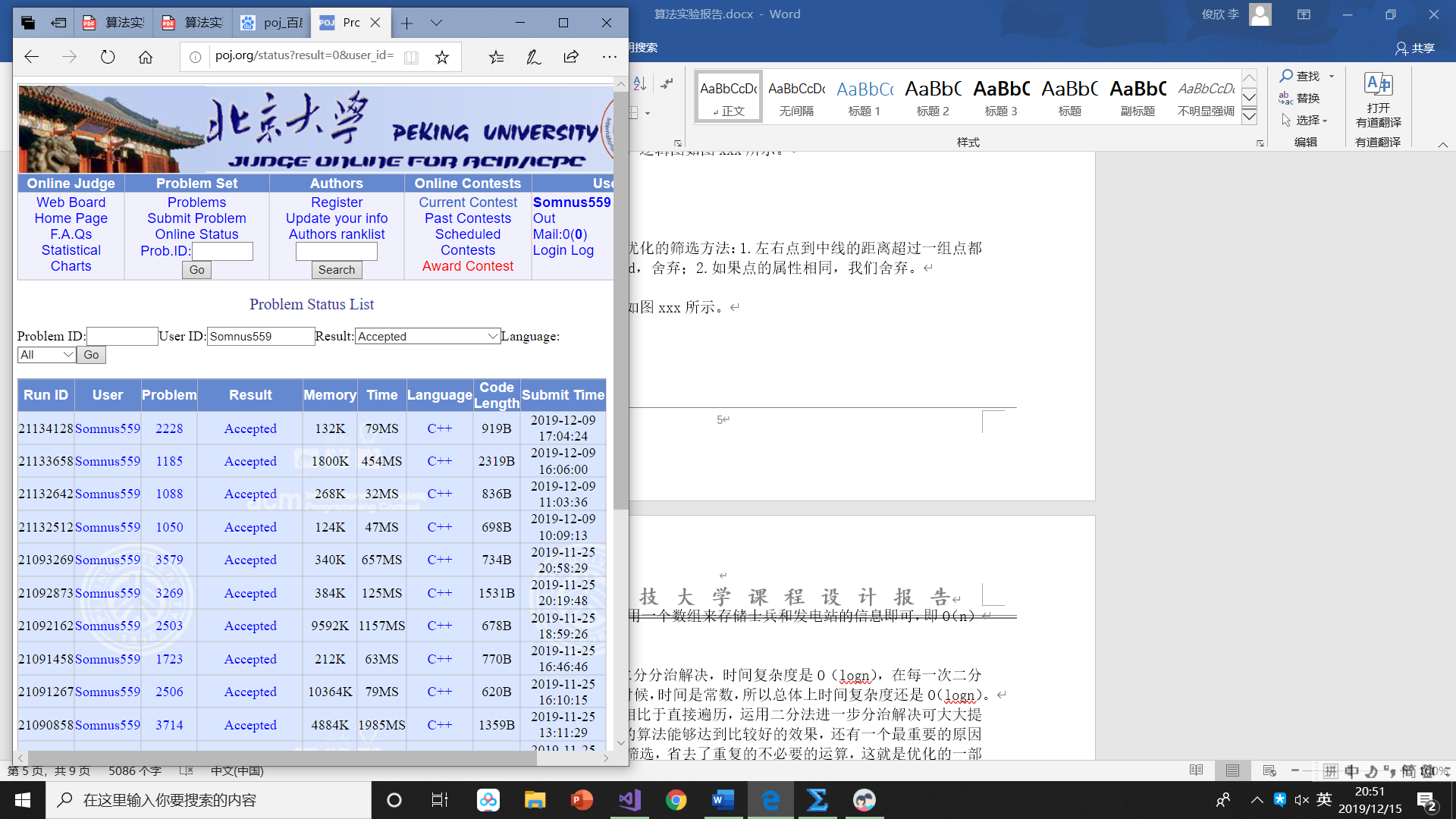


图1-7 3214AC时间和空间使用情况

可以看到空间使用了4884k,估计由于测试的使用样例比较大，时间使用了1985Ms（更加说明了如果使用遍历会造成更恐怖的超时）。下面来具体分析。

空间复杂度： 只需要用一个数组来存储士兵和发电站的信息即可，即O（n）

的空间复杂度。

时间复杂度：使用二分分治解决，时间复杂度是O（logn），在每一次二分合并的时候，时间是常数，所以总体上时间复杂度还是O（logn）。

根据上述分析可知，相比于直接遍历，运用二分法进一步分治解决可大大提高运算速度。之所以我们的算法能够达到比较好的效果，还有一个最重要的原因就是我们上述使用的点的筛选，省去了重复的不必要的运算，这就是优化的一部分。

（4）运行结果的展示

根据上面的分析，写好代码后运行测试。如图1-8所示，跟样例输出一样，结果正确。

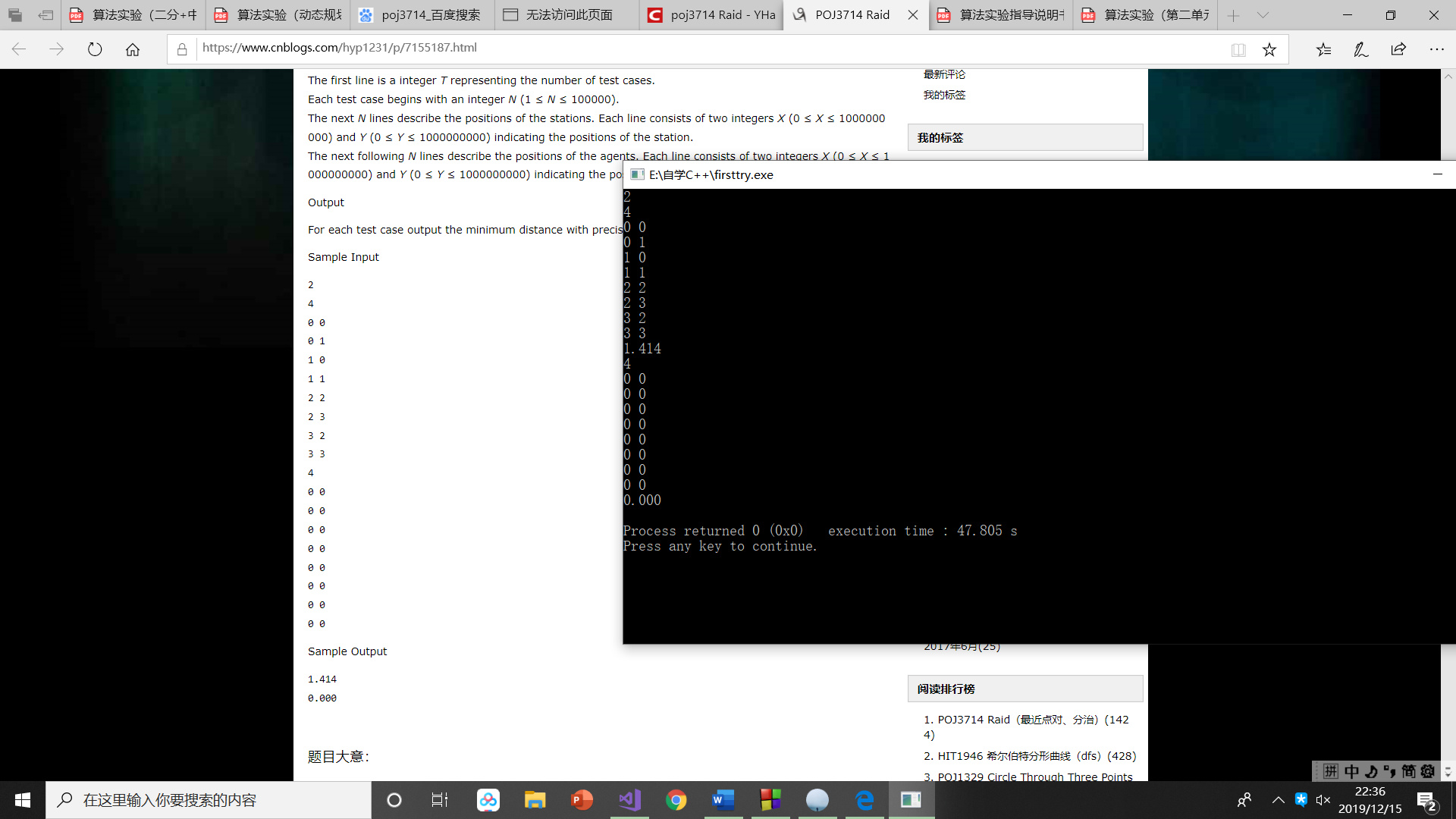


图1-8 测试结果展示

## 

## 2.3 poj1723 Soldiers解题报告

（1）目的分析

Poj1723这道题的题目要求是要将一组士兵能排到一排，使他们移动的折线距离最短（并不是欧氏距离），输出这个最短的距离和。

例如：Input：5

1 2

2 2

1 3

3 -2

3 3

Output:

8

我们可以发现，这道题是想要我们确定一个点即可，即在整个队伍中人每一个士兵的最终最优位置即可。最后将所有移动的距离加起来就是我们想要的答案。

（2）算法的设计

初始一看这个题，会觉得比较混乱，不过我们比较清楚的是当求移动距离最小的题的时候，我们常用的一种方法就是找中位数。这道题并不难，在于它求解距离的种类和欧式距离完全不同。欧氏距离的求解是和x、y都有关的，即x、y是捆绑的元素，而这里的折线距离则完全不同，因为我们的求解是x方向上的

移动和y方向上移动的和，所以最简单的方法就是我们将两个方向分开考虑，实

质上是互不影响的。

因为要排成一排，所以所有士兵的y要移动到相同的y’上，我们将y轴进行排序，可以知道，要想移动得最少就应当取中位数，于是y方向上的问题就很容易被解决了。对于x方向，我们需要注意的是排队时的连续性，所以所有士兵的最终位置在x方向上有一个关联关系就是其连续性。我们设第一位在x=a上，那么很容易知道，第二位、第三位…分别在a+1，a+2…上，所以我们先将士兵们

按照x轴递增的顺序进行排序，得到X1、X2、X3…,按照上述所说，就应该有X1 = a，X2 = a+1，X3 = a+2…,现在我们要决定在a为何值的时候，能使最小。将之前的关系式变形得到X1 = a，X2 -1= a，X3 -2= a，所以我们转变成了求X1，X2-1，X3-2…和在y轴上相同的问题了，因此只需要求出X1，X2-1，X3-2…的中位数即可。即X’= { X1，X2-1，X3-2…},Y = {Y1,Y2,Y3…}，那么最短的额距离就应该是。关键代码如下所示：

sort(x, x + n);

sort(y, y + n);

int middle = (n + 1) / 2 - 1;

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

{

x[i] -= i;//形成X’

}

sort(x, x + n);

int length = 0;

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

{

length += (abs(x[i] - x[middle]) + abs(y[i] - y[middle]));//找到最佳

}

（3）性能的分析

具体用时和空间使用如图1-9所示。

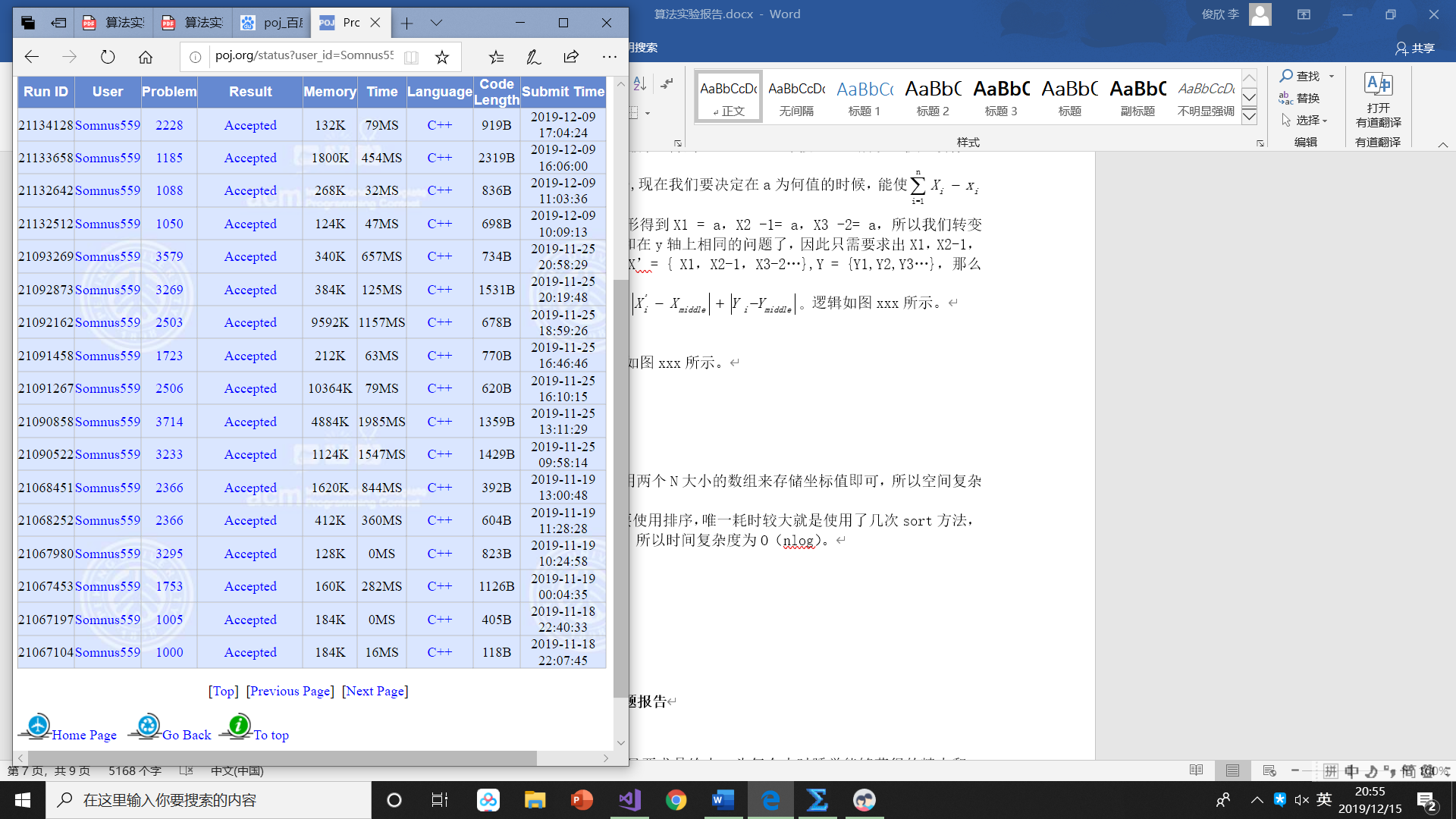


图1-9 1723AC时间和空间使用情况

可以看到空间使用了1723k，时间仅使用了63Ms，算法效果是比较好的。下面来具体分析。

空间复杂度：外部使用两个N大小的数组来存储坐标值即可，所以空间复杂度为O（n）；

时间复杂度：因为需要使用排序，唯一耗时较大就是使用了几次sort方法，时间复杂度为O（nlogn），所以时间复杂度为O（nlog）。

（4）运行结果的展示

根据上面的分析，写好代码后运行测试。如图1-10所示，跟样例输出一样，结果正确。

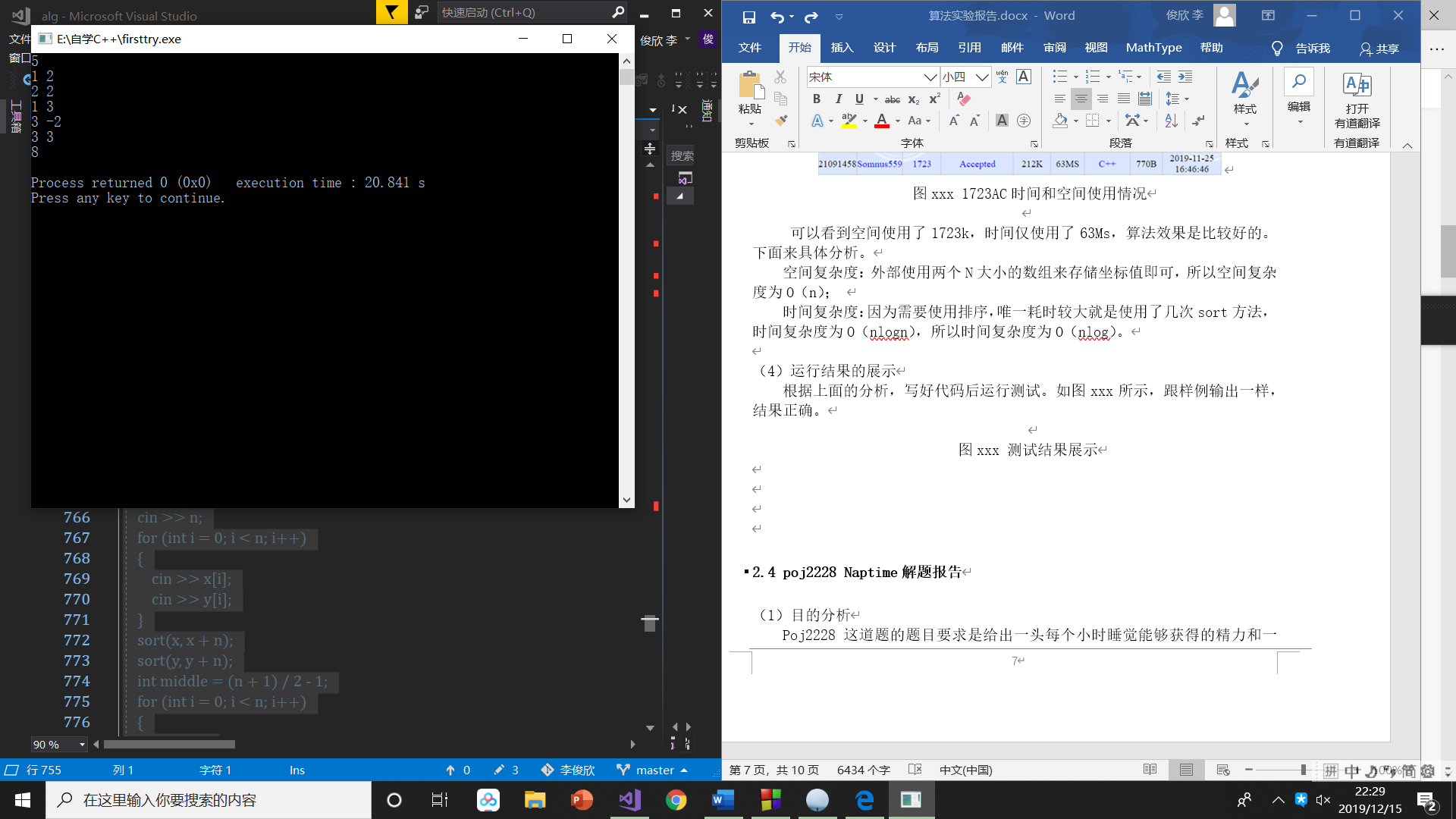


图1-10 测试结果展示

## 2.4 poj2228 Naptime解题报告

（1）目的分析

Poj2228这道题的题目要求是给出一头每个小时睡觉能够获得的精力和一天内（一天被分成了N个时间段）睡觉时间的上限，当连续睡觉的第一个小时精力不计算的时候求一天内（N个时间段，可以是前一天的尾加上后一天的头）最多能够休息得到的精力。

例如：Input：5 3

2

0

3

1

4

Output：

6

根据输入可知，一天被分成了5个时间段，一天最多能睡3个时间段，所以

应该是一天的最后两个时间段和后一天的最开始的一个时间段用来睡觉最优，一共可以得到2+4 = 6的精力。此题的目的就是要我们在时间有限的情况下选择最好的安排能够得到最多的收获。

（2）算法的设计

经过上面的目的分析，因为时间的连续性，很明显这是一道动态规划题。我们注意到这个算法题中需要注意的两个点：1.开始睡觉的第一个时间段不计算；2.一天的时间可以是跨越两天的。为了睡得尽量多，我们睡觉的总时间肯定是用完，即B，而对于睡觉的精力收获存在一个递推关系。

我们设前i个时间段一共睡了j个时间段，而且在第i个时间段得到精力最大，我们可以将这种描述再细分为，最后一个时间段睡了或者没睡，用f[i][j][0/1]表示。所以我们可以得到如下的递推关系式：

 ……①

 ……②

递推式①表示前i个小时一共睡了j个时间段，而且在第i个时间段睡了的情况下得到精力最大，总精力应该是前i-1个时间段一共睡了j-1个小时，而且在第i-1个时间段没有睡的情况下得到精力（其实第i+1时间段也睡了，但是由于是睡觉的第一个时间段，所以不计）或者前i-1个时间段一共睡了j-1个时间段，而且在第i-1个时间段睡的情况下得到精力加上第i个时间段可获得的精力，两者中较大者。

递推式②表示前i个小时一共睡了j个时间段，而且在第i个时间段没睡的情况下得到精力最大，总精力应该是前i-1个时间段一共睡了j个小时，而且在第i个时间段没有睡的情况下得到精力或者前i-1个时间段一共睡了j个时间段，而且在第i个时间段睡的情况下得到精力加上第i个时间段可获得的精力，两者中较大者。

有了上面的两个递推关系式，我们的算法题就算是解了一半。下面我们分成两个情况来考虑，牛只在同一天休息和牛跨越两天休息。两者的区别仅仅是对于最后一个时间段的处理不同以及初始化的条件不同。逻辑如图1-11所示。

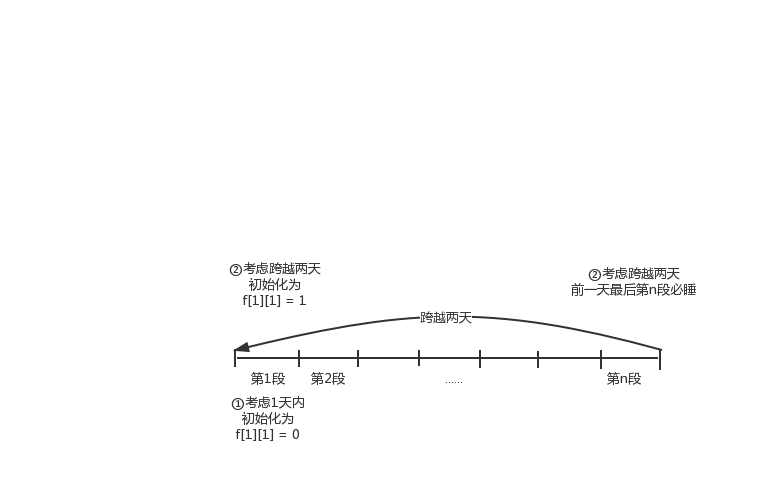


图1-11 线性和环形时间处理差别

当考虑线性的一天时，设f[0][0] = f[1][1] = 0,我们用循环解释i考虑值，从而在式子中省去。因为线性的一天中，第一个时间段一定不会被计算（作为第一个休息的时间段，不计算进去），所以初始化为0。用两层循环，i的值从

2到N，j的值从i和b中较小者到1，由上面的递推式，可以得出ans为睡了b个小时，最后一小时睡或者不睡的最大值。关键代码如下：  
 mem(f); f[0][0] = f[1][1] = 0;//因为第一天一定不会计算进去，所以初始值设为0

go(i, 2, n) back(j, min(i, b), 1) {//线性的一天，一天的第一个小时一定没有不会计算进

去

f[j][0] = max(f[j][0], f[j][1]);

f[j][1] = max(f[j - 1][0], f[j - 1][1] + u[i]);

}

ans = max(f[b][1], f[b][0]);

再考虑成环的一天，若要跨越两天，那么一天中的第一个时间段一定是要睡

觉的，而且前一天的最后一个小时必须睡，初始条件改编为f[1][1] = u[1]；最后只取f[b][1]即可。最后的最大精力就是线性的一天或者跨越的两天中较大者。关键代码如下：

mem(f); f[1][1] = u[1];//考虑两天成环的情况，所以第二天的第一个小时让它强行睡觉

go(i, 2, n) back(j, min(i, b), 1) {//两个限制条件，j<i,j<b,前i个小时睡j个小时，不能超过题目的要求b

f[j][0] = max(f[j][0], f[j][1]);

f[j][1] = max(f[j - 1][0], f[j - 1][1] + u[i]);

}

ans = max(ans, f[b][1]);//因为最后一个小时必须睡，第二天的第一个小时才能睡，不考 虑f[b][0]的情况

（3）性能的分析

具体用时和空间使用如图1-12所示。

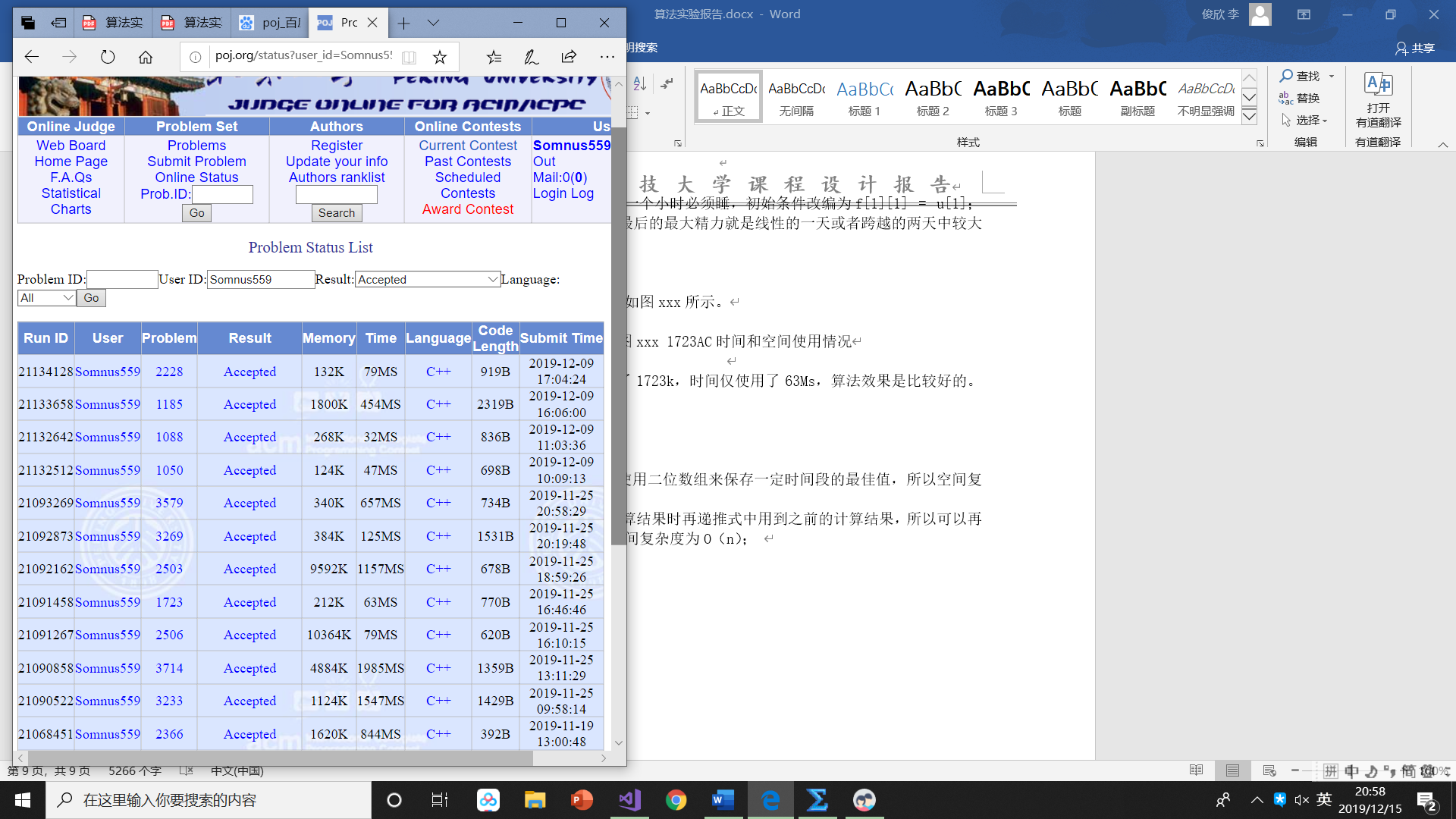


图1-12 2228AC时间和空间使用情况

可以看到空间使用了132k，时间仅使用了79Ms，算法效果是比较好的。下面来具体分析。

空间复杂度： 外部使用二位数组来保存一定时间段的最佳值，所以空间复杂度为O（NB）；

时间复杂度：每次计算结果时再递推式中用到之前的计算结果，所以可以再线性时间里完成计算，时间复杂度为O（n）；

（4）运行结果的展示

根据上面的分析，写好代码后运行测试。如图1-13所示，跟样例输出一样，结果正确。

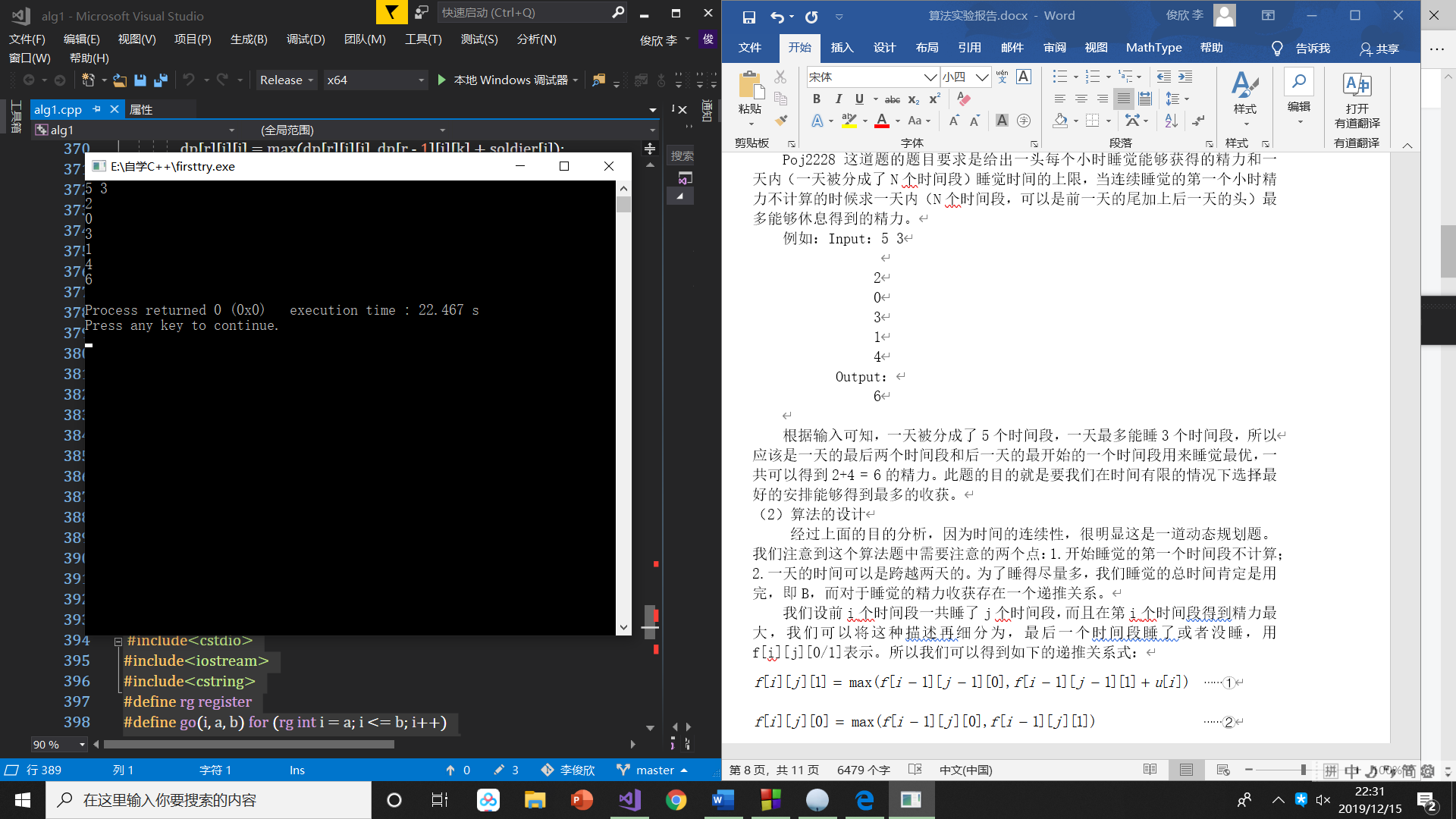


图1-13 测试结果展示

# **3.总结**

## 3.1 已经完成的工作

将算法课上8个课时以及课下完成的总结如下：

1.16道算法题，并且成功在网站上AC；

2.整理16道题的解题思路，并详述了四题AC题目；

3.总结算法题的技巧和方法以及一开始没能AC题目的原因。

## 3.2 进一步需要完善的地方

在实验的过程中，很多次写完代码后发现能运行正确，但是提交之后报错超时，这种情况我多半会直接整体换思路和算法，而没有想过局部优化，所以后续需要对没有AC的一些代码进行进一步优化使其能过通过测试。

## 3.3 心得体会

在此次的算法实验中，16道题完成得比较曲折，但是确确实实在其中学到了很多知识。发现，并不是只有编程和算法知识就能够解决所有问题的，下面首先对我学习到的一些解题技巧做一个总结：

1.解题时所有情况的例举：在poj1753和poj3295中都是用到了考虑所有可能存在的情况，并且用一个二进制数每一位代表的1或0来表示不同的意义，这种解题方法，很方便地将一些字符或者二维空间转换成了一维的甚至是点的情况，极大地简化了我们的解题思路；

2.合理利用递归：递归是一种较为简单的解题方法，在有些算法题中，递归

可以简化我们的逻辑，但是更多情况下，我们需要根据题的具体情况考虑是否使

用递归，因为其递归深度常常可能导致内存溢出。在这次的算法题中，poj1753和poj3295我都使用到了递归，而且效果较好；

3.挑选最短距离的两点问题：不再第一时间考虑以前简单的遍历，在学习了二分和分治之后，这应该是最好的最理想的方法了，将大的问题化成相同规则的小问题；

4.善用排序：在很多的算法题中，我们都需要借助排序来解决问题，例如poj1732，因为使用了排序，而能有效利用中位数的性质来解题。在poj3714中也是运用到了排序，再使用二分、分治思想的。在poj3269中，使用排序后，找到中位数，来确定最佳的牛舍位置及个数。可见，在做算法题时当题中元素比骄傲混乱但是又有空间性的时候，不如考虑一下排序，说不定思路可以打开；

5.中间值的保存：这个思想主要是运用在递推式和动态规划中。在递推中我们就是运用了上一步已有的计算结果，简化了此步的运算。在动态规划中，例如poj1088、poj1185、poj1050的解题思路中都是用到了外部的空间来存储中间值。

6. string耗时太长，很容易超时：在处理已知长度或者形式的字符的时候，如果种类不是很多，可以哈希转化成数字存储每一种结构结果；

7. 结果的提前存储：当答案为一一对应的时候，且种类不多时，最好先考虑每一种结果并存储，然后进行结果查找，时间会更快。例如选做题poj2506即shi 如此。

另外，在实验的过程中有很多次没有AC成功，例如poj2366一开始没有使用二分查找，而是直接一一比较，线性查找，结果超时提交失败，最后改成二分查找结果成功。再如，poj2503这道题的输出一直有问题，因为在结束输入的判断没有使用~scanf\_s("%s", word)，而是使用了读EOF所以样例过不了，最后修改之后成功AC。

以上就是我此次实验的学习收获，感受到了算法不能只是究于理论，还是要实操才能有进步。最后，感谢实验中老师和助教的耐心解答和同学们的热情帮助。