1、电梯问题（elevator.cpp）

7.4.2 题目分析和算法实现

这是一道非常麻烦的模拟题。

可以按时间顺序来模拟，记录每一时刻电梯的状态，电梯的方向，电梯所在楼层，电梯中要去往各楼层的人数，以及各楼层向上和向下的队列。

电梯的状态有空闲、停止和移动三种。

（1）空闲时

a）没有请求时

什么都不做。

b）有请求时

i. 电梯所在楼层有请求

状态改为停止；

- 有向上的请求

  方向改为向上；

  向上的队列中第一个人进入电梯，直到电梯满载或者队列为空。

- 没有向上的请求

  方向改为向下；

  向下的队列中第一个人进入电梯，直到电梯满载或者队列为空。

ii. 电梯所在楼层没有请求

状态改为移动；

- 电梯所在楼层以上楼层有请求

  方向改为向上。

- 电梯所在楼层以下楼层有请求

  方向改为向下。

（2）停止时

a）电梯中有人要在电梯所在楼层出电梯

出电梯。

b）电梯中没有人要在电梯所在楼层出电梯

i. 有人可以进电梯

与电梯方向对应的队列中队头某些人进入电梯。

ii. 没有人可以进电梯

- 电梯上有人要继续走或者电梯的移动方向上有人在等电梯

  状态改为移动；

  改变电梯所在楼层。

- 否则

状态改为空闲。

（3）移动时

如果电梯上没有人要继续走且电梯的移动方向上没有人在等电梯，则电梯改变方向；

a）电梯所在楼层上，有人要出电梯或者有人可以进电梯

状态改为停止（注意：此时时刻不变）。

b）否则

电梯继续移动，改变电梯所在楼层。

7.4.3 参考程序及程序分析

```
#include <stdio.h>
#include <queue>
const int m = 50;
const int n = 105;
enum state1{idle, stop, move};          //电梯的状态：空闲、停止、移动
enum state2{up, down};                  //电梯当前的移动趋势：向上、向下

//node 表示在 t 时刻有一个人在 s 楼，想搭电梯去 d 楼
//qu 里保存了 t 递增的请求列表
struct node{
    int t, s, d;

}qu[n];
std::queue<node> qup[m + 1], qdown[m + 1];
                        //分别保存了电梯当前要处理的向上和向下的请求
int d[m + 1];           //电梯里要在各楼层出电梯的人数
int num;                //电梯里的人数
int p;                  //电梯限载人数

//把秒数 t 转换为时间格式输出
void print(int t){
    printf("%.2d:%.2d ", t / 60, t % 60);
}

//输出在时刻 t 向 f 楼移动，如果 k 为 1 则向上移动，否则为向下移动
void moves(int t, int f, int k){
    print(t);
    if (k == 1) printf("The elevator starts to move up from floor %d.\n", f);
    else printf("The elevator starts to move down from floor %d.\n", f);
}

//输出在时刻 t 停止在 f 楼
void stops(int t, int f){
```

```
    print(t);
    printf("The elevator stops at floor %d.\n", f);
}

//输出在时刻 t 有 k 人进电梯
void enter(int t, int k){
    print(t);
    printf("%d people enter the elevator.\n", k);
}

//输出在时刻 t 有 k 人出电梯
void leave(int t, int k){
    print(t);
    printf("%d people leave the elevator.\n", k);
}

//检查在电梯的移动方向上有没有要处理的请求，也就是 qup 或 qdown 队列里有没有请求
bool checkwait(int f, state2 st){
    if (st == up)    //如果电梯当前是向上移动的，则检查大于 f 的楼层上有没有要处理的请求
    {
        for (int i = f + 1; i <= m; i++)
            if (qup[i].size() > 0 || qdown[i].size() > 0) return true;
    }
    else             //如果电梯当前是向下移动的，则检查小于 f 的楼层上有没有要处理的请求
    {
        for (int i = f - 1; i > 0; i--)
            if (qup[i].size() > 0 || qdown[i].size() > 0) return true;
    }
    return false;
}

//检查电梯里有没有人在电梯的移动方向上出电梯
bool checkgo(int f, state2 st){
    if (st == up)
    {
        for (int i = f + 1; i <= m; i++) if (d[i] > 0) return true;
    }
    else
    {
        for (int i = f - 1; i > 0; i--) if (d[i] > 0) return true;
    }
    return false;
}
```

```
//检查电梯是否要在当前的移动方向上继续移动
bool checkcont(int f, state2 st){
    return checkwait(f, st) || checkgo(f, st);
}

//检查在电梯当前的移动方向下 f 楼有没有人能进电梯
bool checkenter(int f, state2 st){
    return num < p && (st == up && !qup[f].empty() || st == down && !qdown[f].empty());
}

//处理 t 时刻进电梯，保证进电梯后电梯里的人不超过限载人数
void doenter(int t, std::queue<node>& qu){
    int count = 0;
    while (num < p && !qu.empty())
                        //当电梯未满载且进电梯的队列不为空时让队列的第一个人进电梯
    {
        node x = qu.front();          //取出列队的第一个请求
        qu.pop();                     //将第一个请求从队列去掉
        count++;                      //进电梯的人数加 1
        num++;                        //电梯里的人数加 1
        d[x.d]++;                     //目的楼层为 x.d 的人数加 1
    }
    enter(t, count);
}

void solve(int f, int n){
    memset(d, 0, sizeof(d));
    num = 0;                          //当前电梯里的人数为 0
    state1 st1 = idle;                //电梯一开始为空闲状态
    state2 st2 = up;
    int t = 0, i = 0;                 //t 为当前的时刻，i 为还没处理的第一个请求
    while (i < n || st1 != idle)
    {
        if (st1 == idle) t = qu[i].t;
                    //当电梯的状态为空闲时，直接跳过空闲的时间，开始处理当前的请求
        while (i < n && qu[i].t == t)
                    //把请求时刻为 t 的请求放到 qup 和 qdown 队列中
        {
            if (qu[i].s < qu[i].d) qup[qu[i].s].push(qu[i]);
            else qdown[qu[i].s].push(qu[i]);
            i++;
        }
        if (st1 == idle)              //当电梯空闲时
        {
```

```
if (!qup[f].empty() || !qdown[f].empty())
                        //如果电梯所在楼层有请求则停下来，且优先处理向上的
{
    st1 = stop;
    st2 = !qup[f].empty() ? up : down;
    t--; //这里 t 减 1 是因为在后面 t 会统一加 1,而这里这种情况下 t 本来是不用变的
}
else                                    //如果本层没有请求
{
    st1 = move;
    int flag = checkwait(f, up) ? 1 : -1;
//先检查电梯以上的楼层是否有要处理的请求，如果没有说明电梯以下的楼层一定有请求
    st2 = flag == 1 ? up : down;
    moves(t, f, flag);
    f += flag;                          //电梯所在的楼层数加 1 或减 1
}
}
else if (st1 == stop)                   //如果电梯当前状态为停止
{
  if (d[f] > 0)                         //如果电梯里有人要在本层下
  {
    leave(t, d[f]);
    num -= d[f];                        //电梯里的人数减去 d[f]
    d[f] = 0;
  }
  else if (checkenter(f, st2)) //如果电梯当时要处理的请求中有要在本层进电梯的
  {
    if (st2 == up) doenter(t, qup[f]); //只让在 f 楼向上的人进电梯
    else doenter(t, qdown[f]);          //只让在 f 楼向下的人进电梯
  }
  else if (checkcont(f, st2)) //如果电梯需要在当前移动的方向上继续移动
  {
    int flag = st2 == up ? 1 : -1;
    moves(t, f, flag);
    f += flag;
    st1 = move;
  }
  else                                  //当前没有要处理的请求且不需要移动
  {
    st1 = idle;
  }
}
else                                    //如果电梯的状态为移动
{
```

```
            if (!checkenter(f, st2) && !checkcont(f, st2))
                                //如果在本层没有人能进且不需要在当前移动方向上继续移动
            {
                st2 = st2 == up ? down : up;          //转方向
            }
            if (d[f] > 0 || checkenter(f, st2))
                                //如果有人要出电梯或转方向后有人能进电梯
            {
                stops(t, f);
                st1 = stop;
                t--;
            }
            else if (st2 == up) f++;                  //继续向上移动
            else f--;                                 //继续向下移动
        }
        t++;
    }
}

int main(){
    freopen("elevator.in", "r", stdin);               //定义输入输出文件
    freopen("elevator.out", "w", stdout);
    int f, n;
    scanf("%d%d%d", &f, &n, &p);
                    //输入 0 时刻电梯所在的楼层，搭乘电梯的请求个数和电梯限载人数
    for (int i = 0; i < n; i++) scanf("%d%d%d", &qu[i].t, &qu[i].s, &qu[i].d);
    solve(f, n);
    return 0;
}
```

2、夜宵 1 号（gogogo.cpp）

样例解释

使用第 1, 3, 5 种移动方式，移动距离为 3.1+8.9+3.3=15.3。

数据范围

所有输入的实数最多有 3 位小数，绝对值小于 10^{20}，输出答案保留 3 位小数，误差在 0.001 以内均算正确。所有数据 $1 \leqslant n \leqslant 100$。

4.2.2 题目分析和算法实现

本题考查选手深度优先搜索加剪枝的能力。

题面虽然包含了很多内容，但是最后转化为解决一个背包问题：

给出一个实数集合，从中选出一个真子集，使得子集中的实数和超过一个给定常数，并且最小。

假如目标离原点距离为 d=sqrt(x*x+y*y)，手臂距离为 r。如果只使用一种移动方式，假设距离为 d_0，能送达的条件显然为 $d-r \leqslant d_0 \leqslant d+r$。而超过一种移动方式，则显然，它们能组合出来的最长距离为 Sum(d_i)，而最短距离为 Max{0, $2d_{max}$–Sum(d_i)}，其中 d_{max} 为当前移动方式中最长的一条的距离，Sum(d_i)为当前行动方式的总长度。于是我们知道送达条件为 $d-r \leqslant$ Sum(d_i)且 $d+r \geqslant 2d_{max}$–Sum(d_i)，即 Sum(d_i)$\geqslant$Max{$d-r$,$2d_{max}-d-r$}。可以看出，当选取的移动方式的最大值 d_{max} 确定后，这个问题就转化为上述背包问题了。

我们知道，背包问题是一个 NPC 问题。在某些特殊情况下，比如数域比较小，可以使用动态规划解决。但是本题的长度是实数，有效数字达 23 位之多，这一方法并不凑效。

那么，经过去冗化简后，我们面对这个光秃秃的搜索题目应该怎么处理呢？

基本思路就是采用深度优先搜索，加上若干剪枝。

- 长度和大于全局最优值。
- 长度和加上剩余的长度和小于 $d-r$（当 $d \geqslant d_{max}$）或 $2d_{max}-d-r$（当 $d<d_{max}$）。
- 全局最优值不可能更优，best=$d-r$。

除此之外，调整搜索顺序也能减少搜索时间。一种比较好的方式是将长度按由大到小排序。只要再增加一些特殊解的判断，就能通过评委设定的所有数据了。

4.2.3 参考程序及程序分析

```
#include <stdio.h>
#include <math.h>
#include <string.h>

const int maxn = 100;

double a[maxn];          //移动方式的长度
double sum[maxn];        //剩余距离和
double best;             //最优解
double d;                //欧拉距离减半径
double mind;             //搜索下界
```

```
double x, y, r;                    //目标圆的直角坐标和半径

int ans[maxn];                     //最优结果
int mrk[maxn];                     //标记某种移动方式是否被使用
int seq[maxn];                     //递归使用的当前结果
int n;                             //移动方式种类数


//从文件读取数据
void readdata()
{
    freopen("gogogo.in" , "r" , stdin);
    freopen("gogogo.out" , "w" , stdout);
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        scanf("%lf", &a[i]);
    }
    scanf("%lf%lf%lf", &r, &x, &y);
}

//深度搜索最优解，只搜索超过一种移动方式的方案
//dep: 深度，表示当前决策的移动方式编号
//count: 当前使用的移动方式个数
//cur: 当前长度和
void search(int dep, int count, double cur)
{
    if (cur >= best) {
        return;                                          //当前长度和大于最优解，剪枝
    }
    if (cur >= mind && count > 1) {                      //符合距离条件
        best = cur;
        memcpy(ans, mrk, sizeof(mrk));
        return;
    }
    if ((dep >= n)                                       //没有更多的移动方式
        || (cur + sum[dep] < mind)) { //当前长度和加上剩余长度和仍然不能符合条件
        return;
    }
    mrk[dep] = true;
    search(dep + 1, count + 1, cur + a[dep]);   //尝试使用第 dep 种移动方式
    mrk[dep] = false;
    if (best - d + r < 0.001) {
        return;                                          //不可能有更优解出现
    }
```

```
    search(dep + 1, count, cur);
}

/*对移动方式进行冒泡排序，长度由大到小，并且将排序出来对应的编号存进 seq 数组，供输出解的时
候使用*/
void sort()
{
    int i , j , k;
    double t;
    for (i = 0; i < n; ++i) {                    //初始化排列顺序
        seq[i] = i;
    }
    for (i = 0; i < n; ++i)
    for (j = i + 1; j < n; ++j)                  //冒泡排序
    if (a[i] < a[j]) {
        t = a[i];
        a[i] = a[j];
        a[j] = t;
        k = seq[i];
        seq[i] = seq[j];
        seq[j] = k;
    }
}

//求解函数
void solve()
{
    int i;
    sort();
//计算剩余长度和
    sum[n - 1] = a[n - 1];
    for (i = n - 2; i >= 0; --i) {
        sum[i] = sum[i + 1] + a[i];
    }
    double tot = sum[0];                         //总长度
    d = sqrt(x * x + y * y);
    if (tot < d - r) {
//所有移动方式都用了还是不能达到目标圆，则使用所有的移动方式
        printf("%.3lf\n", tot);
        for (i = 0; i < n; ++i) {
            printf("%d ", i + 1);
        }
        printf("\n");
        return;
```

```
    }
    if (d <= r) {                          //出发点在目标圆内，不需要移动
       printf("%.3f\n", 0.0);
       return;
    }

    best = 1e30;                           //初始最优值足够大
  //枚举只有一种移动方式时的解
    for (i = 0; i < n; ++i) {
       if (a[i] >= d - r && a[i] <= d + r && a[i] < best) {
                                           //如果通过第 i 种移动方式一次就能到达
          memset(ans, 0, sizeof(ans));
          ans[i] = true;
          best = a[i];
       }
    }

    memset(mrk, 0, sizeof(mrk));
    for (i = 0; i < n - 1; ++i) {
        mrk[i] = true;
        if (a[i] > d) mind = 2*a[i] - d - r; else mind = d - r;
                                           //用于第二个剪枝条件
        search(i+1, 1, a[i]);
        mrk[i] = false;
    }
    //输出最优解
    printf("%.3lf\n", best);               //最优的移动距离
    memset(mrk , 0 , sizeof(mrk));
    for (i = 0; i < n; ++i)
    if (ans[i]) {
       mrk[seq[i]] = true;
    }
    for (i = 0; i < n; ++i) {              //使用的移动方式
       if (mrk[i]) {
          printf("%d ", i + 1);
       }
    }
    printf("\n");

}

//主函数
int main()
{
    readdata();
    solve();
    return 0;
}
```