算法分析题

2-3

改写二分搜索算法如下：

```
//已排好序的数组 a[0:n-1]
pair<int, int> binarySearch(int []a, int x, int n) {
    int left = 0;
    int right = n - 1;
    int i = -1;
    int j = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (a[mid] == x) {
            i = j = mid;
            break;
        } else if (a[mid] < x) {
            i = mid;
            left = mid + 1;
        } else {
            j = mid;
            right = mid - 1;
        }
    }

    // 若 x 小于数组的第一个元素
    if (i == -1 && j != -1) {
        i = -1;
    }
    // 若 x 大于数组的最后一个元素
    if (j == -1 && i != -1) {
        j = n;
    }

    return {i, j};
}
```

2-4

对于大数 u、v。当 m 比 n 小得多时，可以将 v 分成 n/m 段，每段 m 位。计算 uv 需要做 n/m 次乘法，每次 m 位* m 位的乘法可以用教材中的分治法计算，耗时 O(m^log3)。

则算法总的时间复杂度为 O((n/m)*m^log3) = O(nm^log(3/2))。

2-5

（1）证明：

设 x=2^(n/3)，u，v 及 w=uv 分别可表示为

U = u0 + u1*x + u2*x^2
v = v0 + v1*x + v2*x^2
w = w0 + w1*x + w2*x^2 + w3*x^3 + w4*x^4
将 u、v、w 都看作关于变量 x 的多项式
w0 到 w4 一共有 5 个未知的系数待求解，u0 到 u2，v0 到 v2 可看作常系数
由非线性方程解空间的规律，取不同的五个 xi 值代入多项式一定能解出 w0 到 w5（都是，ui，vi 构成的的表达式）

（2）按此分解设计的求两个 n 位大整数乘积的分治算法需要 5 次 n/3 位整数乘法。分割及并不所需的加减法和数乘运算时间为 O(n)。设 T（n）是算法所需的计算时间，则

T（n）=　O（1）　n =1，
　　　　5T（n/3）　+ O（n）　n>1

由此可得 T（n）= O（n log3（5））

2-8

可以用循环换位的方法。

代码思路如下：

```
void moving(Type a[], int n, int k){
    If( k < n - k)//当 k<n/2 时，把元素向前循环移动，需要 k 次
        for(int i = 0; i < k; i++){
            Type temp = a[0];
            for(int j = 1; j < n; j++)
                a[j-1] = a[j];
            a[n-1] = temp;
        }
    else//反之向后循环移动，需要 n-k 次
        for(int i = k; i < n; i++){
            Type temp = a[0];
            for(int j = n-1; j > 0; j--)
                a[j] = a[j-1];
            a[0] = temp;
        }

}
```

在最坏情况下，算法所需的元素移动次数为 min(k, n-k)*(n+1)。k 看作常数，除了当 k=n/2 计算时间非线性外，其他情况的时间复杂度都为 O(n)。

2-9

// 合并两个已排序的子数组

思路：使用双指针 i 和 j 分别指向两个子数组的起始位置。通过比较 a[i] 和 a[j] 的大小，如果 a[i] 小于等于 a[j]，则 i 指针后移；否则，将 a[j] 插入到合适的位置，同时 i 和 j 指针都后移。

```
void merge(int a[], int k,int n) {
```

```
    int i = 0;
    int j = k;

    while (i < j && j < n) {
        if (a[i] <= a[j]) {
            i++;
        } else {
            int temp = a[j];
            for (int m = j; m > i; m--) {
                a[m] = a[m - 1];
            }
            a[i] = temp;
            i++;
            j++;
        }
    }
}
```

复杂度分析：在最坏情况下，即前段数组所有元素都大于后段，则将后段数组整个移到前段前面，这时就与 2-8 相同，一样是 O(k*n)=O(n),故算法时间复杂度为 O(n)

算法实现题

2-1

```
pair<int, int> findMajority(int arr[], int n) {
    Int mode = arr[0], frequency = 1;

    //统计并求最大频次
    for (int i = 0; i < n; ++i) {
        int curFrequency = 1;
        for (int j = i + 1; j < n; ++j) {
            if (arr[i] == arr[j]) {
                ++curFrequency;
            }
        }
        if (curFrequency > frequency) {
            frequency = curFrequency;
            mode = arr[i];
        }
    }
    return {mode,frequency};
}
```

时间复杂度：该算法使用了两层嵌套的 for 循环。外层循环遍历数组中的每个元素，共执行 n 次，对于外层循环的每一次迭代，内层循环也会遍历数组中的剩余元素，其执行次数平均为 n/2 次。因此，总的时间复杂度为 O(n^2)。

空间复杂度：该算法只使用了常数级的额外空间，没有使用与输入规模 n 相关的额外数组或数据结构，因此空间复杂度为 (O(1))。

2-7

本题实际上就是求 Bell 数的问题，它满足递归公式 $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$，

```
// 计算二项式系数 C(n, k)
int C(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    return C(n - 1, k - 1) + C(n - 1, k);
}

// 递归计算 Bell 数 B(n)
int bellNumber(int n) {
    if (n == 0)
        return 1;
    int result = 0;
    for (int k = 0; k < n; k++) {
        result += C(n - 1, k) * bellNumber(k);
    }
    return result;
}
```

复杂度分析：

时间复杂度：由于在计算 Bell 数时，每次递归调用都会涉及到二项式系数的计算，而二项式系数的计算本身也是递归的，所以该算法的时间复杂度是指数级的，大约为 O(2^n)。

空间复杂度：递归调用会使用系统栈空间，递归深度为 n，因此空间复杂度为 O(n)。

拓展题