算法基础Lab1实验报告

PB19071535徐昊天

一.实验内容

1. 利用随机化快速排序实现数组的字典序排序。
2. 对给定数组进行基数排序。
3. 利用FFT实现多项式乘法。

二.实验目的

1. 利用OJ编程，强化编程能力，培养编程细节，提升编程素养。
2. 熟悉书本和课堂上涉及的算法，进一步巩固算法知识。

三.实验思路

I.快速排序

对数组进行随机化快速排序可直接参考**算法导论**中的算法，并将对单个元素的比较和交换引申至对数组的比较和交换。

构造数组比较函数如下：

```
int compare(int A[3], int B[3])    // 用于比较数组元素的函数
{
    if (A[0] < B[0] || (A[0] == B[0] && A[1] < B[1]) || (A[0] == B[0] && A[1] == B[1] && A[2] < B[2]))
        return 0;
    return 1;
}
```

如上，对长度为3的A、B数组进行比较，若满足A[0] < B[0] 或 A[0] == B[0] ， A[1] < B[1] 或 A[0] == B[0] ， A[1] == B[1] ， A[2] < B[2]，则满足A < B。

构造数组交换函数如下：

```
void exchange(int A[3], int B[3])      // 用于交换数组元素的函数
{
    int temp;
    for (int i = 0;i < 3; i++)
    {
        temp = A[i];
        A[i] = B[i];
        B[i] = temp;
    }
}
```

如上，对长度为3的A、B数组分别交换每一位，即可实现交换数组。

构造数组划分函数如下：

```
int partition(int p, int r)     // 划分数组的函数
{
    int array[3] = {a[r][0], a[r][1], a[r][2]};
    int i = p - 1;
    for (int j = p; j < r; j++)
    {
        if (compare(a[j], array) == 0)
        {
            i++;
            exchange(a[i], a[j]);
        }
    }
    exchange(a[i + 1], a[r]);
    return i + 1;
}
```

参考算法导论，利用以上函数实现对子数组A[p..r]的原址重排。

构造随机抽样函数如下：

```
int randomized_partition(int p, int r)      // 随机抽样
{
    srand((unsigned)time(NULL));
    int i = rand() % (r - p + 1) + p;
    exchange(a[r], a[i]);
    return partition(p, r);
}
```

利用rand函数生成随机数，随机选择划分数组的元素。

构造快排函数如下：

```
void randomized_quicksort(int p, int r)     // 快速排序
{
    if (p < r)
    {
        int q = randomized_partition(p, r);
        randomized_quicksort(p, q - 1);
        randomized_quicksort(q + 1, r);
    }
}
```

随机划分后迭代调用快排函数，实现排序。

II.基数排序

基数排序需分别比较数组元素的每一个十进制位，进而实现排序。

构造计数排序函数如下：

```
// 计数排序，用于对每一十进制位排序
void counting_sort(int array[], int n, int digit)
{
    int temp[10] = {0};    // 提供临时存储空间
    int store[n], i, num;       // store数组存放排序的输出,num用于存储所需排序位上的数
    for (i = 0; i < n; i++)
    {
        num = (array[i] / digit) % 10;
        temp[num]++;
    }
    for (i = 1; i < 10; i++)
        temp[i] = temp[i] + temp[i - 1];
    for (i = n - 1; i >= 0; i--)
    {
        num = (array[i] / digit) % 10;
        store[temp[num] - 1] = array[i];
        temp[num]--;
    }
    for (i = 0; i < n; i++)    // 将储存在Store数组中的元素移入array数组
        array[i] = store[i];
}
```

由于计数排序时间复杂度较低，故选择利用计数排序对数组元素的十进制位比较。参数中**digit**可表示比较的位，利用**num = (array[i] / digit) % 10**将所比较的位存入num中进行排序。

构造基数排序函数如下：

```
// 基数排序函数
void radixsort(int array[], int n)
{
    int max = 0;
    for (int i = 0; i < n; i++)
    {
        if (array[i] > max)
            max = array[i];
    }  // 寻找数组中最大元素
    for (int digit = 1; max / digit != 0; digit = digit * 10)  // digit表示排序的位
    {
        counting_sort(array, n, digit);    // 调用计数排序对每一位进行排序
        if (digit == 1000000000)    // 若到达digit所能到达最大位则终止，防止digit溢出
            break;
    }
}
```

如上，查找出数组元素中的最大值后即可得到所需排序的最大位数，从个位开始逐渐向高位执行排序，并防止位数超出**int**范围导致溢出。

III.多项式乘法

对两个多项式系数执行快速傅里叶变换，得到的结果执行点值乘法后利用逆FFT算法得到两者相乘得到多项式的系数。

定义复数结构体如下：

```
// 定义复数结构体
typedef struct Complex
{
    double real;    // 实部
    double imaginary;   // 虚部
}Complex;
```

其中**real**表示实部,**imaginary**表示虚部。

构造FFT算法函数如下：

```
// 利用FFT方法计算DFT(a)
void recursive_fft(int n, Complex a[], Complex y[], int flag)   // flag为0时运用FFT，否则运用逆FFT
{
    if (n == 1)
    {
        y[0].imaginary = a[0].imaginary;
        y[0].real = a[0].real;
        return;
    }

    int num = 0;
    Complex w, w_n;

    // w = 1
    w.imaginary = 0.0;
    w.real = 1.0;

    Complex a_0[n / 2];
    Complex a_1[n / 2];
    Complex y_0[n / 2];
    Complex y_1[n / 2];
    // 以上数组分别储存偶数和奇数位的元素
    for (int i = 0; i < n / 2; i++)
    {
        a_0[i].imaginary = a[2 * i].imaginary;
        a_0[i].real = a[2 * i].real;
        a_1[i].imaginary = a[2 * i + 1].imaginary;
        a_1[i].real = a[2 * i + 1].real;
    }
    // 迭代
    recursive_fft(n / 2, a_0, y_0, flag);
    recursive_fft(n / 2, a_1, y_1, flag);

    for (int k = 0; k < n / 2; k++)
    {
```

```
        // y[k] = y_0[k] + w * y_1[k]
        y[k].imaginary = y_0[k].imaginary + w.real * y_1[k].imaginary + w.imaginary * y_1[k].real;
        y[k].real = y_0[k].real + w.real * y_1[k].real - w.imaginary * y_1[k].imaginary;
        // y[k + n / 2] = y_0[k] - w * y_1[k]
        y[k + n / 2].imaginary = y_0[k].imaginary - w.real * y_1[k].imaginary - w.imaginary *
    y_1[k].real;
        y[k + n / 2].real = y_0[k].real - w.real * y_1[k].real + w.imaginary * y_1[k].imaginary;
        // w = w * w_n
        if (flag == 0)  // FFT
            num += 2;
        else    // 逆FFT
            num -= 2;
        w.imaginary = sin((double)num / n * Pi);
        w.real = cos((double)num / n * Pi);
    }
}
```

如上，执行以上函数后y数组即储存a数组执行FFT后的结果。其中flag=0时执行FFT，否则执行逆FFT。

构造浮点数四舍五入返回整数函数如下：

```
// 用四舍五入的方式将接近答案的浮点数转化成对应的整数输出
int neartodouble(double num)
{
    int i = (int)num;
    if (num - i >= 0.5)
        return i + 1;
    else
        return i;
}
```

由于执行FFT时运算数皆为浮点数，输出值要求为整型数，故需调用以上函数实现四舍五入。

实现多项式乘法具体过程如下：

```
    recursive_fft(m, A, y_A, 0);
    recursive_fft(m, B, y_B, 0);

    // 用点值乘法得到C数组
    for (int i = 0; i < m; i++)
    {
        y_C[i].imaginary = y_A[i].imaginary * y_B[i].real + y_A[i].real * y_B[i].imaginary;
        y_C[i].real = y_A[i].real * y_B[i].real - y_A[i].imaginary * y_B[i].imaginary;
    }

    // 对C数组执行逆FFT
    recursive_fft(m, y_C, C, 1);
```

以上即为算法导论中实现多项式乘法的具体过程。

四.实验收获与感想

1. 利用OJ编程，养成了更好的编程习惯，促进自身追求更优美的代码风格，提升了编程素养。
2. 提升了编程能力，提前适应上机编程的难度，为将来刷算法题或机试打下了基础。
3. 增强了对代码细节的把控能力，对程序的追求不止于结果正确，而是追求更低的时间空间消耗、数据的完整性和精确性等。
4. 增强了对经典算法的理解，通过复现算法更加深入了解了算法的原理和过程。