***2021***



**算法设计与分析实践报告**



|  |  |
| --- | --- |
| 专 业： | 计算机科学与技术 |
| 班 级： |  |
| 学 号： |  |
| 姓 名： |  |
| 完成日期： | 2022.05.18 |

**目 录**

[1 实验总结 1](#_Toc27595511)

[1.1已做的题目 1](#_Toc27595512)

[1.2 通过的题目 1](#_Toc27595513)

[1.3已做但未通过的题目 1](#_Toc27595514)

[2 POJ2366解题报告 2](#_Toc27595516)

[2.1题目分析 2](#_Toc27595517)

[2.2算法设计 2](#_Toc27595518)

[2.3性能分析 5](#_Toc27595519)

[2.4运行测试 6](#_Toc27595520)

[3 POJ1050解题报告 7](#_Toc27595521)

[3.1题目分析 7](#_Toc27595522)

[3.2算法设计 7](#_Toc27595523)

[3.3性能分析 11](#_Toc27595524)

[3.4运行测试 11](#_Toc27595525)

[4 POJ1042解题报告 12](#_Toc27595526)

[4.1题目分析 12](#_Toc27595527)

[4.2算法设计 12](#_Toc27595528)

[4.3性能分析 17](#_Toc27595529)

[4.4运行测试 17](#_Toc27595530)

[5 POJ1753解题报告 18](#_Toc27595531)

[5.1题目分析 18](#_Toc27595532)

[5.2算法设计 18](#_Toc27595533)

[5.3性能分析 21](#_Toc27595534)

[5.4运行测试 21](#_Toc27595535)

[6 总结 22](#_Toc27595531)

[6.1实验总结 22](#_Toc27595532)

[6.2心得体会和建议 24](#_Toc27595532)

# 1 实验总结

## 1.1已做的题目

已做题目及其分布如表1.1所示。

共计：31题

表1.1 已完成题目

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 第一单元 | 1000 | 1005 | 1753 | 3295 |  |  |
| 分治 | 2366 | 2503 | 3714 | 3233 | 2506 |  |
| 动态规划 | 1050 | 1088 | 1185 | 1636 | 2228 |  |
| 贪心算法 | 1042 | 1328 | 3040 | 1700 | 2586 |  |
| 插分约束 | 1860 | 2387 | 1062 | 3660 | 1201 | 3169 |
| 搜索 | 1324 | 1084 | 2449 | 1475 | 1077 | 1184 |

## 1.2 通过的题目

通过的题目及其序号如图1.1所示。

共计：31题

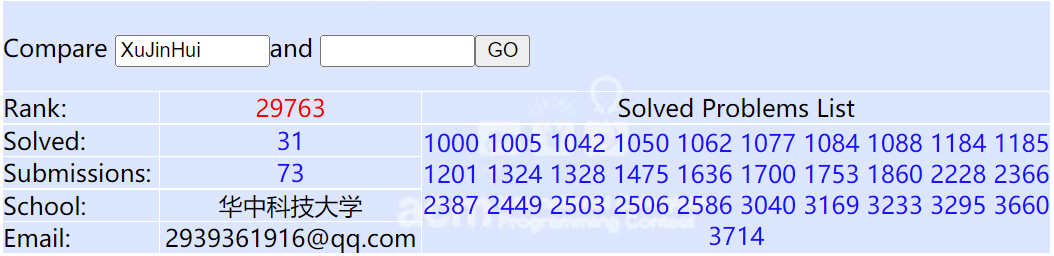


图1.1 已通过题目

## 1.3已做但未通过的题目

无

共计：0题

# 2 POJ2366解题报告

## 2.1 题目分析

POJ2366： Sacrament of the sum

描述：已知两个给定的序列， 一个升序排列，一个降序排列，在这两个序列中各找一个数，它们加起来恰好等于 10000。

输入：在列表的第一行中写入第i个列表的数字Ni的数量。此外，还有第 i 个数字列表，每个数字在其行（Ni 行）。满足以下条件：1 <= Ni <= 50000，列表的每个元素位于 -32768 到 32767 的范围内。

输出：如果可以找到这两个数，则输出“YES”；否则，输出“NO”。

## 2.2算法设计

**2.2.1 设计思路**

1) 采用二分法。即首先对b中元素排序；枚举a中元素，对每一个a中的元素在b中二分查找，看是否有与a中元素和为10000的元素，如果查找到则立即输出“YES”并终止程序，如果没有查找到则继续遍历a的下一元素。若当表a遍历完并且没有找到这两个数，则输出“NO”。二分法的流程图如图2.1所示。

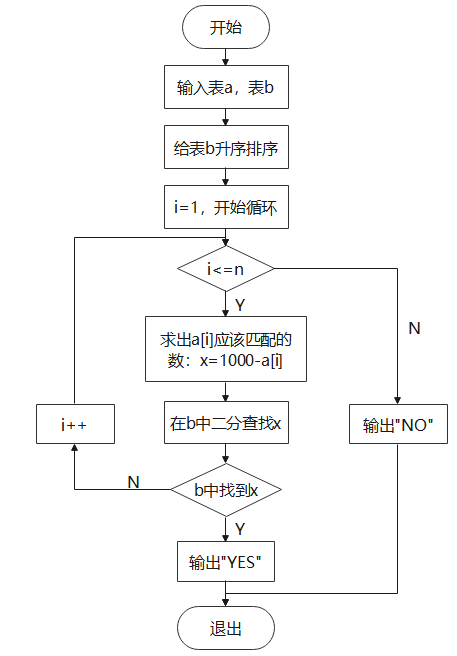


图2.1 二分法流程图

2) 采用hash法，因为题目中列表元素大小的范围为-32768到32767，我们可以开一个大小为80000的数组。我们先遍历读取第一个列表中的元素，元素为i时，可以置hash[50000+i]为1。紧接着我们遍历读取第二个列表中的元素，当元素为i时，查看hash[60000-i]是否为1，如果是则输出“YES”并终止程序。如果第二个列表遍历完并且没有符合题意的两个数，则输出“NO”。hash法的流程图如图2.2所示。

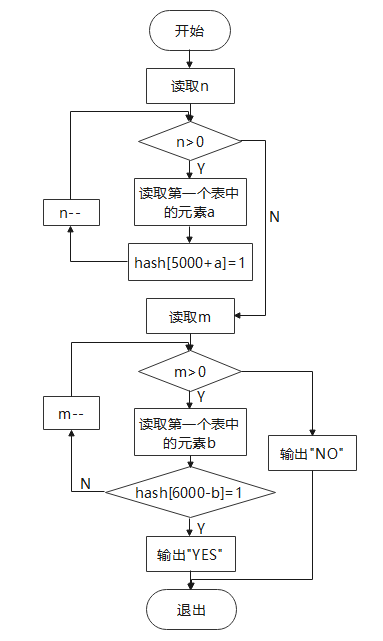


图2.2 Hash法流程图

**2.2.2 程序源代码**

代码段2.1 POJ2366二分法

#include<cstdio>

#include<algorithm>

using namespace std;

const int N=50001;

int n,m,a[N],b[N];

int main()

{

//读取两个表的数据

scanf("%d",&n);

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

scanf("%d",&a[i]);

scanf("%d",&m);

for(int i=1;i<=m;i++)

scanf("%d",&b[i]);

sort(b+1,b+m+1); //为b表排序

int l,r,mid;

for(int i=1;i<=n;i++){

int x=10000-a[i]; //求出a[i]应该匹配的数

l=1;r=m;

while(l<=r){ //二分查找

mid=(l+r)/2;

if(b[mid]<x)

l=mid+1; //找到的数比应该匹配的数小

else if(b[mid]==x)

{

puts("YES");return 0;

}//找到，退出程序

else r=mid-1; //找到的数比应该匹配的数大

}

}

puts("NO"); //没有找到

}

代码段2.2 POJ2366分治法

#include<cstdio>

#include <cstring>

bool hash[100000];

int main()

{

int a,b;

int n,m;

while (scanf("%d",&n)==1)

{

memset(hash,0,sizeof(hash));

bool ans=false;

while (n--)

{

scanf("%d",&a);

hash[50000+a]=1; //读取表a的元素

}

scanf("%d",&m);

while (m--)

{

scanf("%d",&b); //读取表b的元素

if (hash[60000-b]) //判断读取的元素是否与a中的元素和为1000

{

ans=true;

}

}

if (ans)

printf("YES\n");

else printf("NO\n");

}

return 0;

}

3) 由上述分析可知hash法效率更优，但两种方法都可以通过这道题。

## 2.3 性能分析

1) 二分法：

设升序列表大小为m，降序列表大小为n。遍历降序列表的时间为*m*，每次进行二分查找的时间为*O(logn)*，故二分法最坏时间复杂度为。

2) hash法：

hash法甚至不需要完全读取完第二个列表，设升序列表大小为m，降序列表大小为n，最坏的情况即遍历完两个列表，故hash法时间复杂度为。

## 2.4运行测试

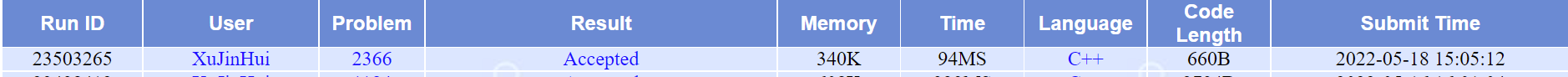


图2.3 POJ2366AC截图

# 3 POJ1050解题报告

## 3.1题目分析

POJ1050：To the Max

描述：求一个最大为100\*100矩阵中的子矩阵中元素之和的最大值，并输出。

输入：由 N \* N 个整数数组组成。输入以单独一行上的单个正整数 N 开始，表示二维正方形数组的大小。后面是由空格分隔的 N^2 个整数。这些是数组的 N^2 个整数，以行优先顺序显示。N的值最大为100。数组中的数字将在 [-127,127] 范围内。

输出：最大子矩形的总和。

## 3.2 算法设计

**3.2.1 设计思路**

如果想要求一个矩阵的子矩阵中元素之和的最大值，枚举原则上是可行的，但时间复杂度达到了，会超时，因此本题应该考虑用动态规划的思想来做。

进一步想到，本题可以用降维的角度考虑：求一个矩阵的子矩阵元素之和的最大值maxMatrix()，可以转化为求一个一维数组的最大子段和maxArray()。

maxArray()的算法思想如下：维护两个变量sumArray和maxSumArray，二者初值都为0，遍历数组，如果当前sumArray大于0，则加上当前元素，否则sumArray等于当前元素（即如果sumArray小于等于0，子段和与当前元素的和也一定小于等于当前元素，所以不如直接赋值），之后判断如果sumArray大于maxSumArray则更新后者。可以看出，这个算法的时间复杂度为线性。其流程图如图3.1所示。

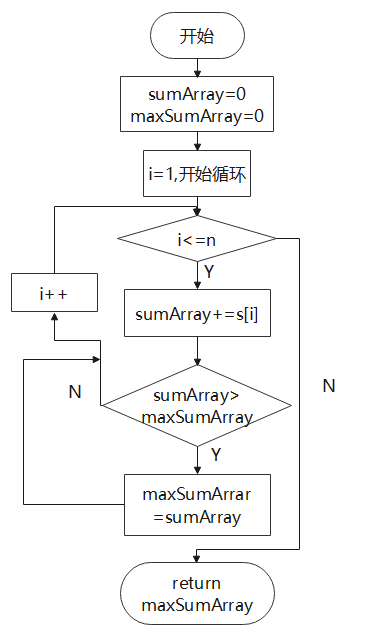


图3.1 maxArray()方法流程图

maxMatrix()的算法思想如下：两重循环枚举不同的行i和j，维护一个数组保存某一列的从i到j的元素之和，对于每次枚举，对数组求一次最大字段和。

maxMatrix()方法的流程图如图3.2所示。

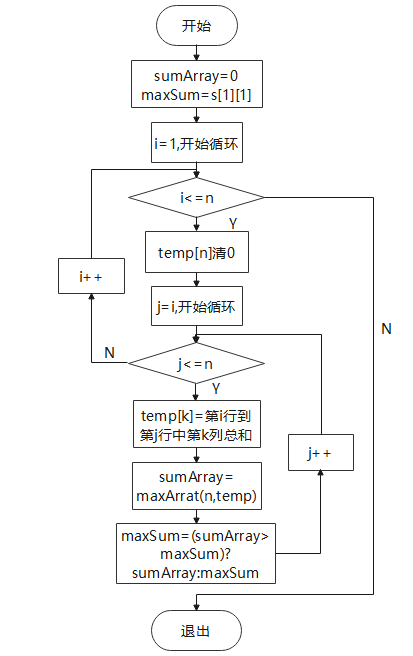


图3.2 maxMartix()方法流程图

**3.2.2 程序源代码**

代码段3.1 POJ1050

#include <iostream>

#include <cstdio>

using namespace std;

const int MAX = 105; //二维数组边长最大值

int n; //二维数组边长

int s[MAX][MAX]; //二维数组

int maxSum; //最大子矩阵和

//求出一行中最大的子段和

int maxArray(int n, int s[])

{

int sumArray = 0, maxSumArray = 0;

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

{

if (sumArray > 0)

sumArray += s[i];

else

sumArray = s[i];

if (sumArray > maxSumArray)

maxSumArray = sumArray; //更新一行最大子段和

}

return maxSumArray;

}

//求出最大子矩阵和

int maxMatrix(int n, int s[][MAX])

{

int maxSum = s[1][1];

int sumArray;

int temp[MAX];

for (int i=1; i<=n; i++) //i从第1行到第n行

{

for (int j=1; j<=n; j++) //行改变时temp清空

temp[j] = 0;

for (int j=i; j<=n; j++) //j从i行到第n行

{

for (int k=1; k<=n; k++)

temp[k] += s[j][k]; //temp[k]表示从第i行到第j行中第k列的总和

sumArray = maxArray(n, temp); //求出该行中最大的子段和

if (sumArray > maxSum)

maxSum = sumArray;

}

}

return maxSum;

}

int main()

{

while(scanf("%d", &n) != EOF)

{

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

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

scanf("%d", &s[i][j]);

maxSum = maxMatrix(n, s);

cout << maxSum << endl;

}

return 0;

}

## 3.3性能分析

考虑时间复杂度，两重循环的时间复杂度为，内部更新数组、求最大子段和的时间复杂度为，因此算法的总时间复杂度为。

## 3.4运行测试

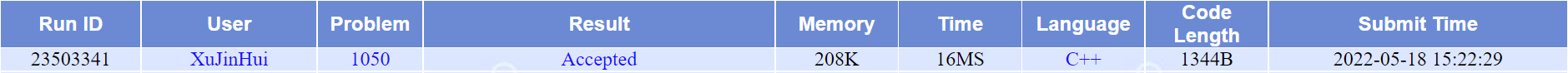


图3.2 POJ1050AC截图

# 4 POJ1042解题报告

## 4.1 题目分析

POJ1042： Gone Fishing

描述：一个人打算在编号 1~n 的湖里钓鱼，钓鱼是单向走的，不能往回走。给你 n 个湖，每个湖初始鱼的数量 pi， 每次每个湖钓鱼后鱼的减少量 di，第 i 个湖到第 i+1 湖的距离时间ti（单位是 5min）， 可以在任何湖停止钓鱼。 求如何钓鱼才能才能在 h 小时内钓鱼量最多。

输入：每个案例都以包含 n 的行开头。接下来是包含 h 的行。接下来，有一行 n 个整数指定 fi (1 <= i <=n)，然后是一行 n 个整数 di (1 <=i <=n)，最后是一行 n - 1 个整数 ti ( 1 <=i <=n - 1)。输入以 n = 0 的情况终止。

输出：对于每个测试用例，打印在每个湖上花费的分钟数，用逗号分隔。接下来是包含预期鱼数的行。如果存在多个计划，选择在1号湖停留时间尽可能长的计划，即使预计在某些间隔内不会捕获任何鱼。如果仍有平局，则选择在湖2花费尽可能长的那个，依此类推。在案例之间插入空行。

## 4.2 算法设计

**4.2.1 设计思路**

这道题的题目要求有些复杂，需要认真模拟题意，同时利用枚举+贪心的思想可以AC。

事实上我们可以将在路上的时间timeWay和钓鱼的时间timeFishing分开考虑，它们的和为h。而在路上的时间timeWay取决于最远到了哪个湖，与钓鱼时间的分配无关。

同时，每个湖都可能为终点，因此我们可以枚举每个湖为终点的情况，这样路上的时间timeWay和钓鱼的时间timeFishing就都确定了，我们直接根据timeFishing的值来分配钓鱼时间即可。但是要注意我们只是在假设每个湖都可能为终点，实际上能不能到这个湖在程序中还要判断，如果到不了更远的湖直接退出循环，即不再考虑这个湖及更远的湖的情况了。

在钓鱼时间的分配上，利用贪心的思想，每个5分钟都选择期望钓鱼最多的那个湖，并且每5分钟要根据题意更新“这个湖”下一次能够钓鱼的数量。如果最后所有湖里都没有鱼但还有时间的话，根据题意就把多余的时间分配给第一个湖。

在上述描述中，虽然我们每隔5分钟根据贪心思想确定一个收益最大的湖，但是实际在钓鱼过程中，John不会走回头路。举一例说明：根据贪心，湖的选择为1->2->1，但是实际过程中，John会在湖1钓鱼10分钟，再去湖2钓鱼5分钟，即使在湖1的最后五分钟钓鱼量不如湖2的五分钟，但是总体上这种方案是最优的。

此贪心算法的流程图如图4.1所示。

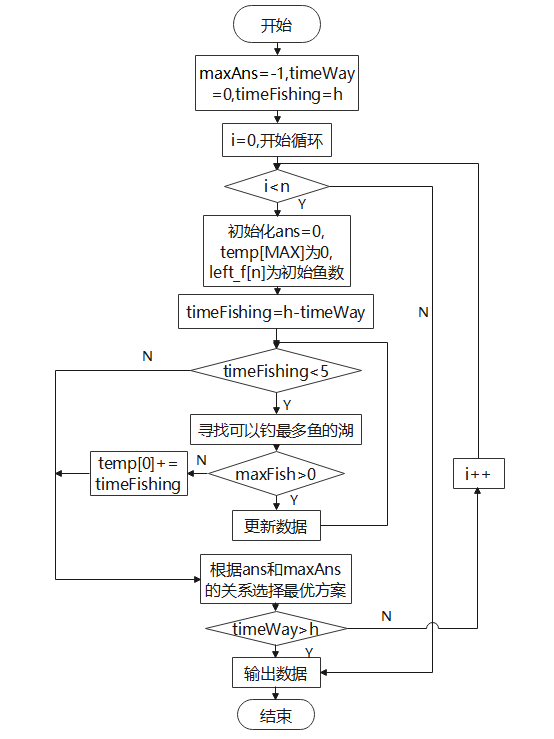


图4.1 贪心算法流程图

2. 注意事项：

1) 因为路为单向可达，故在选取期望最大的湖时，若有两湖或多湖期望相同，序号小的应优先考虑。

2) 在对期望进行减法计算时，若期望相减之后为负数，应将其置为0。

3) 若所有湖都没有鱼，且还有剩余时间，默认将剩余时间加到第一个湖上。

4) 注意输出格式，实验过程中由于未注意输出格式，导致多次未通过。

5) 需要想清楚di在什么时候减少，题目中描述：对于每个湖来说，在开始捕捞之后，每隔5分钟可以捕获的鱼的数量才会减少，所以说不是随着时间推移，所有的湖里的鱼都会一直减少，这也是非常容易犯错的地方。

**4.2.2 程序源代码**

代码段4.1 POJ1042

#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

const int MAX = 30;

int h; //时间(分钟)

int n; //湖数

int f[MAX] = {0}; //初始鱼数

int d[MAX] = {0}; //鱼递减数目

int left\_f[MAX] = {0}; //剩下的鱼

int t[MAX-1] = {0}; //两湖间隔距离

int stay[MAX] = {0}; //各湖停留时间

int maxAns; //最大钓鱼数

void dp()

{

maxAns = -1;

int timeWay = 0; //路上时间

int timeFishing = h; //钓鱼时间

int temp[MAX]; //保存每次枚举的停留时间

//枚举所有湖为终点的情况

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

{

int ans = 0;

memset(temp, 0, sizeof(temp));

timeFishing = h - timeWay; //每次更换一个新的终点，由于路程变远，可用来钓鱼的时间会减少

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

left\_f[j] = f[j];

//贪心算法

while (true)

{

if (timeFishing < 5) //时间不够下一次钓鱼

break;

//寻找可以钓最多鱼的湖

int maxFish = -1;

int id = 0;

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

{

if (left\_f[j] > maxFish)

{

maxFish = left\_f[j];

id = j;

}

}

if (maxFish == 0) //如果没有鱼了，则多余的时间在第一个湖停留

{

temp[0] += timeFishing;

break;

}

//更新数据

temp[id] += 5;

ans += left\_f[id];

left\_f[id] -= d[id];

if(left\_f[id] < 0)

left\_f[id] = 0;

timeFishing -= 5;

}

if (ans > maxAns) //如果该情况下大于最大值则更新

{

maxAns = ans;

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

stay[j] = temp[j];

}

else if (ans == maxAns) //等于最大值选择先到湖停留时间最长的方案

{

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

{

if (temp[j] < stay[j])

break;

if (temp[j] > stay[j])

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

stay[k] = temp[k];

}

}

//在无法到达更远的湖时直接退出循环

timeWay += t[i];

if (timeWay > h)

break;

}

for (int i=0;i<n-1;i++)

cout << stay[i] << ", ";

cout << stay[n-1];

cout << "\nNumber of fish expected: " << maxAns << endl << endl;

}

int main()

{

while(true)

{

cin >> n;

if (n == 0)

break;

cin >> h;

h \*= 60; //hour to minute

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

cin >> f[i];

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

cin >> d[i];

for (int i=0;i<n-1;i++)

{

cin >> t[i];

t[i] \*= 5; //to minute

}

dp();

}

return 0;

}

## 4.3 性能分析

假设while循环的循环次数为*k*，由于本题中有for循环、while循环、for循环3重嵌套循环，故算法的时间复杂度为。若本题使用堆来解，则可把时间复杂度降为*。*

## 4.4 运行测试

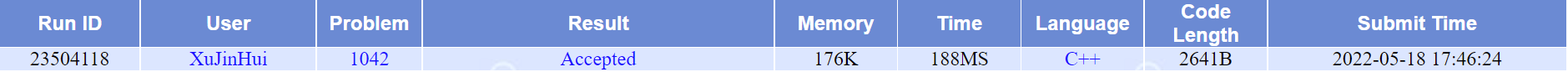


图4.2 POJ1088AC截图

# 5 POJ1753解题报告

## 5.1实验题目

POJ1636：Prison rearrangement

描述： 有 4\*4 的正方形，每个格子要么是黑色，要么是白色，当把一个格子的颜色改变(黑->白或者白->黑)时，其周围上下左右(如果存在的话)的格子的颜色也被反转，求至少反转几个格子可以使 4\*4 的正方形变为纯白或者纯黑。

输入： 输入由 4 行组成，每行 4 个字符“w”或“b”表示游戏场位置。

输出： 输出从给定位置实现游戏目标所需的最少轮数。如果最初实现了目标，则写 0。如果无法实现目标，则输出“Impossible”。

## 5.2 算法设计

**5.2.1 设计思路**

如果用一个4\*4的数组存储每一种状态，不但存储空间很大，而且在穷举状态时也不方便记录。因为每一颗棋子都只有两种状态，所以可以用二进制0和1表示每一个棋子的状态，则棋盘的状态就可以用一个16位的整数唯一标识。而翻转的操作也可以通过通过位操作来完成。显然当棋盘状态id为0（全白）或65535（全黑）时，游戏结束。

对于棋盘的每一个状态，都有十六种操作，首先要判断这十六种操作之后是否有完成的情况，如果没有，则再对这十六种操作的结果分别再进行上述操作，显然这里就要用到队列来存储了。而且在翻转的过程中有可能会回到之前的某种状态，而这种重复的状态是不应该再次入队的，所以维护 Visit[i]数组来判断 id==i 的状态之前是否已经出现过，如果不是才将其入队。如果游戏无法完成，状态必定会形成循环，由于重复状态不会再次入队，所以最后的队列一定会是空队列。

由于0^1=1，1^1=0，所以翻转的操作可以通过异或操作来完成，而翻转的位置可以通过移位来确定。

bfs算法的流程图如图5.1所示。

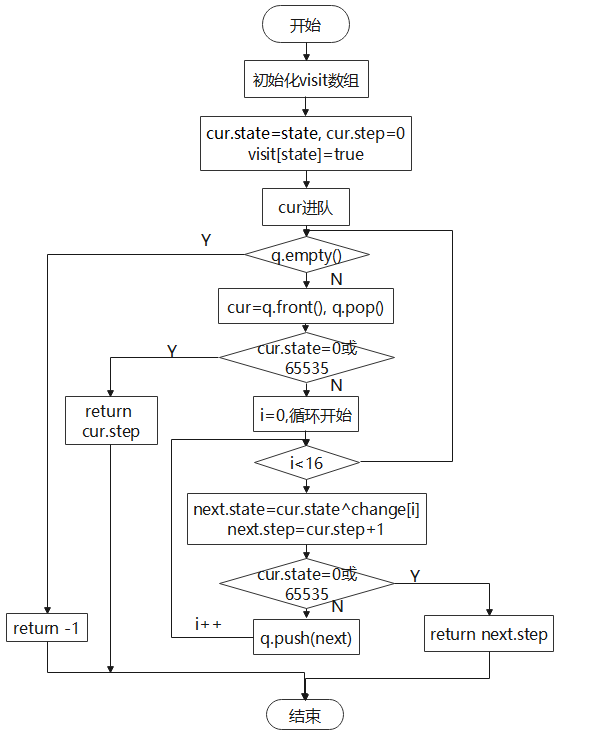


图5.1 bfs算法流程图

**5.2.2 程序源代码**

代码段5.1 POJ1753

#include<iostream>

#include<queue>

#include<cstdio>

#include<memory.h>

using namespace std;

struct Node

{

int state;

int step;

};

bool visit[65536];

int change[16] = //16种状态转换，对应4\*4的翻子位置

{

51200,58368,29184,12544,

35968,20032,10016,4880,

2248,1252,626,305,

140,78,39,19

};

int bfs(int state)

{

int i;

memset(visit,false,sizeof(visit)); //标记每一个状态都未访问过

queue<Node>q;

Node cur,next;

cur.state = state;

cur.step = 0;

q.push(cur);

visit[state] = true;

while(!q.empty())

{

cur = q.front();

q.pop();

if(cur.state == 0 || cur.state == 0xffff) //65535

return cur.step;

for(i=0;i<16;i++)

{

next.state = cur.state^change[i];

next.step = cur.step + 1;

if(visit[next.state])

continue;

if(next.state == 0 || next.state == 0xffff) //65535

return next.step;

visit[next.state] = true;

q.push(next);

}

}

return -1;

}

int main(void)

{

int i,j,state,ans;

char ch[5][5];

while(scanf("%s",ch[0])!=EOF)

{

for(i = 1 ; i < 4 ; ++i)

scanf("%s",ch[i]);

state = 0;

for(i = 0 ; i < 4 ; ++i)

{

for(j = 0 ; j < 4 ; ++j)

{

state <<= 1;

if(ch[i][j] == 'b')

state += 1;

//state ^= (1<<((3-i)\*4+(3-j)));

}

}

ans = bfs(state);

if(ans == -1)

puts("Impossible");

else

printf("%d\n",ans);

}

return 0;

}

## 5.3性能分析

本题通过状压枚举和bfs求解，由于棋盘的大小固定，为4x4棋盘，故本算法的时间和空间复杂度均为。若棋盘变为 nxm的矩阵，利用类似方法，则时间复杂度为，空间复杂度为。

## 5.4运行测试

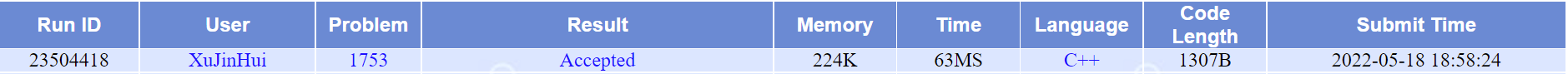


图4 POJ1753AC截图

# 6 总结

## 6.1 实验总结

此次实验主要涵盖了分治、动态规划、贪心算法、最短路径、搜索这几个方面，这些都是最经典也最常用的算法知识，做这些方面的题目也让我在面对一道道陌生的算法题时更加灵活、思路更加开阔。在此简单总结以上几类算法的主要特点和用法：

1. 分治

分治算法的一般解题步骤为：

分解：将原问题分解为若干个规模较小、相对独立、与原问题形式相同的子问题

解决：若子问题规模较小且容易解决时，则直接解；否则，递归地解决各子问题

合并：将各子问题的解“合并”为原问题的解

简单理解就是分而治之，将一个复杂的问题通过一定的方式分解成若干个类似的小问题，故分治法的主要实现方式就是循环递归。

在刚开始遇到这类递归的问题时，由于追求细节，常常迷失在递归过程中，造成自我怀疑。故我们在刚面对递归时，可以首先了解大概思路，而不是纠结于递归中的细节，必要时可以自己动手画递归树帮助理解。

1. 动态规划

动态规划的3个重要解题步骤为：

①定义数组元素的含义。在使用动态规划时，我们一般会用一个数组，来保存历史数据，假设为一维数组 dp[]。那么，规定dp[i]代表什么在解题中十分重要。

②找出数组元素之间的关系式，类似于数学归纳中的关系式。当我们要计算 dp[n] 时，是可以利用 dp[n-1]，dp[n-2].....dp[1]，来推出 dp[n] 的，也就是可以利用历史数据来推出新的元素值，所以我们要找出数组元素之间的关系式。例如 dp[n] = dp[n-1] + dp[n-2]，这个就是他们的关系式。而这一步，也是最难最重要的一步。

③找出初始值。虽然我们知道了数组元素之间的关系式，例如 dp[n] = dp[n-1] + dp[n-2]，我们可以通过 dp[n-1] 和 dp[n-2] 来计算 dp[n]，但是，我们得知道初始值才能真正计算出dp[n]。例如一直推下去的话，会有dp[3] = dp[2] + dp[1]。而 dp[2] 和 dp[1] 是不能再分解的了，所以我们必须要能够直接获得 dp[2] 和 dp[1] 的值，这就是所谓的初始值。

由于动态规划适用于有重叠子问题和最优子结构性质的问题，并且能够记录所有子问题的结果，因此动态规划方法所耗时间往往远少于朴素解法。灵活掌握它对我们提升算法性能有很大的帮助。

在本次实验的题目中，不仅有一维的动态规划，还有二维的动态规划算法，这让我能够在不同的情境下灵活运用，拓展了算法思维。

1. 贪心算法

贪心算法是一种通过局部的最优解构造全局的最优解的算法，其一般解题步骤为：

①根据题意，选取量度标准；

②按量度标准排序并输入；

③判断输入是否与当前已构成的最优解可构成一个可行解。

其核心是找到最优的量度标准。可以认为是在构造解的过程中使用最小的代价，极大或极小化目标函数，从而给接下来构造解留下尽量大的空间。

1. 最短路径

对最短路径最常用的算法总结如下：

①单源算法——Dirkdtra算法(使用最广且必须掌握的算法)

算法思想：每次找到离源点最近的一个点，以该点为中心，更新源点到其他源点的最短路径，贪心的思想。该算法无法判断是否存在负权环路，如果存在，算法将失效。

②单源算法——Bellman\_ford算法(解决负权回路的算法)

算法思想：假设p为源点到节点的最短路径，显然这条路径上最多包含n - 1条边，那么我们可以通过 n - 1 次循环，每次循环松弛所有边，根据路径松弛定理，最终可以得到正确的答案。由于进行了全面的松弛，最后得到的结果根据三角形定则，一定有 dis[v] < dis[u] + w(假设有边u->v = w)，否则即存在负边回路。

③多源算法——Floyd算法(简洁而优雅的算法)

算法思想：该算法通常用以解决所有节点对的最短路径，该算法利用了这样的事实：如果从节点 i 到节点 j，如果存在一条更短的路径话，那么一定是从另一个节点 k 中转而来，即有 d[i][j] = min(d[i][j],d[i][k]+d[k][j])，而d[i][k]和d[k][j]可以用一样的思想去构建，可以看出这是一个动态规划的思想。在构建i,j中，我们通过枚举所有的k值来进行操作。但是，该算法无法判断负权回路。

1. 搜索

①BFS

BFS算法本质上就是从一个图的起点出发开始搜索找到目标终点完成搜索。整个解决步骤一般如下：

起点入队列；以队列非空为循环条件，进行节点扩散（将所有队列节点出队，同时判断出队节点是否为目标节点，获取其邻接结点）；判断获取的节点是否已被遍历，未被遍历节点入队。

②DFS

与广度优先搜索不同，深度优先搜索（DFS）类似于树的先序遍历。在搜索时会尽可能的沿着一条所有路径进行搜索，直到该条路径上所有节点搜索完成，然后切换到另一条路径上进行搜索，直到图的所有节点全部都被遍历。因此广度优先搜索整个过程可以分成如下步骤：

判断终止条件；对节点进行访问并加入到访问链表中；以当前节点的邻接结点为起点，通过递归向更深层次进行搜索。

## 6.2 心得体会和建议

通过最近一段时间做算法题，我有了如下心得收获：

首先要坚持独立完成，即使通过之后也要多阅读他人的题解，百家争鸣，我觉得在与他人思想碰撞中才会产生更大的收获。一道题目往往有不同的解法，多寻找不同的解法有利于我们触类旁通，更能灵活运用自己掌握的数据结构和算法知识。基于此，我觉得POJ一点不便之处就是在交流讨论的便捷性方面，和洛谷、leetcode等在线oj平台还有较大差距。

刷题不在多，要对每一道题认真读题、理解透彻、思考完备，这样才能达到举一反三的效果。此外，若跳跃着做不同部分的算法题，譬如刚做一道动态规划再做一道最短路径，很容易出现囫囵吞枣、一头雾水的情况。所以，刚开始刷题时，可以先做一个类型的题，增加自己对此类型的熟练程度，并及时总结这种类型的题。在熟练掌握这个类型的题目后，再转向下一类型。

每道题的场景都是不一样的，同样的算法在类似的题目中很可能效率不够最优，我在一些题目中就遇到了这类问题，总是思考为什么在这道题里这种算法就不能通过了，这需要我们深刻理解算法的原理，同时也要不断尝试、多多思考其他算法的可能性。

刷题中除了要训练算法思维，也要熟练掌握STL和一些常用的高精度、快排模板，可以提高做题效率，同时也要掌握一些常用的tricks，就比如在POJ1753中，运用压缩的思想，把4x4的数组压缩为16个数字，这样可以大大减少时间复杂度和空间复杂度。