

P1081 [NOIP 2012 提高组] 开车旅行 ☆☆☆☆

题目描述

小 A 和小 B 决定利用假期外出旅行，他们将想去的城市从 1 到 n 编号，且编号较小的城市在编号较大的城市的西边，已知各个城市的海拔高度互不相同，记城市 i 的海拔高度为 h_i ，城市 i 和城市 j 之间的距离 $d_{i,j}$ 恰好是这两个城市海拔高度之差的绝对值，即 $d_{i,j} = |h_i - h_j|$ 。

旅行过程中，小 A 和小 B 轮流开车，第一天小 A 开车，之后每天轮换一次。他们计划选择一个城市 s 作为起点，一直向东行驶，并且最多行驶 x 公里就结束旅行。

小 A 和小 B 的驾驶风格不同，小 B 总是沿着前进方向选择一个最近的城市作为目的地，而小 A 总是沿着前进方向选择第二近的城市作为目的地（注意：本题中如果当前城市到两个城市的距离相同，则认为离海拔低的那个城市更近）。如果其中任何一人无法按照自己的原则选择目的城市，或者到达目的地会使行驶的总距离超出 x 公里，他们就会结束旅行。

在启程之前，小 A 想知道两个问题：

1、对于一个给定的 $x = x_0$ ，从哪一个城市出发，小 A 开车行驶的路程总数与小 B 行驶的路程总数的比值最小（如果小 B 的行驶路程为 0，此时的比值可视为无穷大，且两个无穷大视为相等）。如果从多个城市出发，小 A 开车行驶的路程总数与小 B 行驶的路程总数的比值都最小，则输出海拔最高的那个城市。

2、对任意给定的 $x = x_i$ 和出发城市 s_i ，小 A 开车行驶的路程总数以及小 B 行驶的路程总数。

输入格式

第一行包含一个整数 n ，表示城市的数目。

第二行有 n 个整数，每两个整数之间用一个空格隔开，依次表示城市 1 到城市 n 的海拔高度，即 $h_1, h_2 \dots h_n$ ，且每个 h_i 都是互不相同的。

第三行包含一个整数 x_0 。

第四行为一个整数 m ，表示给定 m 组 s_i 和 x_i 。

接下来的 m 行，每行包含 2 个整数 s_i 和 x_i ，表示从城市 s_i 出发，最多行驶 x_i 公里。

输出格式

输出共 $m + 1$ 行。

第一行包含一个整数 s_0 ，表示对于给定的 x_0 ，从编号为 s_0 的城市出发，小 A 开车行驶的路程总数与小 B 行驶的路程总数的比值最小。

接下来的 m 行，每行包含 2 个整数，之间用一个空格隔开，依次表示在给定的 s_i 和 x_i 下小 A 行驶的里程总数和小 B 行驶的里程总数。

输入输出样例

输入 #1	输出 #1
4	1
2 3 1 4	1 1
3	2 0
4	0 0
1 3	0 0
2 3	
3 3	
4 3	

输入 #2	输出 #2
10	2
4 5 6 1 2 3 7 8 9 10	3 2
7	2 4
10	2 1
1 7	2 4
2 7	5 1
3 7	5 1
4 7	2 1
5 7	2 0
6 7	0 0
7 7	0 0
8 7	
9 7	
10 7	

算法分析

本题数据范围 $n, m \leq 10^5$, 且 x 可达 10^9 , 简单的模拟会超时。解题分为两步：预处理“下一跳”和倍增优化模拟。

1. 预处理：寻找目标城市 ($O(N \log N)$)

题目要求只能向东走，且寻找海拔最近和次近的城市。我们可以[倒序遍历](#)城市（从 n 到 1），维护一个包含当前城市以东所有城市海拔的集合。

- 使用 `set` 存储 `(海拔, ID)`, 并预先插入哨兵值以避免边界判断。

```
set<array<int, 2>> st;
st.insert({inf, n}); st.insert({inf + 1, n});
st.insert({-inf, n}); st.insert({-inf - 1, n});
```

- 对于城市 i (倒序遍历), 在 `set` 中查找第一个海拔 $\geq h_i$ 的元素位置, 并将该位置前后相邻的 4 个元素 (前 2 个, 后 2 个) 设置为候选点。

```

for (int i = n - 1; i >= 0; i--) {
    vector<int> vec;
    auto it = st.lower_bound({h[i]}, i);
    auto tmp = it;

    // lambda: 将合法候选点加入 vec
    auto put = [&](array<int, 2> x) {
        if (x[1] == n) return;
        vec.push_back(x[1]);
    };
    put(*tmp++); put(*tmp--); // 检查后两个
    put(*--tmp); put(*--tmp); // 检查前两个
}

```

- 对这 4 个候选点按照题目规则 (和城市 i 距离优先, 海拔其次) 排序, 即可得到小 B 的目标 (最近) 和小 A 的目标 (次近), 并记录在数组中。

```

// 排序: 距离近优先, 海拔低优先
sort(vec.begin(), vec.end(), [&](int &a, int &b) {
    int dis_a = get_dis(a, i);
    int dis_b = get_dis(b, i);
    if (dis_a == dis_b) return h[a] < h[b];
    return dis_a < dis_b;
});

if (vec.size() >= 2) nxt_a[i] = vec[1]; // A去第二近
if (vec.size() >= 1) nxt_b[i] = vec[0]; // B去最近
st.insert({h[i], i});
}

```

2. 倍增 (Binary Lifting) 优化 ($O(N \log N)$)

由于 A 和 B 是轮流走的, 我们将“A 走一步 + B 走一步”看作一个完整的回合。

- 初始化 ($j = 0$): 计算 A 走一步 + B 走一步后的到达位置及各自路程。

```

for (int i = 0; i < n; i++) { // 初始化 j=0 (A+B 各走一步)
    int a = nxt_a[i]; int b = nxt_b[a];
    if (b == n) continue; // 无法完成一轮
    dis_a[i][0] = get_dis(i, a);
    dis_b[i][0] = get_dis(a, b);
    dp[i][0] = b;
}

```

- 状态转移: 利用 $2^j = 2^{j-1} + 2^{j-1}$ 的性质, 递推计算倍增表。

- 从城市 i 先走 2^{j-1} 轮, 到达中转点 $mid = dp[i][j-1]$;
- 然后从中转点 mid 再走 2^{j-1} 轮, 到达最终目的地 $dp[i][j] = dp[mid][j-1]$ 。

- 将 mid 替换展开，即得到核心递推式：

$$dp[i][j] = dp[dp[i][j-1]][j-1]$$

```
// 倍增递推
for (int j = 1; j < 32; j++) {
    for (int i = 0; i < n; i++) {
        int mid = dp[i][j - 1];
        if (mid == n) continue;
        int end = dp[mid][j - 1];
        if (end == n) continue;

        dis_a[i][j] = dis_a[i][j - 1] + dis_a[mid][j - 1];
        dis_b[i][j] = dis_b[i][j - 1] + dis_b[mid][j - 1];
        dp[i][j] = end;
    }
}
```

3. 处理询问 ($O(M \log N)$)

对于给定的起点 s 和限额 x ，我们利用倍增数组快速跳转：

- (1) **倍增跳跃：**从最高位（如 $j = 31$ ）向下枚举。如果走 2^j 轮的总距离（A+B）不超过剩余 x ，则跳过去，累加路程并更新当前位置。

```
int a = 0, b = 0;
// 倍增跳跃 (A+B 成对跳)
for (int i = 31; i >= 0; i--) {
    int _a = dis_a[beg][i];
    int _b = dis_b[beg][i];
    // 如果能跳且不超距离
    if (_a + _b > x || dp[beg][i] == n) continue;

    a += _a; b += _b;
    x -= _a + _b;
    beg = dp[beg][i];
}
```

- (2) **最后一步 A 的处理：**倍增只处理了完整的轮次。循环结束后，可能剩余的 x 还足够小 A 单独再走一步（但不够 B 走了），需要特判加上这一步。

```
// 特判 A 的最后一步
if (nxt_a[beg] != n && get_dis(nxt_a[beg], beg) <= x) {
    a += get_dis(nxt_a[beg], beg);
}
return {a, b};
```

4. 细节处理

- 第一问比较：比值 $\frac{A}{B}$ 的比较应小心处理 $B = 0$ 的无穷大情况。若比值相同，需输出海拔最高的。

```
int s0 = 0;
long double tot = inf + 1;
for (int i = 0; i < n; i++) {
    auto tmp = cul(i, x0); // 计算从 i 出发，限距 x0 的结果
    int a = tmp[0]; // 小a走的距离
    int b = tmp[1]; // 小b走的距离

    long double _tot = (b == 0 ? inf : (a + 0.0) / b);
    if (tot > _tot || (tot == _tot && h[i] > h[s0])) {
        s0 = i;
        tot = _tot;
    }
}
```

解题总结与核心考点

本题是一道结合了数据结构预处理与倍增优化的经典提高组试题，主要考察点如下：

1. 倍增 (Binary Lifting) 思想的应用：

- 常规的模拟在 X 很大时会超时。利用倍增，我们将“走 K 步”拆分为若干个 2^i 步的组合，将单次查询复杂度从 $O(N)$ 降至 $O(\log N)$ 。
- 本题的特殊性在于 A 和 B 轮流移动，因此定义 $dp[i][j]$ 为“A和B各走 2^j 步”是一个处理不对称移动的经典技巧。

2. 利用 STL Set 进行邻居查找：

- 在一维坐标轴上动态查找前驱和后继（最近和次近），利用 `std::set` 的有序性和 `lower_bound` 是标准解法。
- 哨兵技巧：在 Set 中预先插入极值 (INF)，可以极大地简化边界条件的判断代码。

3. 细节与边界处理：

- 残余步数：倍增只能处理成对的步数，循环结束后 A 可能还能单独走一步，这是极其容易遗漏的边界。
- 精度与除零：在比较 A/B 比值时，需注意 $B = 0$ 的情况，以及浮点数比较的精度问题（或转换为乘法比较）。