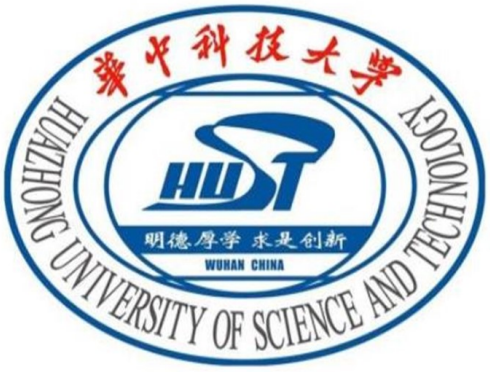
華中科技大學

《算法设计与分析》

实 验 总 结 报 告



计 算 机 科 学 与 技 术 学 院

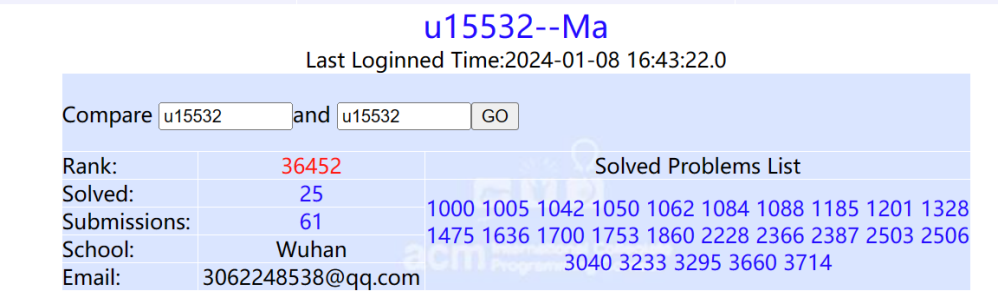
目　　录

1 实验总结 …… 1
1.1 已做且 AC 的题目 …… 1
1.2 尝试做的题目 …… 1
1.3 题解报告 …… 1
2 POJ 3714 解题报告 …… 2
2.1 题目大意 …… 2
2.2 算法设计 …… 2
2.3 性能分析 …… 4
2.4 运行测试 …… 4
3 POJ 1185 解题报告 …… 5
3.1 题目大意 …… 5
3.2 算法设计 …… 5
3.3 性能分析 …… 6
3.4 运行测试 …… 6
4 POJ 1328 解题报告 …… 7
4.1 题目大意 …… 7
4.2 算法设计 …… 7
4.3 性能分析 …… 8
4.4 运行测试 …… 8
5 POJ 1860 解题报告 …… 9
5.1 题目大意 …… 9
5.2 算法设计 …… 9
5.3 性能分析 …… 11
5.4 运行测试 …… 11
6 总结 …… 12
6.1 在实验过程中的几点收获 …… 12
6.2 个人感想 …… 12

1 实验总结

1.1 已做且 AC 的题目

- 第一单元四道题:1000、1005、1753、3295
- 分治五道题:2366、2503、3714、3233、2506
- 动态规划五道题:1050、1088、1185、1636、2228
- 贪心策略 4 道题：1042、1328、3040、1700
- 最短路 + 差分 5 道题：1860、2387、1062、3660、1201
- 搜索 2 题：1084、1475



POJ AC 记录

1.2 尝试做的题目

尝试学习 A* 算法处理搜索部分题目，但因为时间来不及和难度过大，遂放弃。

1.3 题解报告

报告选择在解决过程中遇到困难或印象深刻的 4 道典型题，分析问题并阐述个人的解题思路。

2 POJ 3714 解题报告

2.1 题目大意

有两组坐标，一组是 n 个发电站的坐标，一组是 n 个士兵的坐标。求哪个士兵离发电站的距离最短。

这个问题可以看作是一个最近点对问题，即找出平面上的两组点中距离最近的一对点。其中一组点表示能量站的位置，另一组点表示特工的位置。我们需要找出特工距离最近的能量站。可以用分治算法进行处理，可以高效解决。

2.2 算法设计

首先我们需要对问题进行建模将发电站和士兵的位置表示为二维平面上的点。可以使用一个结构体来表示每个点，包括 x 坐标、y 坐标和类型（发电站或士兵）。

```
#结构体的定义
struct Point
{
    double x, y;
    int type;
}a[200005];
```

solve 函数是分治算法的核心部分。它接收一个左边界 l 和一个右边界 r，表示当前需要处理的点的范围。进入函数体后要先处理基本的情况，如果 l 和 r 相等，表示只有一个点，返回一个较大的距离。如果 l 和 r 相差 1，并且两个点的类型不同，计算它们之间的距离并返回。如果 l 和 r 相差 1，并且两个点的类型相同，返回一个较大的距离。

```
#处理三种情况
if (l == r)
   return Max;
if (l == r - 1 && a[l].type != a[r].type)
   return sqrt(pow(fabs(a[l].x - a[r].x), 2) + pow(fabs(a
      [l].y - a[r].y), 2));
if (l == r - 1 && a[l].type == a[r].type)
```

```
    return Max;
```

在基本情况处理之后，使用分治法进行递归处理。首先计算中间点的索引 mid，然后递归调用 solve 函数计算左半部分和右半部分的最小距离，分别记为 d1 和 d2。

```
#对三种情况进行递归求解
int mid = l + r >> 1;
double d1 = solve(l, mid);
double d2 = solve(mid + 1, r);
double d = min(d1, d2);
```

接下来，将从左边界 l 到右边界 r 中距离中间点的 x 距离小于 d 的点放入一个临时数组 temp 中，并按照 y 坐标从小到大排序。遍历 temp 数组中的点，计算每对点之间的距离，并更新最小距离 d。最后，清空临时数组 temp，返回最小距离 d。

```
#对子问题的解进行合并
int i;
for (i = l; i <= r; i++)
    if(fabs(a[i].x - a[mid].x) < d)
        temp.push_back(a[i]);
sort(temp.begin(), temp.end(), cmpy);
int j;
for (int i = 0; i < temp.size(); i++)
    for (int j = i + 1; j < temp.size() && temp[j].y -
       temp[i].y < d; j++){
        if(temmp[i].type == temp[j].type)
            continue;
        double d3 = sqrt(pow(fabs(temp[i].x - temp[j].x),
            2) + pow(fabs(temp[i].y - temp[j].y), 2));
        d = min(d, d3);
    }
```

2.3 性能分析

- 时间复杂度: 排序操作 $sort(a+1, a+2*n+1, cmp1)$ 的时间复杂度为 $O(logn)$，其中 n 是点的数量。分治算法的时间复杂度为 $O(nlogn)$，因为每次递归都将点集划分为两部分，递归深度为 $O(logn)$，而在每个递归层级上的操作都是线性时间复杂度的。对于每个递归层级，遍历 temp 数组的时间复杂度为 $O(n)$，而遍历 temp 数组的内部循环时间复杂度为 $O(n^2)$。因此，总体时间复杂度为 $O(n^3)$。
- 空间复杂度：除了输入数据之外，额外使用的空间主要是临时数组 temp。temp 数组的大小取决于每次递归中满足条件的点的数量，最坏情况下可能为 $O(n)$。综上所述，总体空间复杂度为 $O(n)$。
- 算法改进：在遍历 temp 数组查找距离小于 d 的点时，可以使用更高效的数据结构，例如平衡树（如红黑树）或者线段树，以加速查找过程，将时间复杂度优化为 $O(nlogn)$。在代码中，使用了全局变量和全局数组。这样的设计不利于代码的封装和扩展性。可以考虑将相关变量和函数封装在一个类中，更好地组织代码结构。

2.4 运行测试

平台测试集本地运行结果：

```
2
4
0 0
0 1
1 0
1 1
2 2
2 3
3 2
3 3
1.414
4
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
```

User	Problem	Result	Memory	Time	Language
u15532	3714	Accepted	4880K	1422MS	C++

POJ3714 AC 截图

3 POJ 1185 解题报告

3.1 题目大意

题目要求在一个 N*M 的网格地图上部署炮兵部队，使得在防止误伤的前提下，最大化能够摆放的炮兵部队数量。每个格子有两种可能的状态：山地（”H”）和平原（”P”）。在平原上可以部署一支炮兵部队，而山地不能部署。炮兵部队的攻击范围是沿横向左右各两格，沿纵向上下各两格。需要考虑的限制条件是任何两支炮兵部队之间不能互相攻击，即任意一支炮兵部队都不在其他炮兵部队的攻击范围内。

3.2 算法设计

建模：使用了状态压缩的技巧，将每一行的地势信息转换为二进制表示。其中，1 表示山地，0 表示平原。这样可以利用二进制数的特性来表示每个格子是否可以放置炮兵部队。在预处理阶段，代码通过遍历所有可能的状态，判断每个状态是否合法。其中合法状态满足以下条件：同一行中，每三个连续的格子中只能同时存在一个 1，即炮兵不能互相攻击。同时，统计每个合法状态中 1 的数量。预处理完合法状态后，代码进入动态规划的阶段。

```
bool check(int x) // 判断这个状态是否合法
{
    int cnt = 0, sum;
    while (x)
    {
        if ((x & 1) && cnt)
            return 0;
        if (x & 1)
            cnt = 3;
        if (cnt)
            cnt--;
        x >>= 1;
```

```
    }
    return 1;
}
```

动态规划的过程中，可以利用三维数组 f 来存储状态转移的结果。其中，设 $f[i][j][k]$ 表示第 i 行，这一行和上一行的放置方案为 j 和 k 的方案数。设上上行的状态为 l，则有

$$f[i][j][k] = max(f[i][j][k], f[i-1][k][l] + sum[j])$$

其中 $sum[j]$ 表示状态为 j 的放置个数。

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= tot; j++)
        if (i == 1 || !(q[j] & map[i - 2]))
            for (int k = 1; k <= tot; k++)
                if (!(q[k] & map[i - 1]) && !(q[j] & q[k]))
                    for (int l = 1; l <= tot; l++)
                        if (!(q[l] & map[i]) && !(q[j] & q[l]) && !(q[k] & q[l]))
                            f[i][l][k] = max(f[i][l][k], f[i - 1][k][j] + sum[l]);
```

优化后的 DP 处理状态转移代码

3.3 性能分析

这样处理后，时间复杂度为 $O(n \times (2^m)^3)$，空间复杂度为 $O(nm^2)$，时空效益非常低。但是我们会发现枚举时有太多状态是不成立的。也就是说这个状态中就有两个 1 之间的距离小于 3,。所以我们可以预处理出所有的合法的状态（即任意两个 1 之间的距离至少为 3），然后直接枚举这些状态（顺便预处理出 sum 数组）。这样可以省去很多的无用状态，大大减少时间复杂度。然后我们为了减少空间，可以设 $f[i][j][k]$ 表示第 i 行，这一行的状态是第 j 个合法的状态，上一行的状态是第 k 个合法的状态时的最多放置方案数。方程形式总体不变。

3.4 运行测试

平台测试集本地运行结果：

```
5 4
PHPP
PPHH
PPPP
PHPP
PHHP
6
PS D:\Programme\algorithm>
```

User	Problem	Result	Memory	Time	Language
u15532	1185	Accepted	1704K	375MS	C++

POJ 1185 AC 截图

4 POJ 1328 解题报告

4.1 题目大意

假设海岸线是一条无限延伸的直线。陆地在海岸线的一侧，而海洋在另一侧。每一个小的岛屿是海洋上的一个点。雷达坐落于海岸线上，只能覆盖 d 距离，所以如果小岛能够被覆盖到的话，它们之间的距离最多为 d。计算出能够覆盖给出的所有岛屿的最少雷达数目。

4.2 算法设计

这道题目可以使用贪心策略来解决。对于每个小岛，我们需要找到它能够被雷达覆盖到的最远距离。我们可以根据小岛的位置和雷达的覆盖范围，计算出小岛的左右边界。接下来，我们对所有小岛按照左边界进行排序。这样做的目的是为了贪心地选择雷达的位置，使得雷达的覆盖范围能够尽可能多地覆盖小岛。

我们从左到右遍历排序后的小岛，使用贪心策略选择雷达的位置。我们维护一个当前的覆盖范围，初始时将左边界和右边界设为第一个小岛的边界。然后，我们逐个考虑后续的小岛，如果一个小岛的左边界在当前覆盖范围内，那么我们可以保持当前的覆盖范围不变，只需要更新右边界为当前小岛的右边界。如果一个小岛的左边界在当前覆盖范围外，说明当前的雷达无法覆盖到该小岛，我们需要选择一个新的雷达位置，将当前的覆盖范围更新为该小岛的边界。通过贪心策略的选择，我们可以得到覆盖所有小岛所需的最少雷达数目。

```
#运用贪心思想实现核心代码
void solve() {
    if (flag){
      printf("Case %d: -1\n", num);
      return;
    }
    sort(ils + 1, ils + 1 + n, cmp);
    db nowl, nowr, nextl, nextr;
    nowl = ils[1].first;
```

```
        nowr = ils[1].second;
        ans++;
        for(int i = 2; i <= n; i++) {
            nowl = ils[i].first;
            nowr = ils[i].second;
            if (nowl > nowr) {
                nowl = nextl;
                nowr = nextr;
                ans++;
            }
            else {
                nowl = max(nowl, nextl);
                nowr = min(nowr, nextr);
            }
        }
        printf("Case %d: %d\n", num, ans);
}
```

4.3 性能分析

外层的 while 循环会执行多次，直到输入的小岛数量 n 和雷达的覆盖范围 d 同时为零为止。因此，该部分的时间复杂度可以看作是 $O(T)$，其中 T 表示测试用例的数量。内层的 for 循环会遍历所有的小岛，时间复杂度为 $O(n)$。在 for 循环中，对每个小岛计算边界、排序小岛数组的时间复杂度均为 $O(nlogn)$。solve() 函数中的贪心策略的时间复杂度为 $O(n)$。

因此，总体的时间复杂度为 $O(T * nlogn)$。

4.4 运行测试

平台测试集本地运行结果：

```
3 2
1 2
-3 1
2 1
Case 1: 2

1 2
0 2
Case 2: 1

0 0
```

User	Problem	Result	Memory	Time	Language
u15532	1328	Accepted	248K	47MS	C++

POJ 1328 AC 截图

5 POJ 1860 解题报告

5.1 题目大意

给定 N 种货币，某些货币之间可以相互兑换，现在给定一些兑换规则，问能否从某一种货币开始兑换，经过一些中间货币之后，最后兑换回这种货币，并且得到的钱比之前的多。

5.2 算法设计

将每种货币看作图中的一个节点，共有 N 个节点。对于每个兑换点，可以按照给定的货币兑换率和手续费，构建有向边。如果一个兑换点可以将货币 A 兑换为货币 B，那么我们在图中从节点 A 指向节点 B，边的权重为兑换率乘以(1-手续费)。同理，如果兑换点可以将货币 B 兑换为货币 A，那么我们在图中从节点 B 指向节点 A，边的权重为兑换率乘以（1-手续费)。图中的起始节点为货币 S 所对应的节点，目标是找到一条最短路径（权重之积最大)，使得最终金额更大。

```
#边类的定义及其函数
struct Edge {
        int from, to;
        double r, c;
        Edge(int f, int t, double r, double c) : from(f),
            to(t), r(r), c(c) {}
        Edge() {}
};
```

```
void addEdge(int from, int to, double r1, double c1,
   double r2, double c2) {
        edges.push_back(Edge(from, to, r1, c1));
        edges.push_back(Edge(to, from, r2, c2));
        int len = edges.size();
        G[from].push_back(len-2);
        G[to].push_back(len-1);
}
```

实现的思路：使用 Bellman-Ford 算法求解最短路径。初始化距离数组 dis，将起始节点的距离设为给定的初始金额 fu，其他节点的距离设为 0。进行 n-1 次松弛操作，每次遍历所有边，更新距离数组 dis。如果更新了某个节点的距离，表示找到了更优的路径。最后再进行一次遍历，如果仍然存在可以松弛的边，则说明存在正环，即存在一条路径可以无限增加金额。如果最终金额 dis[目标节点]大于初始金额 fu，则可以通过一系列货币兑换操作增加金额，返回”YES”；否则返回”NO”。

```
bool bellmanFord(int s) {
    queue<int> q;
    memset(inq, 0, sizeof(inq));
    memset(cnt, 0, sizeof(cnt));
    for(int i = 1; i <= n; i++)    dis[i] = -INF;
    dis[s] = v;
    inq[s] = true;
    q.push(s);

    while(!q.empty()) {
        int u = q.front();  q.pop();
        inq[u] = false;
        for(int i = 0; i < G[u].size(); i++) {
            Edge& e = edges[G[u][i]];
            if(dis[u] > -INF && dis[e.to] < (dis[u] - e.c) * e.r) {
                dis[e.to] = (dis[u] - e.c) * e.r;
                if(!inq[e.to]) {
                    q.push(e.to);
                    inq[e.to] = true;
                    if(++cnt[e.to] > n) return false;
                }
            }
        }
    }
    return true;
}
```

Bellman_Ford 算法实现

5.3 性能分析

- 时间复杂度: 初始化距离数组 dis，时间复杂度为 $O(N)$。进行 n-1 次松弛操作，每次遍历所有边，时间复杂度为 $O((n-1)*E)$。最后再进行一次遍历，判断是否存在正环，时间复杂度为 $O(M)$。总的时间复杂度为 $O(N+(n-1)*E+E)$，可以简化为 $O(nE)$。
- 空间复杂度：结构体数组存储兑换点的信息，空间复杂度为 $O(n)$。距离数组 dis，空间复杂度为 $O(E)$。总的空间复杂度为 $O(n+E)$。
- 算法改进：使用 Dijkstra 算法：由于该问题中不存在负权边，可以使用 Dijkstra 算法求解单源最短路径。相比于 Bellman-Ford 算法，Dijkstra 算法的时间复杂度更低，为 $O((n+e)logn)$，其中 n 为节点数，e 为边数。使用优先队列（堆）来维护当前最短路径的选择过程。在构建图时，还可以根据题目给定的兑换率和手续费的关系，进行一些剪枝操作。例如，如果存在两个节点 A 和 B，其中 A 可以直接兑换到 B，而 B 又可以直接兑换回 A，并且兑换率和手续费满足一定条件，那么可以将 A 和 B 之间的边直接省略，从而减少图的规模和计算量。

5.4 运行测试

平台测试集本地运行结果：

```
3 2 1 20.0
1 2 1.00 1.00 1.00 1.00
2 3 1.10 1.00 1.10 1.00
YES
```

User	Problem	Result	Memory	Time	Language
u15532	1860	Accepted	184K	0MS	C++

POJ1860 AC 截图

6 总结

6.1 在实验过程中的几点收获

- 动态规划是我印象最为深刻的算法，它往往比分治更高效又比适用于一些贪心不能处理的情况。我认为处理动态规划问题关键在于确定状态和状态转移方程。状态是问题的关键属性，它的变化会影响问题的解。通过定义合适的状态和状态转移方程，我们可以将问题转化为子问题的求解，并利用已知的子问题解来推导出更大规模问题的解。这种自底向上的求解方法能够有效地降低时间复杂度。
- 在求解问题时我们应该遵循问题分析到结合理论贴合实际情况设计算法，在设计算法的时候，依据题目分析的内容，在草稿纸上列出一个简单的数据测试过程可以更快地找到算法的设计要点以及思路。

6.2 个人感想

算法对思维逻辑的要求极高，学习新的算法有利于开拓逻辑思维。它要求我们敏锐地把握问题的本质，进行数学建模，并选择最合适的算法来解决问题。我认为在今后，我们要时常做一些算法题保持手感与相应的思维，在不断地熟悉之后我相信我们的算法水平会有更大的进步。不畏惧算法，抽丝剥茧般去梳理算法的构建逻辑，我们才能将算法思想运用到未来的工程项目中，写出高质量且易于维护的代码。