HW02 递归与分治策略

一 相关知识

- 本作业涉及递归与分治策略的相关知识，学习要点在于理解递归的概念，掌握设计有效算法的分治策略。
- 分治法应用极其广泛，其主要思想就是分而治之，将一个复杂问题分割为多个可以解决的小问题进行处理，最后再进行合并，这也是递归法的原理。
- 递归算法指直接或间接地调用自身的算法。递归算法结构清晰，可读性强，但是存在运行时效率较低，耗费较多时间、空间资源的缺点。
- 递归函数存在两个要素，是边界条件和递归方程，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

二.问题实现

1. **题目描述**

> 最小的K个整数：输入整数数组arr，找出其中最小的k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。**备注：使用线性时间选择方法完成。**

2. **问题分析**

这道题要求从n个数中选出最小的k个数，很显然这是一个需要进行排序的问题，思考常见的排序算法时间复杂度，如下：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定

本题要求使用线性时间选择完成，线性时间选择方法通常指的是在O(n)时间复杂度内解决问题的算法，也就是说要以O(n)的时间复杂度实现，即一趟就选出所需要的K个数。观察上表其实并没符合的，但是上述时间复杂度为完全排序所需，这道题仅选出K个就可，不必全部排序。

这样就注意到了快速排序，因为快速排序思路为选取一个枢轴变量p，再声明两个临时变量left，right分别从左右向数轴变量移动，并进行比较，将小于枢轴元素的放到左边，大于枢轴变量的放到右边，随后再以left为枢轴变量继续上述过程。由此我们可以发现，利用快速排序的思想，可以在一趟移动后选出所需要的K个最小元素，即左侧K个元素，通过这个方法划分出所需要的元素，从而实现线性时间复杂度。

3. **算法设计**

上述分析已知找前K大或者前K小的问题不需要对整个数组进行O(NlogN)的排序，本题可以直接通过快排切分排好第K小的数（下标为K-1），那么它左边的数就是比它晓得另外K-1个数，同时也满足了线性时间要求。

在算法的设计实现上，使用了Vector，便于移动，以及具有动态长度。SortK函数用于利用快排排好并挑选出K个数。findKnums函数则将挑选出的K个数存储到结果容器res中，随后输出。

4. **遇到的问题**

边界条件的处理；递归终止条件及递归参数的传递；

5. **源代码**

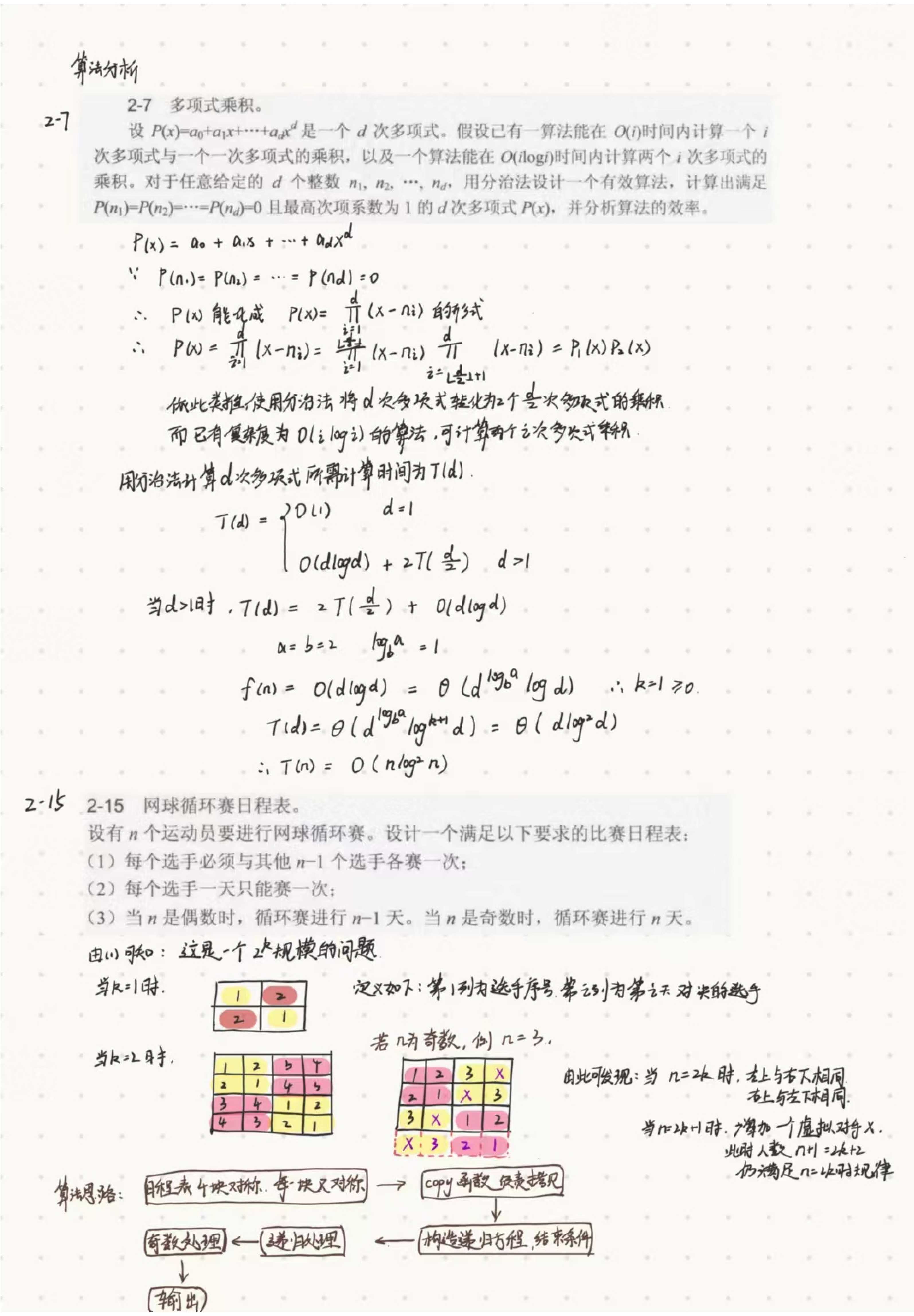
```
#include<iostream>
#include<vector>
using namespace std;
// @brief 基于快速排序的思想，找到数组中最小的k个数
// @param arr 数组
// @param left 左边界
// @param right 右
void sortK(vector<int>& arr,int left,int right,int k)
{
    if (left > right)
        return;
    int i = left,j=right,tmp = arr[left];
    while (i<j){
        //tmp作为枢轴变量，交换左右位置的元素
        while (i < j && arr[j] >= tmp) j--;
        while (i < j && arr[i] <= tmp) i++;
        if(i<j)
            swap(arr[i],arr[j]);
    }
    arr[left] = arr[i]; //将枢轴元素放到正确的位置
    arr[i] = tmp;
    if (i == k - 1) return;
```

```
// @brief 调用sortK函数选出最小的k个数，并存在res中
vector<int> findKnums(vector<int>& arr, int k)
{
    vector<int> res;
    if (arr.size() < k)
        return res;
    sortK(arr,0,arr.size()-1,k);
    for (int i = 0; i < k; i++)
        res.push_back(arr[i]);//将最小的k个数放到res中
    return res;
}
int main(){
    int n, k;
    cout << "n= ;k= ;" << endl;
    cin >> n >> k;
    vector<int> arr(n);
    cout<<"arr[]="<<endl;
    for (int i = 0; i < n; ++i)
        cin >> arr[i];
    vector<int> res = findKnums(arr,k);
    for (int i = 0; i < res.size(); i++)
        cout << res[i] << " ";
```

```
    else if(i<k-1)
        return sortK(arr,i+1,right,k);//递归右边
    else
        return sortK(arr,left,i-1,k);//递归左边
}
```

```
    cout << endl;
    return 0;
}
```

三.算法分析题



算法分析

2-7 2-7 多项式乘积。

设 $P(x)=a_0+a_1x+\cdots+a_dx^d$ 是一个 d 次多项式。假设已有一算法能在 $O(i)$时间内计算一个 i 次多项式与一个一次多项式的乘积，以及一个算法能在 $O(i\log i)$时间内计算两个 i 次多项式的乘积。对于任意给定的 d 个整数 $n_1, n_2, \cdots, n_d$，用分治法设计一个有效算法，计算出满足 $P(n_1)=P(n_2)=\cdots=P(n_d)=0$ 且最高次项系数为 1 的 d 次多项式 $P(x)$，并分析算法的效率。

$P(x) = a_0 + a_1x + \cdots + a_dx^d$

$\because P(n_1) = P(n_2) = \cdots = P(n_d) = 0$

$\therefore P(x)$ 能化成 $P(x) = \prod_{i=1}^{d}(x-n_i)$ 的形式

$\therefore P(x) = \prod_{i=1}^{d}(x-n_i) = \prod_{i=1}^{\lfloor \frac{d}{2} \rfloor}(x-n_i)\prod_{i=\lfloor \frac{d}{2} \rfloor+1}^{d}(x-n_i) = P_1(x)P_2(x)$

依此类推，使用分治法将 d 次多项式转化为2个 $\frac{d}{2}$ 次多项式的乘积

而已有复杂度为 $O(i\log i)$ 的算法，可计算两个 i 次多项式乘积

用分治法计算 d 次多项式所需计算时间为 $T(d)$

$$T(d) = \begin{cases} O(1) & d=1 \\ O(d\log d) + 2T(\frac{d}{2}) & d>1 \end{cases}$$

当 $d>1$ 时，$T(d) = 2T(\frac{d}{2}) + O(d\log d)$

$a=b=2 \quad \log_b a = 1$

$f(n) = O(d\log d) = \theta(d^{\log_b a}\log d) \quad \therefore k=1 \geq 0$

$T(d) = \theta(d^{\log_b a}\log^{k+1} d) = \theta(d\log^2 d)$

$\therefore T(n) = O(n\log^2 n)$

2-15 2-15 网球循环赛日程表。

设有 n 个运动员要进行网球循环赛。设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他 $n-1$ 个选手各赛一次；

（2）每个选手一天只能赛一次；

（3）当 n 是偶数时，循环赛进行 $n-1$ 天。当 n 是奇数时，循环赛进行 n 天。

由(1)可知：这是一个 2^k 规模的问题

当k=1时.

1	2
2	1

定义如下：第1列为选手序号，第i列为第i天对决的选手

当k=2时.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

若n为奇数，例 $n=3$.

1	2	3	X
2	1	X	3
3	X	1	2
X	3	2	1

由此可发现：当 $n=2k$ 时，左上与右下相同，右上与左下相同.

当 $n=2k+1$ 时，增加一个虚拟对手X.
此时人数 $n+1=2k+2$
仍满足 $n=2k$ 时规律

算法思路：日程表4块对称，每一块又对称 → copy函数 实现 → 构造递归方程，结束条件 → 递归处理 → 奇数处理 → 输出

接2-15：经过上述分析后发现n为偶数时最为简便，此时日程表左上对右下，左下对右上；当n为奇数时，可以引入一个虚拟选手X，这样使得n+1变为偶数，也可以按照上述思路进行处理。同时我发现可以将日程表无限分下去，例如分成四块后，每一小块又可以再分成四块，且相互之间的关系仍相同，如此递归下去，直到不可再分为止（终止条件n==1），所以我们可以构造一个递归方程。

算法实现上：最主要的是递归函数和copy函数的编写，copy函数需要分两种情况，一种是n为奇数，一种是n为偶数，n为偶数时依次将左上角，右下角，右上角，左下角进行拷贝即可。对于奇数时需要先将n+1赋值给m作为列，随后进行拷贝即可，但是要注意标记出虚拟选手（*）。

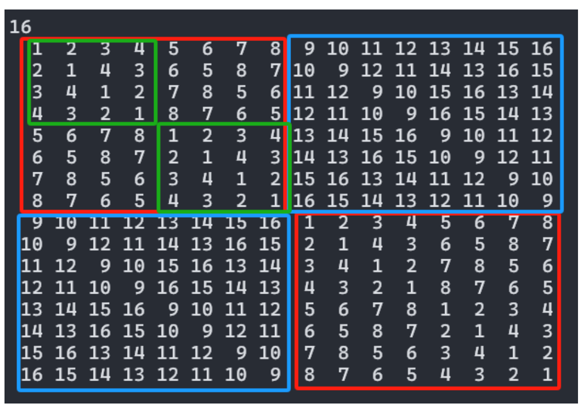
```
/// @brief 递归函数
/// @param n
void schedule(int n)
{
    if (n == 1) { //终止条件
        a[1][1] = 1;
        return;
    }
    if (n % 2)
        n++;
    schedule(n / 2);
    if (n / 2 > 1 && (n / 2) % 2 != 0)
        copyodd(n); //n为奇数时
    else
        copyeven(n);//n为偶数时
}
```

递归方程分析：

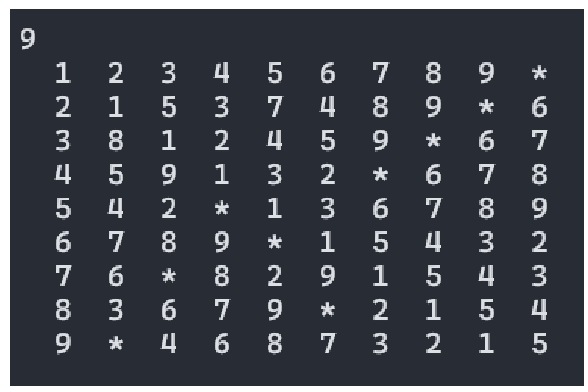
$$T(k)=\begin{cases} O(1) & n=2 \\ T(k/2)+O(n^2) & n>2, n \bmod 2=0 \\ T(k+1) & n>2, n \bmod 2=1 \end{cases}$$

解此递归方程可得：$T(k)=O(n^2)$
算法是渐进意义下的最优算法。
除此之外还注意到有种利用多边形几何关系的方法

运行截图：



```
16
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 2  1  4  3  6  5  8  7 10  9 12 11 14 13 16 15
 3  4  1  2  7  8  5  6 11 12  9 10 15 16 13 14
 4  3  2  1  8  7  6  5 12 11 10  9 16 15 14 13
 5  6  7  8  1  2  3  4 13 14 15 16  9 10 11 12
 6  5  8  7  2  1  4  3 14 13 16 15 10  9 12 11
 7  8  5  6  3  4  1  2 15 16 13 14 11 12  9 10
 8  7  6  5  4  3  2  1 16 15 14 13 12 11 10  9
 9 10 11 12 13 14 15 16  1  2  3  4  5  6  7  8
10  9 12 11 14 13 16 15  2  1  4  3  6  5  8  7
11 12  9 10 15 16 13 14  3  4  1  2  7  8  5  6
12 11 10  9 16 15 14 13  4  3  2  1  8  7  6  5
13 14 15 16  9 10 11 12  5  6  7  8  1  2  3  4
14 13 16 15 10  9 12 11  6  5  8  7  2  1  4  3
15 16 13 14 11 12  9 10  7  8  5  6  3  4  1  2
16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```



```
9
  1  2  3  4  5  6  7  8  9  *
  2  1  5  3  7  4  8  9  *  6
  3  8  1  2  4  5  9  *  6  7
  4  5  9  1  3  2  *  6  7  8
  5  4  2  *  1  3  6  7  8  9
  6  7  8  9  *  1  5  4  3  2
  7  6  *  8  2  9  1  5  4  3
  8  3  6  7  9  *  2  1  5  4
  9  *  4  6  8  7  3  2  1  5
```

源代码：

```
#include<iostream>
#include<iomanip>
using namespace std;
int a[1001][1001];
int b[1001];
/// @brief n为偶数copy相应部分
void copyeven(int n)
{
    int m=n/2;
    for (int i = 1; i <= m; i++){
        for (int j = 1; j <= m; j++){
            a[i][j + m] = a[i][j] + m;//右上角
            a[i + m][j]= a[i][j + m];//左下角
            a[i + m][j + m]=a[i][j];//右下角
        }
    }
}
/// @brief n为奇数copy相应部分
void copyodd(int n)
{
    int m = n / 2;
    for (int i = 1; i <= m; ++i) {
        b[i] = m + i;
        b[m + i] = b[i];
    }
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= m + 1; j++) {
            if (a[i][j] > m) {
                a[i][j] = b[i];
                a[m+i][j] = (b[i]+m)%n;
            }
            else {
                a[m + i][j] = a[i][j] + m;
            }
        }
        for (int j = 2; j <= m; j++) {
            a[i][m + j] = b[i + j - 1];//右上角
            a[b[i+j-1]][m + j] = i;//左下角
        }
    }
}
```

```
/// @brief 递归函数
void schedule(int n)
{
    if (n == 1) { //终止条件
        a[1][1] = 1;
        return;
    }
    if (n % 2)
        n++;
    schedule(n / 2);
    if (n / 2 > 1 && (n / 2) % 2 != 0)
        copyodd(n); //n为奇数时
    else
        copyeven(n);//n为偶数时
}
int main()
{
    int n;
    cin >> n;
    schedule(n); //调用递归函数进行处理
    //输出日程表，特别是奇数时的输出
    int m = n; //用于奇数时+1改变列数
    bool flag = true;//偶数标志
    if (n % 2 == 1) {
        m++;
        flag = false;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (!flag && a[i][j] == m) {
                cout<<"  *";//虚拟选手
            }
            else
                cout<<setw(3)<<a[i][j] ;//格式化输出
        }
        cout << endl;
    }
    return 0;
}
```

四.算法设计题

2-6 排列的字典序问题

1. **题目描述：**

> n个元素{1,2,......, n }有n!个不同的排列。将这n!个排列按字典序排列，并编号为0，1，...，n!-1。每个排列的编号为其字典序值。例如，当n=3时，6 个不同排列的字典序值如下。给定n以及n个元素{1,2,......, n }的一个排列，计算出这个排列的字典序值，以及按字典序排列的下一个排列。
>
> | 字典序值 | 0 | 1 | 2 | 3 | 4 | 5 |
> |---|---|---|---|---|---|---|
> | 排列 | 123 | 132 | 213 | 231 | 312 | 321 |

2. **算法设计：**

康拓展开：是一个全排列到一个自然数的双射，常用于构建哈希表时的空间压缩。 康托展开的实质是计算当前排列在所有由小到大全排列中的顺序，因此是可逆的。

$$X = a_n(n-1)! + a_{n-1}(n-2)! + \cdots + a_1 \cdot 0!$$

由排列计算字典序：因为排列是按字典序排名的，因此越靠前的数字优先级越高。也就是说如果两个排列的某一位之前的数字都相同，那么如果这一位如果不相同，就按这一位排序。设给定的 {1，2，.....n} 的排列为∏，其字典序值为 rank(∏,n).按字典序的定义显然有：

$$(\prod[1]-1*(n-1)! \leq rank(\prod,n) \leq \prod[1]*(n-1)!-1$$

$$rank(\prod,n) = (\prod[1]-1)*(n-1)! + rank(\prod,n-1)$$

```
/// @brief 计算当前排列在所有排列中的顺序（递归实现）
/// @param data 排列数据
/// @param n    排列长度
/// @return 当前排列的顺序
int order(int* data, int n) {
    if (n <= 1) return 0;
    int count = 0;
    for (int i = 1; i < n; ++i)
        if (data[0] > data[i]) count++;
    return count * factorial(n - 1) + order(data + 1, n - 1);
}
```

由排列计算下一个排列：按字典序的定义可设计从一个排列计算下一个排列的算法。对于给定的排列∏，

①首先找到下标 i ,使得∏[i] < ∏[i+1],且∏[i +1] > ∏[i+2] > ...> ∏[n];

②其次找到下标 j，使得∏[i]<∏[j]且对所有j<k≤n 有∏[k]<∏[i];

③然后交换∏[i] 和 ∏[j];

④最后将排列[∏[i+1],∏[i+2],...,∏[n]]反转。

```
/// @brief 将排列 data 的下一个字典序排列存储在 data 中
/// @param data 排列数据
/// @param n    排列长度
void nextOrder(int* data, int n){
    int i = n - 2;
    while (i >= 0 && data[i] >= data[i + 1]) i--;
    if (i >= 0) {
        int j = n - 1;
        while (data[j] <= data[i]) j--;
        swap(data[i], data[j]);
        reverse(data + i + 1, data + n);
    }
}
```

时间复杂度分析：

①**factorial** 函数：计算阶乘，时间复杂度为 O(n)

②**order** 函数通过递归实现了计算排列在所有排列中的顺序。在每次递归调用中，需要进行一次循环来计算当前元素之后比当前元素小的个数。因此，总体上需要进行 n(n-1)/2 次比较，其中 n 是排列长度。因此，时间复杂度为 O(n^2)

③**nextOrder** 函数：时间复杂度取决于 **reverse** 函数，而 **reverse** 函数的时间复杂度是 O(n)，时间复杂度为 O(n)

④整体时间复杂度为 O(n^2)，其中 n 为排列的长度

3. **遇到的问题：**这道题显然是运用递归方法，所以关键在于找到递归方程和边界条件。这也是一直卡住的地方，后来在某篇博客中了解到这题实际运用了康托展开，于是找了一篇认真的看了下，了解到康拓展开是排列序与自然数之间的双射，所以自然而然既可以从字典序得出排列，也可以从排列得到字典序，从而预测下一个排列。但是实际实现还是比较复杂，边看边理解边debug才实现。

4. **源代码：**

```
#include<iostream>
#include<algorithm>
using namespace std;
/// @brief 递归方法计算阶乘
/// @param n
```

```
/// @brief 将排列 data 的下一个字典序排列存储在 data 中
/// @param data 排列数据
/// @param n    排列长度
void nextOrder(int* data, int n){
    int i = n - 2;
```

```
/// @return 阶乘结果
int factorial(int n)
{
    if (n <= 1)return 1;
    else
        return factorial(n - 1) * n;
}

/// @brief 计算当前排列在所有排列中的顺序（递归实现）
/// @param data 排列数据
/// @param n    排列长度
/// @return 当前排列的顺序
int order(int* data, int n) {
    if (n <= 1) return 0;
    int count = 0;
    for (int i = 1; i < n; ++i) {
        if (data[0] > data[i]) count++;
    }
    return count * factorial(n - 1) + order(data + 1, n - 1);
}
```

```
    while (i >= 0 && data[i] >= data[i + 1]) i--;
    if (i >= 0) {
        int j = n - 1;
        while (data[j] <= data[i]) j--;
        swap(data[i], data[j]);
        reverse(data + i + 1, data + n);
    }
}
int main(){
    int* data = new int[1000];
    int n;
    while (cin >> n){
        for (int i = 0; i < n; ++i)
            cin >> data[i];
        cout << order(data, n) << endl;
        nextOrder(data, n);
        for (int i = 0; i < n; ++i)
            cout<<data[i]<<" ";
        cout << endl;
    }
    delete[] data;
    return 0;
}
```

2-11 整数因子分解问题

1. **题目描述：**

> 大于1的正整数n可以分解为：n=x1 * x2 * ... * xm。 例如，当n=12 时，共有8 种不同的分解式： 12=12； 12=6 * 2； 12=4 * 3； 12=3 * 4； 12=3 * 2 * 2； 12=2 * 6； 12=2 * 3 * 2； 12=2 * 2 * 3。 对于给定的正整数n，计算n共有多少种不同的分解式。

2. **算法设计：**设f(n)为n的不同分解式个数，这道题显然要使用递归分治法。因为我们可以将一个大数分解成更小的因子，而每个因子又可以再继续分解，直到无法再分解为止。下面这个递归方程就是上述的数学表述，如果i是n的因子，那么继续分解，并把所有分解f(n/i)加起来。至于边界条件，发现f(n)中会有f(n/n)=f(1)=1，所以可以设置f(n)的初始值设置为1，而不能为0.

$$\mathbf{f(n) = \sum_{n\%i==0} f(n/i),}$$
$$\mathbf{2 <= i <= n}$$

另外，由上题可以发现，其实并不需要让for循环到n，因为如果让n=12，i=2，那么n/i=6，实际上在i=2时，i=6的递归情况也开始处理了，没必要到i=6，n/i=2再反着来一遍。所以循环条件到sqrt(n)即可。最后判断一下i * i == n，因为左右因子都一样怎么交换都一样，所以只用加上f(i)即可。

```
int ans = 0;
/// @brief  递归函数，用于计算给定整数的因子数量
/// @param n  待分解的整数
void solve(int n)
{
    if (n == 1)
        ans++;
    else
        for (int i = 2; i <= sqrt(n); ++i) {
            if (n % i == 0) {
                solve(i);
                if (i != n / i)
                    solve(n / i);
            }
        }
    if (n > 1) // 处理质数的情况
        ans++;
}
```

时间复杂度主要由递归部分决定。for循环的迭代次数取决于 sqrt(n)，对于每个因子，递归调用 solve 函数两次，一次是对因子本身的处理，另一次是对 n 除以该因子的结果的处理。因此，递归树的深度为 O(log n)。
考虑到每层递归都需要 O(sqrt(n)) 的时间来确定因子，并且递归树的深度为 O(log n)，因此整个递归部分的时间复杂度为
O(sqrt(n) * log n)。

3. **遇到的问题：**

质数被忽略：处理质数情况的目的是确保质数被正确地计算其因子数量。质数是只能被1和自身整除的数，因此其只有两个因子。在递归过程中，质数不会进入for循环中的条件判断，因为其平方根大于自身，这导致质数的因子数量会被错误地计算为0。因此，为了确保质数的因子数量能够正确地被计算，需要在最后增加对质数情况的处理，将因子数量加1

4. **源代码：**

```
#include <iostream>
#include <cmath>
using namespace std;
int ans = 0;
/// @brief  递归函数，用于计算给定整数的因子数量
/// @param n  待分解的整数
```

```
void solve(int n){
    if (n == 1)
        ans++;
    else
        for (int i = 2; i <= sqrt(n); ++i) {
            if (n % i == 0) {
                solve(i);
                if (i != n / i)
                    solve(n / i);
            }
        }
    if (n > 1) // 处理质数的情况
        ans++;
}

int main(){
    int n;
    while (cin >> n) {
        ans = 0;
        solve(n);
        cout << ans << endl;
    }
    return 0;
}
```

五.总结

本次作业主要围绕递归与分治的思想展开。比方在整数因子分解问题中，递归是一种简单明了的方法，在递归过程中，通过分解问题为更小规模的子问题，不断调用自身来解决。此外，在优化代码时，观察递归树，使用分治的思想，通过筛选因子的范围，避免了重复计算，提高了代码的效率。总之，本次作业让我明白了递归与分治在解决问题时的重要性和灵活性。