《算法设计与分析》 期中大作业

2151140 王谦

一、选择问题

题目

问题定义：给定线性序集中 n 个元素和一个整数 k，1≤k≤n，要求找出这 n 个元素中第 k 小的元素，（这里给定的线性集是无序的）。下面三种是可行的方法：

（1）**基于堆的选择：**不需要对全部 n 个元素排序，只需要维护 k 个元素的最大堆，即用容量为 k 的最大堆存储最小的 k 个数，总费时 O（k+（n-k）*logk）

（2）**随机划分线性选择**（教材上的 RandomizedSelect)：在最坏的情况下时间复杂度为 $O(n^2)$,平均情况下期望时间复杂度为 O(n)。

（3）**利用中位数的线性时间选择：**选择中位数的中位数作为划分的基准，在最坏情况下时间复杂度为 O(n)。

请给出以上三种方法的算法描述，用你熟悉的编程语言实现上述三种方法。并通过实际用例测试，给出三种算法的运行时间随 k 和 n 变化情况的对比图（表）。

算法思想

(1)基于堆的选择：

首先将前k个元素放入一个大小为k的最大堆中。

然后遍历剩余的n-k个元素，如果当前元素小于最大堆的根节点，则用该元素替换根节点，并重新调整最大堆。

最后，最大堆的根节点即为第k小的元素。

(2)随机划分线性选择：

首先从待排序的n个元素中随机选取一个枢纽元素pivot。

然后将所有小于等于pivot的元素放到数组左边，所有大于pivot的元素放到数组右边。

如果pivot的位置为k，则直接返回pivot。

否则，如果pivot的位置小于k，则在右边子数组中寻找第k-pivot位置的元素。

否则，在左边子数组中寻找第k个元素。

(3)利用中位数的线性时间选择:

首先将待排序的n个元素划分成n/5组，每组包含5个元素或少于5个元素。

然后对每个子数组排序，并找出每个子数组的中位数，将所有中位数放到数组的前面。

选取中位数数组的中位数作为枢纽元素pivot，将数组划分为左、右两部分，并计算pivot在整个数组中的位置pos。

如果pos = k，则直接返回pivot。

否则，如果k < pos，则递归在左半部分寻找第k小的元素。

否则，在右半部分寻找第k - pos小的元素。

选择编程语言及环境

编程语言是C++，使用的编译器是Visual Studio 2022，代码如下:

(1)基于堆的选择:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <cstring>
#include <ctime>
using namespace std;

const int MAXN = 100000;

int n, k;
int a[MAXN];

//堆排序中的最大堆方法
void max_heapify(int* a, int now, int size)
{
    int left = now * 2 + 1; //左子节点下标
    int right = now * 2 + 2; //右子节点下标
    int largest = now;
    if (left < size && a[left] > a[largest]) //如果左子节点大于当前节点
        largest = left; //更新最大值下标为左子节点
    if (right < size && a[right] > a[largest]) //如果右子节点大于当前节点和左子节点
        largest = right; //更新最大值下标为右子节点
    if (largest != now) //如果最大值不等于当前节点
    {
        swap(a[now], a[largest]); //交换最大值和当前节点
        max_heapify(a, largest, size); //递归调整子树
    }
}

//维护大小为k的最大堆
void maintain_heap(int* a, int size)
{
    for (int i = size / 2 - 1; i >= 0; i--) //从最后一个非叶子节点开始调整，直到根节点
```

```
        max_heapify(a, i, size);
}

//基于堆的选择算法
int heap_select(int* a, int size, int k)
{
    maintain_heap(a, k); //维护大小为k的最大堆
    for (int i = k; i < size; i++) //遍历后面n-k个元素
    {
        if (a[i] < a[0]) //如果当前元素小于最大堆的根节点
        {
            swap(a[i], a[0]); //替换根节点
            max_heapify(a, 0, k); //重新调整最大堆
        }
    }
    return a[0]; //最大堆的根节点即为第k小的元素
}

int main()
{
    srand(time(NULL)); //初始化随机数种子
    cout << "请输入元素个数n和第k小的元素位置k: " << endl;
    cin >> n >> k;
    cout << "请输入" << n << "个元素: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    cout << "第" << k << "小的元素是: " << heap_select(a, n, k) << endl; //输出第k小的元素
    return 0;
}
```

(2)随机划分线性选择:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <cstring>
#include <ctime>

using namespace std;

const int MAXN = 100000; //定义数组最大长度

int n, k;
int a[MAXN]; //定义数组

//生成[l,r]内的随机整数
inline int rand_int(int l, int r)
{
    return l + rand() % (r - l + 1);
}
```

```
//随机划分子数组，并返回枢纽元素下标
int partition(int l, int r)
{
    int p = rand_int(l, r); //在[l,r]中随机选取一个数作为枢纽元素
    swap(a[p], a[r]); //将枢纽元素和右端点交换
    int i = l - 1; //i表示小于等于枢纽元素的元素的最右边的位置
    for (int j = l; j < r; j++) //遍历区间[l,r-1]
    {
        if (a[j] <= a[r]) //如果当前元素小于等于枢纽元素
        {
            i++; //将小于枢纽元素的元素放到左边
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[r]); //把枢纽元素换到中间
    return i + 1; //返回枢纽元素的位置
}

//递归寻找第k小的元素
int randomized_select(int l, int r, int k)
{
    if (l == r) //如果区间长度为1
        return a[l]; //直接返回该元素
    int q = partition(l, r); //随机划分子数组
    int pos = q - l + 1; //计算枢纽元素的位置
    if (pos == k) //如果找到了第k小的元素
        return a[q]; //直接返回该元素
    else if (k < pos) //如果第k小的元素在左边
        return randomized_select(l, q - 1, k); //递归寻找左半部分
    else //否则第k小的元素在右边
        return randomized_select(q + 1, r, k - pos); //递归寻找右半部分
}

int main()
{
    srand(time(NULL)); //初始化随机数生成器
    cout << "请输入元素个数n和第k小的元素位置k：" << endl;
    cin >> n >> k; //输入n和k
    cout << "请输入" << n << "个元素：" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i]; //输入每个元素
    }
    cout << "第" << k << "小的元素是：" << randomized_select(0, n - 1, k) << endl;
//输出第k小的元素
    return 0;
}
```

(3)利用中位数的线性时间选择：

```
#include <iostream>
#include <algorithm>
#include <ctime>

using namespace std;
```

```cpp
const int MAXN = 100000; //定义数组最大长度

int n, k; //n表示数组长度，k表示要求的第k小的元素
int a[MAXN]; //定义数组a

inline int rand_int(int l, int r) //生成[l,r]内的随机整数
{
    return l + rand() % (r - l + 1);
}

int partition(int l, int r, int p) //划分子数组，并返回枢纽元素下标
{
    swap(a[p], a[r]); //将枢纽元素放到数组末尾
    int i = l - 1;
    for (int j = l; j < r; j++)
    {
        if (a[j] <= a[r])
        {
            i++;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[r]); //将枢纽元素放到正确位置
    return i + 1; //返回枢纽元素下标
}

int median(int l, int r) //计算子数组的中位数，并返回其下标
{
    if (l == r)
        return l;
    int i = l;
    for (; i + 4 <= r; i += 5)
    {
        sort(a + i, a + i + 5); //对每个长度为5的子数组排序
        swap(a[l + (i - l) / 5], a[i + 2]); //将每个子数组中位数放到前面
    }
    if (i <= r)
    {
        sort(a + i, a + r + 1); //对最后不足5个元素的子数组排序
        swap(a[l + (i - l) / 5], a[i + (r - i) / 2]); //将最后一个子数组的中位数放到前
面
    }
    int mid = l + (r - l) / 10; //计算中位数下标
    return mid;
}

int median_select(int l, int r, int k) //递归寻找第k小的元素
{
    if (l == r)
        return a[l];
    int p = median(l, r);
    int q = partition(l, r, p);
    int pos = q - l + 1;
    if (pos == k)
```

```
            return a[q];
        else if (k < pos)
            return median_select(l, q - 1, k);
        else
            return median_select(q + 1, r, k - pos);
}

int main()
{
    srand(time(NULL)); //初始化随机数种子
    cout << "请输入元素个数n和要求的第k小的元素位置k:" << endl;
    cin >> n >> k; //输入数组长度n和要求的第k小的元素值k
    cout << "请输入" << n << "个整数:" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i]; //输入数组元素
    }
    int ans = median_select(0, n - 1, k); //寻找第k小的元素
    cout << "第" << k << "小的数是:" << ans << endl; //输出第k小的元素
    return 0;
}
```

系统的输入输出运行结果

(1)基于堆的选择:

(2)随机划分线性选择:

(3)利用中位数的线性时间选择:

算法分析

(1)基于堆的选择：

主要分为三个部分：max_heapify()函数、maintain_heap()函数和heap_select()函数。

- max_heapify()函数用于维护最大堆性质。它的输入参数包括数组a，当前节点下标now，以及数组大小size。它判断当前节点和其左右子节点中的最大值，如果最大值不等于当前节点，则交换它们的值，并递归调整子树。
- maintain_heap()函数用于维护大小为k的最大堆。它的输入参数包括数组a和数组大小size。它从最后一个非叶子节点开始，依次调用max_heapify()函数，直到根节点，使得数组a满足最大堆的性质。
- heap_select()函数用于寻找第k小的元素。它的输入参数包括数组a、数组大小size和要寻找的第k小的元素位置k。它先将前k个元素放入一个大小为k的最大堆中，然后遍历剩余的n-k个元素，如果当前元素小于最大堆的根节点，则用该元素替换根节点，并重新调整最大堆。最后，最大堆的根节点即为第k小的元素。

时间复杂度实际上可以达到O(k+(n-k)logk)。

(2)随机划分线性选择：

主要由两个函数组成：partition()函数和randomized_select()函数。

- partition()函数用于随机划分子数组，并返回枢纽元素的下标。它的输入参数包括子数组的左端点l和右端点r。它通过选取一个随机数作为枢纽元素，将枢纽元素放到数组末尾，然后遍历数组，将小于等于枢纽元素的元素放到数组左边，将大于枢纽元素的元素放到数组右边，最后将枢纽元素放到正确位置，并返回其下标。
- randomized_select()函数用于递归寻找第k小的元素。它的输入参数包括子数组的左端点l、右端点r和要寻找的第k小的元素位置k。它先选取一个随机数p作为枢纽元素，然后调用partition()函数进行划分，并根据枢纽元素的位置和k的大小关系来决定在左半部分或右半部分继续递归。如果枢纽元素的位置正好是k，则直接返回该元素；否则，在子数组的左半部分或右半部分继续寻找第k小的元素。

时间复杂度实际上可以达到O(n)，但是最坏的情况下会达到O(n^2)。

(3)利用中位数的线性时间选择：

主要由三个函数组成：partition()函数、median()函数和median_select()函数。

- partition()函数用于划分子数组，并返回枢纽元素的下标。它的输入参数包括子数组的左端点l、右端点r和指定的枢纽元素p。它将枢纽元素放到数组末尾，然后遍历数组，将小于等于枢纽元素的元素放到数组左边，将大于枢纽元素的元素放到数组右边，最后将枢纽元素放到正确位置，并返回其下标。

- median()函数用于计算子数组的中位数，并返回其下标。它的输入参数包括子数组的左端点l和右端点r。它首先将子数组划分成n/5组，对每个长度为5的子数组排序，并将每个子数组的中位数放到前面。然后对最后不足5个元素的子数组排序，将最后一个子数组的中位数放到前面。最后，计算中位数的下标并返回。
- median_select()函数用于递归寻找第k小的元素。它的输入参数包括子数组的左端点l、右端点r和要寻找的第k小的元素位置k。它调用median()函数计算中位数的下标p，然后调用partition()函数进行划分，并根据枢纽元素的位置和k的大小关系来决定在左半部分或右半部分继续递归。如果枢纽元素的位置正好是k，则直接返回该元素；否则，在子数组的左半部分或右半部分继续寻找第k小的元素。

时间复杂度实际可以达到O(n)，即使是最坏的情况下也不会像(2)一样达到O(n^2)。

二、博物馆警卫巡逻问题

题目

凸多边形是每个内角小于 180 度的多边形。

博物馆是具有 n 个顶点的凸多边形的形状。博物馆由警卫队通过巡逻来确保馆内物品的安全。博物馆的安全保卫工作遵循以下规则，以尽可能时间经济的方式确保最大的安全性：

（1）警卫队中每个警卫巡逻都沿着一个三角形的路径；该三角形的每个顶点都必须是多边形的顶点。

（2）警卫可以观察其巡逻路径三角形内的所有点，并且只能观察到这些点；我们说这些点由该警卫守护并覆盖。

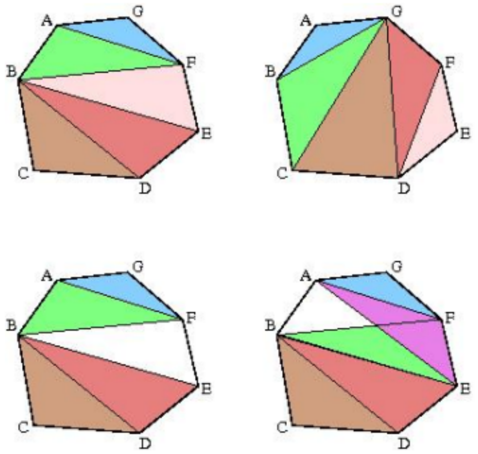
（3）博物馆内的每一处都必须由警卫人员守护。

（4）任何两个警卫巡逻所在的三角形在其内部不会重叠，但它们可能具有相同的边。

在这些限制条件下，警卫的成本是警卫巡逻所沿路径的三角形的周长。

我们的目标是找到一组警卫，以使警卫队的总成本（即各个警卫的成本之和）尽可能小。给定博物馆顶点的 x 坐标和 y 坐标以及这些顶点沿博物馆边界的顺序，设计一种算法求解该问题，并给出算法的时间复杂性。

请注意，我们并未试图最小化警卫人数。我们希望使警卫队巡逻的路线的总长度最小化，假定任何线段的长度都是线段端点之间的欧几里得距离，并且可以在恒定时间内计算该长度。



上面是说明本问题的四个图形。博物馆是多边形 ABCDEFG（顶点的逆时针序）。每个彩色（阴影）三角形对应一个警卫，

上面的两个图显示了一组警卫（它们的三角形），它们满足安全保卫规则。在左上方，警卫队巡逻了三角形 AFG（蓝色），ABF（绿色），BEF（淡红色），BDE（浅红色）和 BCD（棕色）的边界。在右上方，守卫巡逻 ABG（蓝色），BCG（绿色），CDG（棕色），DFG（浅红色）和 DEF（浅红色）。

底部的两个图显示了一组不满足这些规则的三角形：在左下图中，博物馆的一部分没有任何警卫守护（覆盖）（无阴影三角形 BEF），而在右下图中，粉色三角形（AEF）和绿色三角形（BEF）相交。

算法思想

该问题描述了一个凸多边形的形状和大小，以及一个初始警卫巡逻的位置。我们需要在保证所有部分都被巡逻到的前提下，选择所有三角形周长和最小的方案。

根据该问题的特点，我们可以将其转化为求解给定凸多边形的最优三角剖分方案。具体来说，我们可以定义一个 dp 数组，其中 dp[i][j] 表示对于凸多边形的第 i 个顶点到第 j 个顶点之间的所有顶点，它们组成的多边形的最优三角剖分方案对应的周长之和。在计算子问题的最优解时，我们需要枚举断点 k，并计算出 dp[i][j] 的值，从而得到当前问题的最优解。同时，我们还需要记录下每个子问题的最优策略，即哪条对角线是最优的。最后，我们可以根据最优策略输出所有子问题的弦，从而得到整个凸多边形的最优三角剖分方案。

选择编程语言及环境

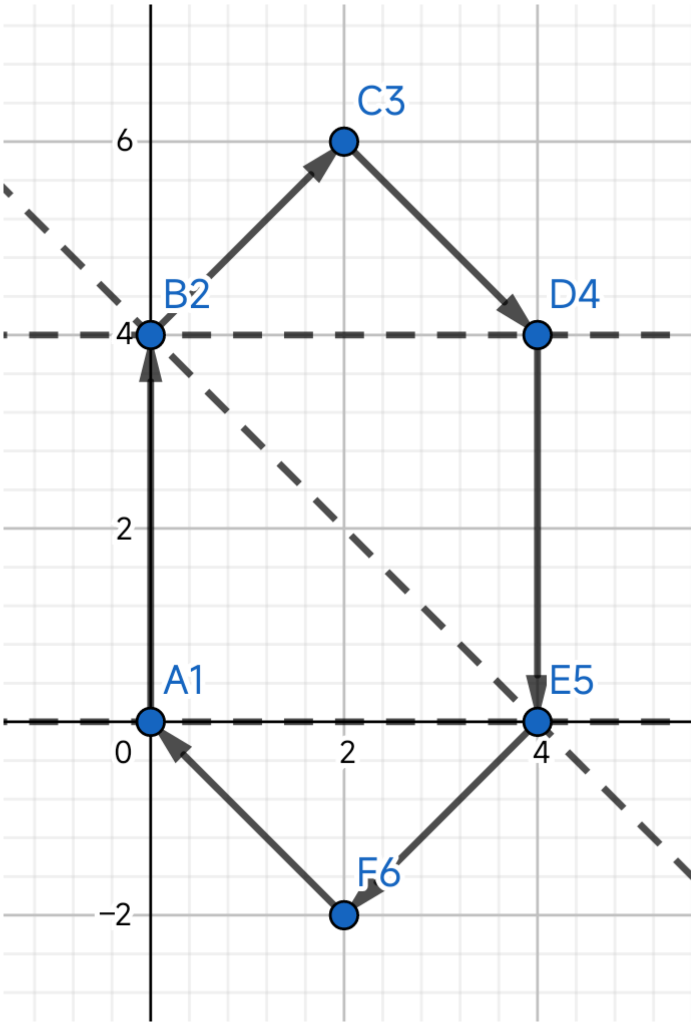
编程语言是C++，使用的编译器是Visual Studio 2022，代码如下：

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

const int N = 111;
int n;
double x[N], y[N]; // 各顶点坐标
```

```
double dp[N][N]; // 记录最优值
int s[N][N]; // 记录最优策略

// 计算两点之间的距离
double dist(int i, int j)
{
    return sqrt((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]));
}

// DP算法求解多边形划分三角形周长之和最小的最优三角剖分
double convexPolygonTriangulation()
{
    for (int d = 2; d <= n - 1; d++) { // d 为问题规模，d = 2 实际有三个点
        for (int i = 1; i <= n - d; i++) { // 控制 i 值
            int j = i + d; // j 值
            dp[i][j] = -1; // 初始化为负值，用于判断是否已被计算过
            for (int k = i + 1; k < j; k++) { // 枚举断点
                double temp = dp[i][k] + dp[k][j] + dist(i, j) + dist(i, k) +
dist(k, j);
                if (dp[i][j] < 0 || dp[i][j] > temp) {
                    dp[i][j] = temp;
                    s[i][j] = k;
                }
            }
        }
    }
    return dp[1][n]; // 返回最优值
}

// 根据最优策略输出所有子问题的弦
void print(int i, int j)
{
    if (i + 1 >= j) return; // 只有三个点以上才存在对角线
    cout << "{" << i << "," << j << "," << s[i][j] << "}" << endl;
    print(i, s[i][j]);
    print(s[i][j], j);
}

int main()
{
    int i, j;
    cout << "请输入凸多边形的顶点个数 n: " << endl;
    cin >> n;
    cout << "请依次输入各顶点的坐标(x,y): " << endl;
    for (i = 1; i <= n; i++) {
        cin >> x[i] >> y[i];
    }
    // 计算DP数组中的最优值和最优策略
    double minSumLen = convexPolygonTriangulation();
    // 输出最优三角剖分中所有三角形的周长之和
    cout << "划分出的所有三角形周长之和最小值是: " << endl;
    cout << minSumLen << endl;
    // 输出所有对角线上的弦
    cout << "凸多边形的最优三角剖分方案为: " << endl;
    print(1, n);
```

```
    return 0;
}
```

系统的输入输出运行结果



算法分析

该算法使用动态规划的思想，通过计算子问题的最优解和最优策略，逐步计算出原问题的最优解和最优策略。具体来说，它首先定义了存储顶点坐标、最优值和最优策略的数组 `x`, `y`, `dp` 和 `s`，然后，它通过 `dist` 函数计算两点之间的距离，通过 `convexPolygonTriangulation` 函数计算出多边形的最优三角剖分及其周长之和，通过 `print` 函数输出所有子问题的弦，并在主函数中调用这些函数以完成整个算法的执行。

时间复杂度：

在该算法中，需要计算 n 个子问题。对于每个子问题 dp[i][j]，需要枚举断点 k，因此需要进行 O(n) 次循环。而在计算每个子问题时，需要对剩余的两个子多边形进行递归求解，这也就是说，每次递归都会减少一个顶点，因此需要进行 n-3 次递归。在递归过程中，需要计算当前子问题对应的最优值和最优策略，这需要进行 O(1) 的计算。因此，总时间复杂度为：

$$T(n) = \sum_{i=3}^{n} \sum_{j=1}^{n-i+1} O(n) + O(1) = \frac{n(n^2 - 3n + 4)}{6} = O(n^3)$$

空间复杂度： 在该算法中，需要存储顶点坐标、最优值和最优策略的数组 x, y, dp 和 s，每个数组的大小为 O(n^2)。因此，总空间复杂度为 O(n^2)。 综上所述，该算法的时间复杂度为 O(n^3)，空间复杂度为 O(n^2)。

三、主元素问题

题目

设 A 是含有 n 个元素的数组，如果元素 x 在 A 中出现的次数大于 n/2，则称 x 是 A 的主元素，

（1）如果 A 中的元素是可以排序的，设计一个 O(nlogn)时间的算法，判断 A 中是否存在主元素；

（2）对于（1）中可排序的数组，能否设计一个 O(n)时间的算法；

（3）如果 A 中元素只能进行“是否相等”的测试，但是不能排序，设计一个算法判断 A 中是否存在主元素。

算法思想

(1)要求的O(nlogn)算法，排序算法：

如果数组可以排序，可以使用一种基于排序的算法来解决主要元素问题。首先将数组进行排序，然后计算排序后数组中每个元素出现的次数，最后判断其中是否存在任何一个元素出现次数超过 n/2 次。因为排序的时间复杂度为 O(nlogn)，计数器的时间复杂度为 O(n)，因此，总的时间复杂度为 O(nlogn)。

(2)(3)要求的O(n)的算法,只比较是否相等而不排序，投票算法：

如果数组可以排序，也可以使用一种基于投票的算法来解决主要元素问题。该算法的基本思想是遍历数组，用一个计数器记录当前元素出现的次数，并不断更新候选元素。如果遇到新的元素与候选元素相同，则将计数器加 1；否则将计数器减 1。当计数器归零时，说明前面出现的所有元素出现次数都不足一半，可以将当前元素设为新的候选元素。最终，如果候选元素出现次数大于 n/2，则该元素即为主要元素。由于投票算法的时间复杂度为 O(n)，因此，总的时间复杂度为 O(n)。

选择编程语言及环境

编程语言是C++，使用的编译器是Visual Studio 2022，代码如下：

(1)要求的O(nlogn)算法，排序法：

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

// 判断数组中是否存在主要元素
int findMajorityElement(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) {
        return -1;
    }
    sort(nums.begin(), nums.end()); // 排序
    int count = 1, candidate = nums[0];
    for (int i = 1; i < n; ++i) {
        if (nums[i] == candidate) {  // 如果遇到相同元素，则计数器 +1
            count++;
        } else {
            if (count > n / 2) {     // 如果当前元素不同，则判断之前的候选元素是否为主要
元素
                return candidate;
            }
            count = 1;              // 否则更新候选元素和计数器
            candidate = nums[i];
        }
    }
    return (count > n / 2) ? candidate : -1;
}

int main() {
    cout << "请输入数组大小：";     // 提示用户输入数组大小
    int size;
    cin >> size;
    vector<int> nums(size);
    cout << "请输入数组元素：" << endl;   // 提示用户输入数组元素
    for (int i = 0; i < size; ++i) {
        cin >> nums[i];
    }

    int majorityElement = findMajorityElement(nums);   // 查找主要元素

    if (majorityElement != -1) {    // 如果主要元素存在，则输出该元素
        cout << "主要元素为：" << majorityElement << endl;
    }
    else {  // 否则提示不存在主要元素
        cout << "不存在主要元素！" << endl;
    }

    return 0;
}
```

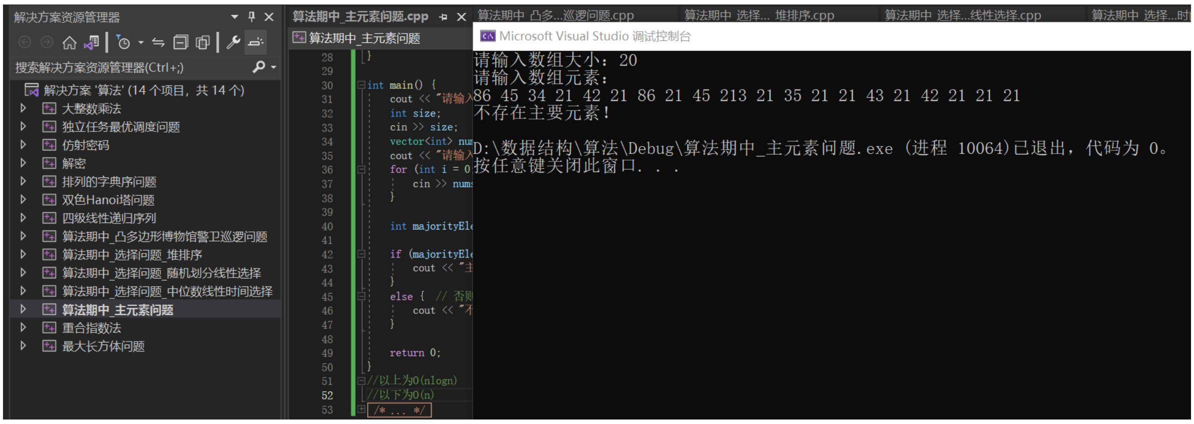
(2)(3)要求的O(n)的算法,只比较是否相等而不排序，投票算法：

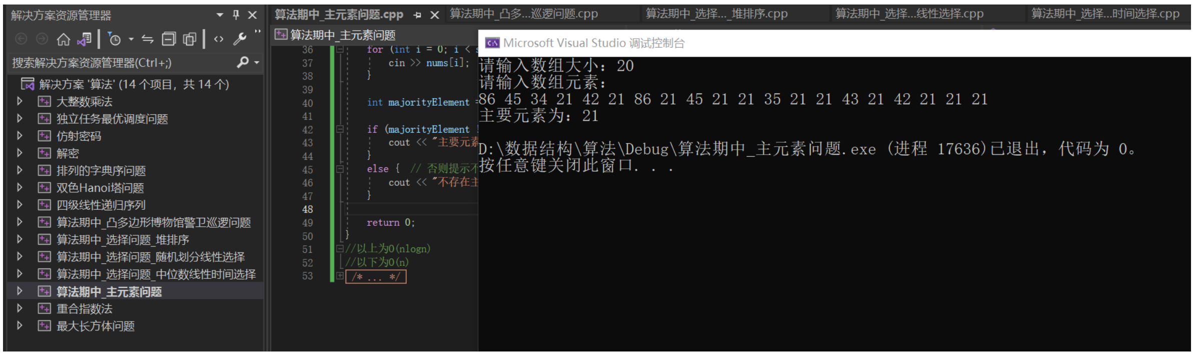
```
#include <iostream>
#include <vector>

using namespace std;

// 查找给定数组的主要元素（出现次数超过一半的元素）
```

```cpp
int findMajorityElement(vector<int>& nums) {
    int candidate = 0, count = 0;  // 初始化候选元素和计数器

    // 遍历整个数组
    for (int num : nums) {
        if (count == 0) {           // 如果计数器为 0，则将当前元素设为候选元素
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;  // 如果当前元素等于候选元素，则计数器
+1，否则 -1
    }

    count = 0;       // 重新初始化计数器
    // 再次遍历整个数组
    for (int num : nums) {
        if (num == candidate) {     // 统计候选元素在数组中出现的次数
            count++;
        }
    }

    // 如果候选元素出现次数大于 n/2，则返回该元素，否则返回 -1
    return (count > nums.size() / 2) ? candidate : -1;
}

int main() {
    cout << "请输入数组大小：";      // 提示用户输入数组大小
    int size;
    cin >> size;
    vector<int> nums(size);
    cout << "请输入数组元素：" << endl;    // 提示用户输入数组元素
    for (int i = 0; i < size; ++i) {
        cin >> nums[i];
    }

    int majorityElement = findMajorityElement(nums);    // 查找主要元素

    if (majorityElement != -1) {     // 如果主要元素存在，则输出该元素
        cout << "主要元素为：" << majorityElement << endl;
    }
    else {  // 否则提示不存在主要元素
        cout << "不存在主要元素！" << endl;
    }

    return 0;
}
```

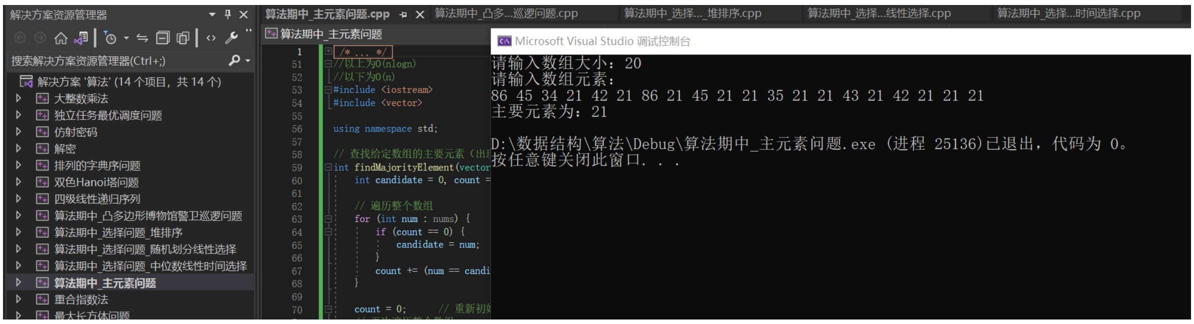
系统的输入输出运行结果

(1)要求的O(nlogn)算法，排序法：





(2)(3)要求的O(n)的算法，只比较是否相等而不排序，投票算法：



算法分析

(1)要求的O(nlogn)算法，排序法：

该程序的主要思想是先将数组进行排序，然后遍历整个数组，计算每个元素在数组中出现的次数，并判断其中是否存在主要元素。

- 输入数据部分：时间复杂度为 O(n)，其中 n 为数组大小；空间复杂度为 O(n)，需要一个长度为 n 的 vector 存储数组元素。
- 排序部分：时间复杂度为 O(nlogn)，因为使用了 std::sort 函数进行快速排序。
- 统计候选元素出现次数部分：时间复杂度为 O(n)，需要遍历一次数组，统计候选元素在数组中出现的次数。
- 判断是否有主要元素部分：时间复杂度为 O(1)。

综上所述，该算法的时间复杂度为 O(nlogn)，空间复杂度为 O(n)。

(2)(3)要求的O(n)的算法，只比较是否相等而不排序，投票算法：

当用户输入一个包含 n 个元素的数组时，该程序使用了摩尔投票算法来查找主要元素。其主要思想是：

1. 初始化 candidate 和 count，candidate 表示当前可能成为主要元素的候选元素，count 表示当前候选元素出现的次数。
2. 遍历数组中所有元素。如果 count 为零，则将当前元素设为候选元素，否则如果当前元素等于候选元素，则 count++，否则 count--。
3. 第一遍遍历结束后，candidate 存储的就是可能的主要元素，但仍需要再次遍历整个数组进行验证。
4. 在第二次遍历中，统计 candidate 在数组中出现的次数，如果其次数大于 n/2，则 candidate 就是主要元素，否则不存在主要元素。

该程序的主要思想是用一个变量记录当前候选元素，同时用另一个变量记录当前候选元素出现次数，而当遇到新的元素时进行消除。

- 输入数据部分：时间复杂度为 O(n)，其中 n 为数组大小；空间复杂度为 O(n)，需要一个长度为 n 的 vector 存储数组元素。
- 统计候选元素出现次数部分：时间复杂度为 O(n)，需要再次遍历一次数组，统计候选元素在数组中出现的次数。
- 判断是否有主要元素部分：时间复杂度为 O(1)。

综上所述，该算法的时间复杂度为 O(n)，空间复杂度为 O(1)。