算法设计与分析-上机实验作业

吴天阳 2204210460 强基数学 002

1 问题一

给定由 n 个数组成的集合 S，从中找出与 S 的中位数最近的 $n/4$ 个元素，请设计一个最坏情况下时间复杂度为 $\mathcal{O}(n)$ 的算法.

1.1 问题分析

首先要实现**线性时间选择**代码，即 `select(l, r, k)` 函数，在 $\mathcal{O}(n)$ 时间复杂度下找到数组 `a[l,...,r]` 中第 k 大的数字. 当 $k = \lfloor (n+1)/2 \rfloor$ 时就是查找中位数.

1.1.1 randomizedSelect 方法

考虑快排的思路，我们考虑划分函数 `partition(l,r)` 对数组 `a[l,...,r]` 进行划分：随机选取一个元素 `a[i]` 作为基准元素，然后将 `a[l,...,r]` 中小于等于 `a[i]` 的元素移动到左侧，大于等于 `a[i]` 的元素移动到右侧，通过该方法，可以得到小于等于 `a[i]` 的元素个数记为 m.

假设当前查询的第 k 大的数字，如果 $k \leqslant m$ 则在 `a[l,...,m]` 中继续查找第 k 大的元素，否则在 `a[m+1,...,r]` 中查找第 $k-m$ 大的元素，该操作可以递归完成.

该时间复杂度平均为 $\mathcal{O}(n)$，但是如果每次选取的最小的元素作为基准，而我们查询的是最大的元素，则时间复杂度最差可能达到 $\mathcal{O}(n^2)$，于是我们需要对其进行改进.

1.1.2 select 方法

在 `randomizedSelect` 方法的基础上，我们发现由于每次是随机选取的一个基准，该基准可能较差 (过于偏向左侧或右侧)，所以我们期望选择一个较为靠中间的元素，这样就能保证每次必定减少 n 的某一个量级，从而稳定最坏复杂度.

假设当前处理的序列长度为 n，序列为 `a[1,...,n]`，具体思想就是找当前序列的**中位数的中位数**作为基准.

具体做法: 我们假设以 5 个元素作为一个小组, 将原数组划分为 $\lfloor n/5 \rfloor$ 个模 5 的剩余类, 例如 $n = 17$, 则划分后的结果为 `[*****|*****|*****|**]`(`*`表示数组中的元素, 一共得到 3 个小组)，然后我们对每个小组中的元素使用冒泡排序，冒泡 3 次即可找到小组中的中位数, 我们将中位数标记为 `$` 符号, 则排序后数组为 `[**$**|**$**|**$**|**]`. 然后我们将每个小组的中位数全部移动到整个数组的左侧，即 `[$$$**|*****|*****|**]`，于是我们已经获得了中位数序列，即数组中开头三个，为了求解中位数的中位数，我们再递归调用 `select` 函数. 以上面例子为例，我们只需求解 `select(1,3,2)` 从而求解 `[$$$]` 中的中位数，即可得到原数组的中位数的中位数.

获得中位数的中位数 `x` 后，再以 `x` 作为基准元素，利用类似快排的划分函数 `partition(l,r,x)`(只不过这次给定了基准元素 `x`)，从而对原数组左右划分，再判断我

们要查询的第 k 大元素是在基准的左侧还是右侧，递归查找即可.

注：如果有存在多个重复基准元素 `x`，我们需要将划分后的**基准元素全部聚集在中间**，假设有 m 个重复基准元素，于是原数组最终应该划分为 `[00000|$$$$|11111]`，其中`0`、`1`分别表示小于、大于基准元素的值，`$` 表示基准元素. 于是原数组被划分为三分，如果当前第 k 大元素落在中间的区间中，则直接返回基准元素，否则判断第 k 大元素在左侧还是右侧，递归查找即可.

上述算法那的关键就是找到了当前序列的中位数的中位数，从而每次划分为两半时，基准元素左侧至少会有 $\lfloor n/4 \rfloor$ 个元素. 保证每次查找至少可以减少 $\lfloor n/4 \rfloor$ 的数量级.

理论复杂度分析，设对序列长度为 n 的序列调用 `select` 函数需要 $T(n)$ 的时间，则查找中位数的中位数至多使用 $T(n/5)$，使用基准 x 划分原数组，两个数组中至多还有 $3n/4$ 个元素，则进一步递归调用至多使用 $T(3n/4)$ 时间. 综上，$T(n)$ 递推式为

$$T(n) = Cn + T(n/5) + T(3n/4),\ (C \approx 3)$$

常数 C 主要包含每个小组的冒泡排序、`partition`、聚集基准元素所花的时间.

1.1.3 查找距离中位数最近的 k 个数

有了 `select` 函数，这个问题就非常容易解决了，首先找到原数组的中位数 `mid`，然后求原数组与 `mid` 的绝对值之差，存为数组 `b[]`，然后查找数组 `b[]` 的第 k 大元素 `delta`，那么说明和 `mid` 相差 `delta` 距离的元素就是距离中位数 `mid` 最近的 k 个元素，最后搜索一遍原数组，将满足 $|a[i] - mid| \leqslant delta$ 的元素输出出来即可.

1.2 算法实现

```cpp
const int N = 1e8;

int n, k, a[N], b[N], tmp[N];

void bubble(int l, int r) {  // 对 a[l,r) 进行冒泡排序
    for (int i = l; i < r; i++)
        for (int j = i + 1; j < r; j++)
            if (a[i] > a[j]) swap(a[i], a[j]);
}
// 划分函数，返回值为：最靠左的 mid 下标，与 mid 相同数个数
pair<int, int> partition(int l, int r, int mid) {
    int i = l - 1, j = r;
    while (1) {
        while (a[++i] < mid && i < r);
        while (a[--j] > mid && j >= l);
        if (i >= j) break;
        swap(a[i], a[j]);
    }
```

```cpp
    // 例 mid=5, [0 3 2 5 5 5 5 14 12 11], ll=2, rr=7
    int ll = j, rr = j+1;  // 与 mid 相同数合并称一块
    for (int i = 0; i < ll; i++) {
        while (a[ll] == mid && ll > l) ll--;
        if (a[i] == mid) swap(a[i], a[ll]), ll--;
    }
    for (int i = r-1; i > rr; i--) {
        while (a[rr] == mid && rr < r-1) rr++;
        if (a[i] == mid) swap(a[i], a[rr]), rr++;
    }
    return make_pair(ll+1, rr-ll-1);
}

int select(int l, int r, int k) {  // 返回 a[l,r) 中第 k 大的元素
    if (r - l < 5) { // 如果元素个数小于 5
        bubble(l, r);
        return a[l+k-1];
    }
    // [00000|00000|00000|000]  以五个进行一个划分，每个子区间长度为 5
    for (int i = 0; i < (r - l) / 5; i++) {
        int s = l + 5 * i, t = s + 5;  // 处理子区间 [s,t)
        for (int j = s; j < s+3; j++)  // 仅需做 3 次冒泡
            for (int k = j+1; k < t; k++)
                if (a[j] > a[k]) swap(a[j], a[k]);
        swap(a[l+i], a[s+2]);  // 将 [s,t) 的中位数移动到数列开头
    }
    int x = select(l, l+(r-l)/5, (r-l+5)/10);  // 递归找到中位数的中位数
    auto p = partition(l, r, x);
    int mid = p.first, same = p.second, less = mid-l;  // 以 mid 作为快排基准进行排序
    if (k <= less) return select(l, mid, k);  // 在左半区间
    else if (k <= less + same) return x;  // 在中间区间就是中位数
    return select(mid+same, r, k-less-same);  // 在右半区间
}

int main() {  // 求解距离中位数 n/4 近的数
    cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];
    memcpy(tmp, a, sizeof(int) * n);
    int mid = select(0, n, (n+1)/2);  // 先求出中位数
    cout << "mid: " << mid << '\n';
    for (int i = 0; i < n; i++) a[i] = abs(a[i] - mid);  // 求出绝对值数组
    int k = select(0, n, n/4), tot = n/4;  // 绝对值数组中前 n/4 分位数
```

```
    memcpy(a, tmp, sizeof(int) * n);
    cout << "Around Mid: ";
    for (int i = 0; i < n && tot; i++)  // 再从原数组中找绝对值差小于等于 k 的
        if (abs(a[i] - mid) <= k) {
            cout << a[i] << ' ';
            tot--;
        }
    cout << '\n';

    // 将原数组排序，检查结果是否正确
    cout << "\n" << "After sorted(Check): " << '\n';
    sort(a, a + n);
    for (int i = 0; i < n; i++)
        cout << a[i] << ' ';
    cout << '\n';
    return 0;
}
```

1.3 测试结果与性能分析

测试结果 (两个测试数据 Input 为输入数据，Output 为输出结果，mid 为中位数，Around Mid 为中位数最近的 $n/4$ 个值):

```
Input:
9
1 2 2 3 4 5 5 8 9
Output:
mid: 4
Around Mid: 3 4

Input:
20
34 26 98 31 26 88 90 39 68 95 80 78 69 7 3 48 32 39 9 63
Output:
mid: 39
Around Mid: 34 31 39 32 39
```

为了和直接排序的速度进行比较，我分别构造了 $n = 10^7$ 和 $n = 10^8$ 的序列长度，然后分别利用 select 函数和直接排序输出第 k 大元素速度进行了比较，代码如下

```
int main() {  // 用于测试 select 函数速度
    ios::sync_with_stdio(0);
    cin.tie(0);
    freopen("in.in", "r", stdin);  // 数据读入
```

```
    cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];
    clock_t start = clock();
    cout << "My Answer: " << select(0, n, 10) << '\n';
    clock_t end = clock();
    cout << "Use Time: " << end - start << "ms" << '\n';

    int *tmp = new int[sizeof(a)/sizeof(int)];
    memcpy(tmp, a, sizeof(a));
    start = clock();
    sort(a, a + n);
    cout << "Check: " << a[10-1] << '\n';
    end = clock();
    cout << "Use Time: " << end - start << "ms";
    return 0;
}
```

测速速度结果为:

$n = 10^7$ 结果 select 函数 697ms，直接排序所用时间 1481ms.

$n = 10^8$ 结果 select 函数 5866ms，直接排序所用时间 12523ms.

可以看出来 select 函数快了一倍左右，效果不错.

1.4 数据生成代码

```
#include <iostream>
#include <time.h>
using namespace std;

int main() {  // 7-1 随机生成数据
    srand(time(NULL));  // 随机种子
    freopen("7-1.in", "w", stdout);  // 输出文件到"7-1.in" 中
    int n = 1e7;  // 序列总长度
    cout << n << '\n';
    for (int i = 0; i < n; i++) {
        printf("%d ", rand() % 100);
    }
    cout << '\n';
    return 0;
}
```

2 问题二

设 `A[1..M]`, `B[1..N]`,`C[1..L]` 是三个任意序列，A、B、C 的公共超序列定义为一个包含 A、B、C 为子序列的序列. 请设计实现一个动态规划算法，找出 A、B、C 的最短公共超序列.

2.1 问题分析

2.1.1 二维最短共超序列

考虑两个序列 `a[1,...,n]`,`b[1,...,m]` 的最短共超序列,设二维动态规划数组 `dp[i][j]` 表示 `a[1,...,i]` 和 `b[1,...,j]` 的最短共超序列长度，有如下状态转移方程：

$$dp[i,j]=\begin{cases} dp[i-1,j-1]+1, & a[i]=b[j],\\ \min\left\{dp[i,j-1],dp[i-1,j]\right\}+1, & a[i]\neq b[j].\end{cases}$$

初始化：$dp[i,0]=i,\ (i=1,2,\cdots,n), dp[0,j]=j,\ (j=1,2,\cdots,m)$.
设计思路：若 $a[i]=a[j]$，则 `(i,j)` 的最短共超序列可以在 `(i-1,j-1)` 的最短共超序列基础上加上 `a[i]`; 若 $a[i]\neq a[j]$，则从 `(i-1,j)` 的最短共超序列基础上加上 `a[i]`，也可以从 `(i,j-1)` 的最短共超序列基础上加上 `b[j]` 得到，所以取 `dp[i][j-1]`,`dp[i-1][j]` 中较小者进行转移得到.

生成最短共超序列的方法：通过数组 `fa[i][j]` 记录每次 `dp[i][j]` 的转移来自于谁,于是也可以得到每次需要加上哪一个元素. 例如:记 `fa=0,1,2`分别表示从 `dp[i-1][j-1]`, `dp[i-1][j]`, `dp[i][j-1]` 转移得到的,那么对应增加的元素分别是 `a[i]`, `a[i]`, `b[j]`(第一个也可以是 `b[i]`，因为两者相同).

然后从 `dp[n][m]` 开始逆推，逆推方向是 `dp[i][j]` 转移方向，还有 `(i,j)` 处需增加的元素，两者均可通过 `fa[i][j]` 的取值得到.

二维最短共超序列求解可以参考 CSDN - 最短公共超序列（最短公共父序列）.

2.1.2 三维最短共超序列

有了二维的最短共超序列求解办法，我们进一步讨论三维最短共超序列的求解问题. 考虑三个序列 `a[1,...,n]`, `b[1,...,m]`, `c[1,...,l]` 的最短共超序列. 设三维动态规划数组 `dp[i][j][k]` 表示 `a[1,...,i]`,`b[1,...,m]`,`c[1,...,l]` 的最短共超序列，有如下状态转移方程：

$$dp[i,j,k]=\begin{cases} dp[i-1,j-1,k-1]+1, & a[i]=b[j]=c[k],\\ dp[i-1,j-1,k]+1, & a[i]=b[j],\\ dp[i-1,j,k-1]+1, & a[i]=c[k],\\ dp[i,j-1,k-1]+1, & b[j]=c[k],\\ \min\left\{dp[i-1,j,k],dp[i,j-1,k],dp[i,j,k-1]\right\}+1 & \text{否则}.\end{cases}$$

初始化：$dp[i,j,0], dp[i,0,k], dp[0,j,k]$，$(i \in [1,n], j \in [1,m], k \in [1,l])$ 分别是二维最短共超序列的解，也就是说 $dp[i,j,0] = dp'[i,j]$，$dp'[i,j]$ 是 `a[1,...,i],b[1,...,j]` 的最短共超序列的解. (**2.1.1** 中所介绍的方法求解)

三维最短共超序列的状态转移方程设计思路与二维完全一致，这里不详细解释，反向求解最优解过程也完全一致，只是把二维问题换为三维，`fa` 数组的取值变为 7 个.

2.1.3 不能将三维问题转化为两个二维问题求解

这里说明不能通过分别求解数组 `a,b` 的最短共超序列 `d`，再求解 `d,c` 的最短共超序列，从而得到 `a,b,c` 的最短共超序列.

反例：设 `a="abed"`，`b="ecaa"`，`c="eacd"`，则 `d="abedcaa"`，于是得到错误的最短共超序列为`"abedcaacd"`，长度为 9；而最优解应该为`"ecabeacd"`，长度为 8.

2.2 算法分析

主要是设计如何在三维数组中计算二维的最短共超序列，记少掉的维度为 `dim`，由于二维中 `fa` 数组只有 3 种取值，而三维中有 7 中取值，所以降到二维后还需将 `fa` 数组进行对应，记为 `idx[3]`. 于是就有了下述代码中 12 到 23 行的内容.

```
1  #include <iostream>
2  #include <string.h>
3  #include <algorithm>
4  using namespace std;
5  const int N = 310;
6  char s[3][N];  // 初始字符串
7  int dp[N][N][N], fa[N][N][N]; // fa={0,...,6} 分别表示 7 个转移方向，具体方向如下面所定义
8  int dx[7] = {-1, -1, -1, 0, -1, 0, 0};
9  int dy[7] = {-1, -1, 0, -1, 0, -1, 0};
10 int dz[7] = {-1, 0, -1, -1, 0, 0, -1};
11 
12 // dim 为排除的维数，idx[0,1,2] 分别为返回 a-1,b-1;a-1;b-1 的转移方向编号
13 int dim, idx[3];
14 // 返回 a 数组在排除 dim 维度后的 i,j 对应指针
15 int* get_prt(int a[][N][N], int i, int j) {
16     if (dim == 0) return &a[0][i][j];
17     else if (dim == 1) return &a[i][0][j];
18     return &a[i][j][0];
19 }
20 // 返回数组 a 在排除 dim 维度后的 i,j 元素值
21 int get(int a[][N][N], int i, int j) {return *get_prt(a, i, j);}
22 // 设置数组 a 在排除 dim 维度后的 i,j 元素值
23 void set(int a[][N][N], int i, int j, int x) {*get_prt(a, i, j) = x;}
```

```
// 处理 a,b 的最短共超序列
void subsolve(char a[], char b[]) {
    int l1 = strlen(a+1), l2 = strlen(b+1);
    for (int i = 1; i <= l1; i++) {
        for (int j = 1; j <= l2; j++) {
            if (a[i] == b[j]) {
                set(dp, i, j, get(dp, i-1, j-1) + 1);
                set(fa, i, j, idx[0]);
            } else if (get(dp, i-1, j) < get(dp, i, j-1)) {
                set(dp, i, j, get(dp, i-1, j) + 1);
                set(fa, i, j, idx[1]);
            } else {
                set(dp, i, j, get(dp, i, j-1) + 1);
                set(fa, i, j, idx[2]);
            }
        }
    }
}

// 返回 a,b,c 的最短公共超序列
string solve(char a[], char b[], char c[]) {
    // 由于下标从 1 开始，所以求长度也需要指针 +1
    int l1 = strlen(a+1), l2 = strlen(b+1), l3 = strlen(c+1);
    // 初始化 dp 数组
    for (int i = 1; i <= l1; i++) dp[i][0][0] = i, fa[i][0][0] = 4;
    for (int i = 1; i <= l2; i++) dp[0][i][0] = i, fa[0][i][0] = 5;
    for (int i = 1; i <= l3; i++) dp[0][0][i] = i, fa[0][0][i] = 6;
    // 分别求解三个二维维度上的子问题
    dim = 2, idx[0] = 1, idx[1] = 4, idx[2] = 5;
    subsolve(a, b);
    dim = 1, idx[0] = 2, idx[1] = 4, idx[2] = 6;
    subsolve(a, c);
    dim = 0, idx[0] = 3, idx[1] = 5, idx[2] = 6;
    subsolve(b, c);
    for (int i = 1; i <= l1; i++) {
        for (int j = 1; j <= l2; j++) {
            for (int k = 1; k <= l3; k++) {
                if (a[i] == b[j] && b[j] == c[k]) {
                    dp[i][j][k] = dp[i-1][j-1][k-1] + 1;
                    fa[i][j][k] = 0;
                } else if (a[i] == b[j]) {
                    dp[i][j][k] = dp[i-1][j-1][k] + 1;
```

```
                    fa[i][j][k] = 1;
                } else if (a[i] == c[k]) {
                    dp[i][j][k] = dp[i-1][j][k-1] + 1;
                    fa[i][j][k] = 2;
                } else if (b[j] == c[k]) {
                    dp[i][j][k] = dp[i][j-1][k-1] + 1;
                    fa[i][j][k] = 3;
                } else {
                    int tmp[] = {dp[i-1][j][k], dp[i][j-1][k], dp[i][j][k-1]};
                    if (tmp[0] < max(tmp[1], tmp[2])) {
                        dp[i][j][k] = tmp[0] + 1;
                        fa[i][j][k] = 4;
                    } else if (tmp[1] < max(tmp[0], tmp[2])) {
                        dp[i][j][k] = tmp[1] + 1;
                        fa[i][j][k] = 5;
                    } else {
                        dp[i][j][k] = tmp[2] + 1;
                        fa[i][j][k] = 6;
                    }
                }
            }
        }
    }
    string ret;
    for (int i = l1, j = l2, k = l3; i || j || k;) {
        int f = fa[i][j][k];
        if (f == 0) ret.push_back(a[i]);
        else if (f == 1) ret.push_back(a[i]);
        else if (f == 2) ret.push_back(a[i]);
        else if (f == 3) ret.push_back(b[j]);
        else if (f == 4) ret.push_back(a[i]);
        else if (f == 5) ret.push_back(b[j]);
        else if (f == 6) ret.push_back(c[k]);
        i += dx[f], j += dy[f], k += dz[f];
    }
    reverse(ret.begin(), ret.end());  // 得到最长公共子序列 lcs
    return ret;
}

int main() {
    for (int i = 0; i < 3; i++) cin >> s[i]+1;  // 输入三个字符串，下标从 1 开始
```

```
    clock_t start = clock();
    string ans = solve(s[0], s[1], s[2]);
    clock_t end = clock();
    cout << "My Answer: " << ans << '\n';
    cout << "Length: " << ans.size() << '\n';
    cout << "Time: " << end - start << " ms";
    return 0;
}
```

2.3 测试结果与性能分析

算法总时间复杂度为 $\mathcal{O}(nml)$，空间复杂度也为 $\mathcal{O}(nml)$，预计处理每个序列长度均不超过 300(内存允许可以开到 600，程序里写的数组最大只有 300，可以修改更大的大小)，随机生成三个长度均为 300 的序列，计算用时为 483ms；长度均为 600 的序列，计算用时为 3927ms. 以下为几个小样例的运行结果：

```
Input:                              Input:
abed                                abeadeac
ecaa                                ecaacacc
eacd                                eacebaed
Output:                             Output:
My Answer: ecabeacd                 My Answer: eacebaeacdeacc
Length: 8                           Length: 14

Input:
baddbdceacaecaaabeda
dadadbeeacbeecddabce
abebcbeeaeccbeeaddad
Output:
My Answer: badadabebdcbeeaecacbeecaaddabceda
Length: 33
```

2.3.1 数据生成代码

```
#include <iostream>
#include <time.h>
using namespace std;

int main() {  // 7-2 随机生成数据
    srand(time(NULL));
    freopen("7-2.in", "w", stdout);
    int n = 20;  // 每个序列的长度均为 n
    string s[3];
```

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < n; j++)
            s[i].push_back(rand() % 10 + 'a');
        cout << s[i] << '\n';
    }
    return 0;
}
```

3 问题三

给定二维平面上 n 个点对 $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$，假定所有点之间的横纵坐标均不相同. 对于任意两个点 (x_i, y_i) 和 (x_j, y_j)，若 $x_i > x_j$ 且 $y_i > y_j$，则称 (x_i, y_i) 支配 (x_j, y_j). 不被任何其他点支配的点称为 maxima. 请设计一个贪心算法，在 $\mathcal{O}(n \log n)$ 时间内找出所有的 maxima.

3.1 问题分析

可以先通过对点的 x 轴作为排序基准，不妨令所有点已经按 x 轴从小到大排序，即点列满足 $\{(x_i, y_i) : x_i < y_j,\ i < j\}$. 由于一个点 (x_i, y_i) 是 maxima 点，当且仅当，不存在 $j = 1, 2, \cdots, n$ 使得 $x_j > x_i$ 且 $y_j > y_i$. 显然，点 (x_n, y_n) 一定是 maxima 点.

再考虑第 i 个点，由于点列已经按 x 轴排序，则 (x_i, y_i) 所有可能的支配点 (x_j, y_j) 均在 $j > i$ 中，所以只需再考虑是否存在 $y_j > y_i$ 即可判断是否存在支配点，而我们只需关心这样最大的 y_j 即可，也就是说若 $y_i < \max\limits_{j>i} y_j$，则 (x_i, y_i) 不是 maxima 点，反之，(x_i, y_i) 是 maxima 点.

3.2 算法分析

我们只需先对原数组进行排序，然后逆序遍历，记录 y 轴的最大值，根据上述判断即可找到 maxima 点.

```
#include <iostream>
#include <utility>
#include <algorithm>
using namespace std;

const int N = 1e6 + 10;
int n;
pair<int, int> a[N];  // 利用 pair 存储点对坐标

int main() {
    cin >> n;
```

```
    for (int i = 0; i < n; i++) cin >> a[i].first >> a[i].second;
    sort(a, a+n);  // pair 二元组自带比较关系，优先 x 从小到大
    int mx = 0;  // 记录最大 y 值
    cout << "maxima: " << '\n';
    for (int i = n-1; i >= 0; i--) {
        if (a[i].second < mx) continue;
        mx = max(mx, a[i].second);
        cout << a[i].first << ' ' << a[i].second << '\n';
    }
    return 0;
}
```

3.3 测试结果与性能分析

由于只是用了一次排序，所以总时间复杂度为 $\mathcal{O}(n\log n)$. 简单生成了 10 个点的数据，绘制成图像如下所示

输入数据：

10

8 7

6 4

9 5

1 2

5 1

0 0

4 9

3 8

2 6

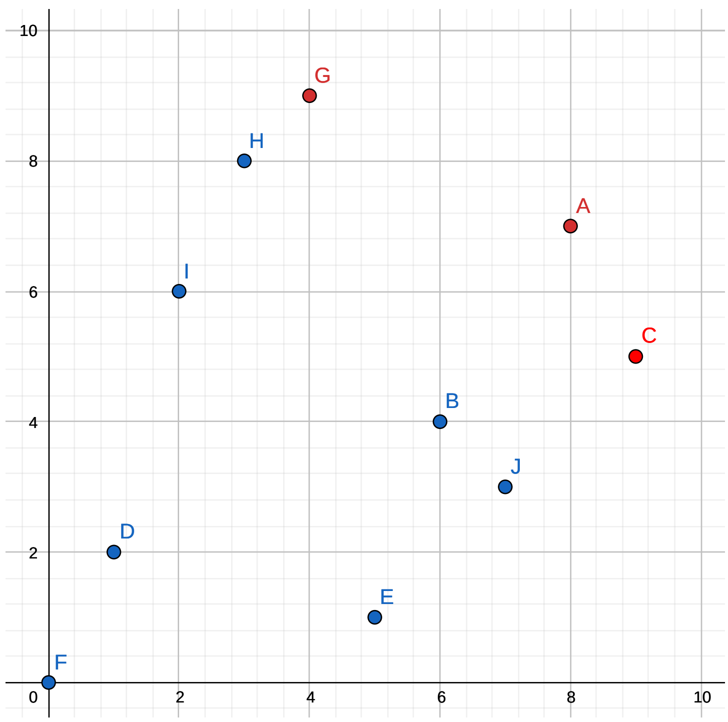
7 3

输出结果：

9 5

8 7

4 9



红色节点表示 maxima 节点.

3.3.1 数据生成代码

```
int x[1000000], y[1000000];
int main() {  // 7-3 随机生成数据
    srand(time(NULL));
    freopen("7-3.in", "w", stdout);
    int n = 10;  // 总点数目，生成点范围在 [0,...,n), [0,...,n) 中间，保证两两
↪   之间横纵坐标不同
```

```
    cout << n << '\n';
    for (int i = 0; i < n; i++) x[i] = y[i] = i;
    random_shuffle(x, x + n);
    random_shuffle(y, y + n);
    for (int i = 0; i < n; i++) cout << x[i] << ' ' << y[i] << '\n';
    return 0;
}
```

4 问题四

设有 a, b, c, d, e 五个整数和+,-,*,/四个运算符. 对于任意给定的整数 n，请设计一个算法，用给定的 5 个整数生成一个算术表达式 (每个整数和每个运算符只能使用 1 次)，使其运算结果等于 n.

4.1 问题分析

考虑直接递归枚举所有可能的算术表达式形式，时间复杂度为 $O(5!4!)$，详细计算全部算术表达式总数为

$$5 + (5 \times 4) \times 4 + (5 \times 4 \times 3) \times (4 \times 3) + (5 \times 4 \times 3 \times 2) \times (4 \times 3 \times 2) + 5 \times 4 = 6565$$

4.2 算法分析

直接使用递归枚举全部算术表达式的可能性，再使用逆波兰表达式计算每个算术表达式的结果，若与 n 相等则输出结果.

```
#include <iostream>
#include <stack>
#include <map>
using namespace std;

// use[][0] 表示每个数是否使用过，uses[][1] 表示每个符号是否用过
bool use[5][2];
// now[i] 表示结果中第 i 位的值，i 为偶数则是数字，反之为运算符
int now[9];
// 存储全部解
map<int, int> mp;

int n, a[5], rk[256];
int opt[] = {'+', '-', '*', '/'};
stack<string> ans;

```

```
void execute(stack<int> &stk, int opt) {
    int b = stk.top(); stk.pop();
    int a = stk.top(); stk.pop();
    if (opt == '+') stk.push(a+b);
    if (opt == '-') stk.push(a-b);
    if (opt == '*') stk.push(a*b);
    if (opt == '/') stk.push(a/b);
}

int calc(int x) {  // 利用逆波兰表达式求解当前计算式 now[] 的值
    stack<int> stk_num, stk_opt;
    for (int i = 0; i < x; i++) {
        if (i % 2 == 0) stk_num.push(now[i]);  // 数字
        else {  // 运算符
            while (!stk_opt.empty() && rk[now[i]] >= rk[stk_opt.top()]) {
                execute(stk_num, stk_opt.top());
                stk_opt.pop();
            }
            stk_opt.push(now[i]);
        }
    }
    while (!stk_opt.empty()) {
        execute(stk_num, stk_opt.top());
        stk_opt.pop();
    }
    mp[stk_num.top()]++;
    return stk_num.top();
}

string vec2str(int x) {  // 将 now[] 数组转化为字符串
    string ret = "";
    for (int i = 0; i < x; i++) {
        if (i % 2 == 0) ret += to_string(now[i]);
        else ret.push_back(now[i]);
    }
    return ret;
}

void dfs(int x) {  // x 为当前枚举位置
    if (x % 2 == 1 && calc(x) == n) {
        string s = vec2str(x);
        ans.push(vec2str(x));
    }
```

```cpp
    if (x == 9) return;  // 枚举到头了
    if (x % 2 == 0) {  // 偶数位为数字
        for (int i = 0; i < 5; i++) {
            if (use[i][0]) continue;
            use[i][0] = 1;
            now[x] = a[i];
            dfs(x+1);
            use[i][0] = 0;
        }
    } else {  // 奇数位为运算符
        for (int i = 0; i < 4; i++) {
            if (use[i][1]) continue;
            use[i][1] = 1;
            now[x] = opt[i];
            dfs(x+1);
            use[i][1] = 0;
        }
    }
}

int main() {
    rk['+'] = rk['-'] = 1;  // 设置优先级
    cin >> n;
    for (int i = 0; i < 5; i++) cin >> a[i];
    dfs(0);
    if (ans.size() == 0) cout << "No Solution" << '\n';
    cout << "Total Solution: " << ans.size() << '\n';
    while (!ans.empty()) cout << ans.top() << '\n', ans.pop();

    // 反向求出每个 n 对应的解的个数
    printf("\nn\tSolution Num\n");
    int tot = 0;
    for (auto i : mp) cout << i.first << '\t' << i.second << '\n', tot +=
↪   i.second;
    cout << "Total expression: " << tot << '\n';
    return 0;
}
```

4.3 测试结果与性能分析

上述代码还能够输出该数值全部可能达到的结果，但由于篇幅过大这里就不进行演示，只展示与 n 相等的表达式结果：

```
Input:                  Input:                  Input:
151                     666                     5
1 2 3 10 100            3 10 20 33 97           3 10 20 33 97
Output:                 Output:                 Output:
Total Solution: 6       Total Solution: 8       Total Solution: 8
100/2*3+1               97/10-3+33*20           97/33+3
100*3/2+1               97/10-3+20*33           20/10+3
3*100/2+1               97/10+33*20-3           20*10/97+3
1+100/2*3               97/10+20*33-3           10*20/97+3
1+100*3/2               33*20-3+97/10           3+97/33
1+3*100/2               33*20+97/10-3           3+20/10
                        20*33-3+97/10           3+20*10/97
                        20*33+97/10-3           3+10*20/97

Input:                  Input:                  Input:
122                     -1036                   -101
3 10 20 33 97           3 10 20 33 97           3 10 20 33 97
Output:                 Output:                 Output:
Total Solution: 1       Total Solution: 1       Total Solution: 1
97/20*33-10             20-97/3*33              97-20/3*33
```