作业1

算法分析题1-6

1-6 对于下列各组函数$f(n)$和$g(n)$，确定$f(n)=O(g(n))$或$f(n)=\Omega(g(n))$或$f(n)=\theta(g(n))$，并简述理由。

（1）$f(n)=\log n^2$；	$g(n)=\log n+5$	（5）$f(n)=10$；	$g(n)=\log 10$
（2）$f(n)=\log n^2$；	$g(n)=\sqrt{n}$	（6）$f(n)=\log^2 n$；	$g(n)=\log n$
（3）$f(n)=n$；	$g(n)=\log^2 n$	（7）$f(n)=2^n$；	$g(n)=100n^2$
（4）$f(n)=n\log n+n$；	$g(n)=\log n$	（8）$f(n)=2^n$；	$g(n)=3^n$

$(1) f(n) = logn^2，\ g(n) = logn + 5$

根据对数性质$f(n) = 2logn$，而$g(n)$主要受$logn$控制，当n很大时，常数项+5对n的增长影响很小，$g(n) \approx logn$，因此，$f(n)$和$g(n)$是同阶函数。只是$f(n)$多了一个常数因子2。所以：

$$f(n) = \Theta(g(n))$$

$(2) f(n) = logn^2，\ g(n) = logn + 5$

同样，$f(n) = 2logn$，而$g(n) = \sqrt{}(n)$，当$n \to \infty$，$\sqrt{}(n)$增长比$logn$快，因此：

$$f(n) = O(g(n))$$

$(3) f(n) = n，\ g(n) = log^2 n$

$f(n)$是线性函数，而$g(n)$是对数的平方，当$n \to \infty$时，n的增长速率远快于$log^2 n$，因此：

$$f(n) = \Omega(g(n))$$

$(4) f(n) = nlogn + n，\ g(n) = logn$

$f(n)$中的线性项n使得增长速度远大于$g(n)$，因为$g(n)$只是一个对数函数。因此：

$$f(n) = \Omega(g(n))$$

$(5) f(n) = 10，\ g(n) = log10$

这是两个常数，他们的数量级相同。因此：

$$f(n) = \Theta(g(n))$$

$(6) f(n) = log^2 n，\ g(n) = logn$

$f(n)$是$g(n)$的平方，因此$f(n)$增长比$g(n)$快得多。因此：

$$f(n) = \Omega(g(n))$$

$(7) f(n) = 2^n，\ g(n) = 100n^2$

$f(n)$是指数函数，$g(n)$是多项式函数。当n增加时，指数函数的增长速度要远远快于任何多项式函数。因此：

$$f(n) = \Omega(g(n))$$

$(8) f(n) = 2^n，\ g(n) = 3^n$

二者都是指数函数，底数大的增长较快。因此：

$$f(n) = O(g(n))$$

算法实现题1-1

1-1　统计数字问题。

问题描述：一本书的页码从自然数 1 开始顺序编码直到自然数 n。书的页码按照通常的习惯编排，每个页码都不含多余的前导数字 0。例如，第 6 页用数字 6 表示而不是 06 或 006 等。数字计数问题要求对给定书的总页码 n，计算书的全部页码分别用到多少次数字 0, 1, 2, …, 9。

算法设计：给定表示书的总页码的十进制整数 n（$1 \leqslant n \leqslant 10^9$），计算书的全部页码中分别用到多少次数字 0, 1, 2, …, 9。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有 1 行，给出表示书的总页码的整数 n。

结果输出：将计算结果输出到文件 output.txt。输出文件共 10 行，在第 k（k=1, 2, …, 10）行输出页码中用到数字 k–1 的次数。

输入文件示例	输出文件示例
input.txt	output.txt
11	1
	4
	1
	1
	1
	1
	1
	1
	1
	1

Ⅰ. 暴力枚举

1.思路

从1到n遍历所有页码，分离页码的每一位数字进行统计。

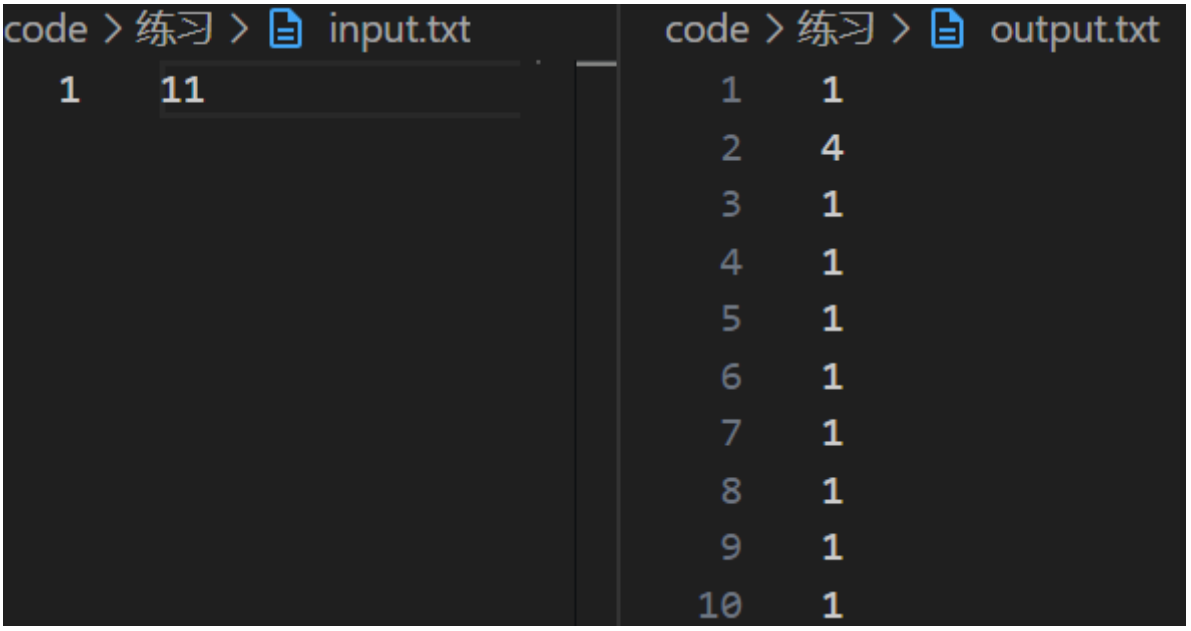
2.代码

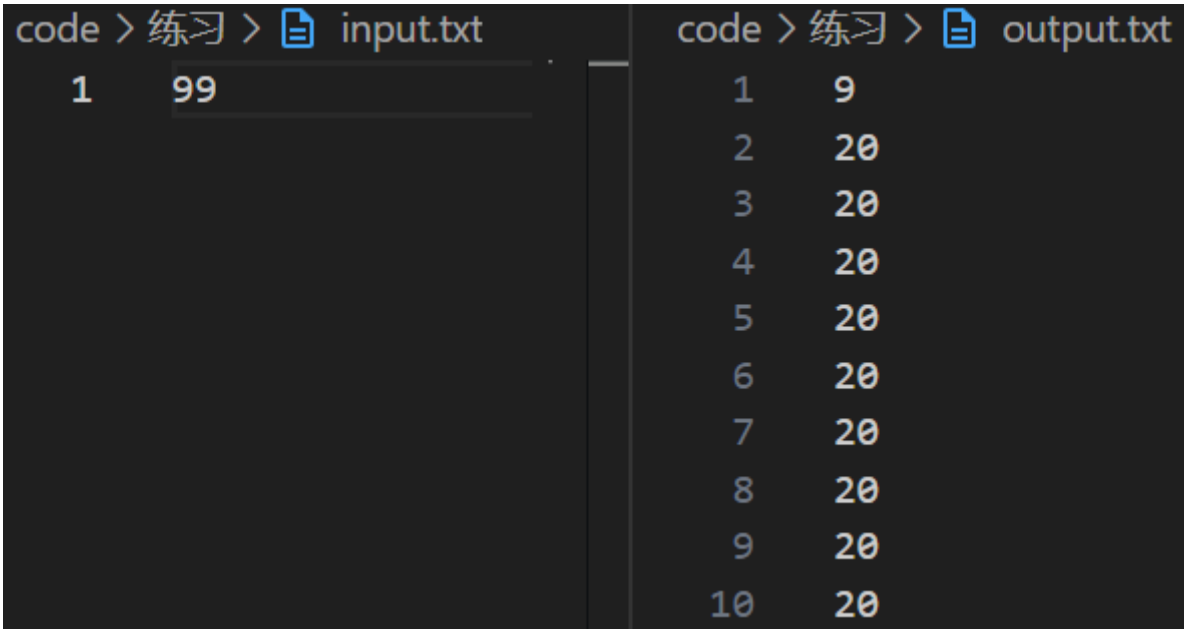
```
//暴力解法
#include<bits/stdc++.h>
#define endl '\n'
typedef long long ll;
using namespace std;
ll a[10]; //统计
int n;
void func(ll y) {
    while(y) a[y % 10]++, y /= 10; //统计该页码中出现的数字
}
int main() {
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n;
    for(ll i = 1; i <= n; i++) { //遍历所有页码
        func(i);
    }
    for(int i = 0 ; i <= 9; i++) ofs << a[i] << endl;
```

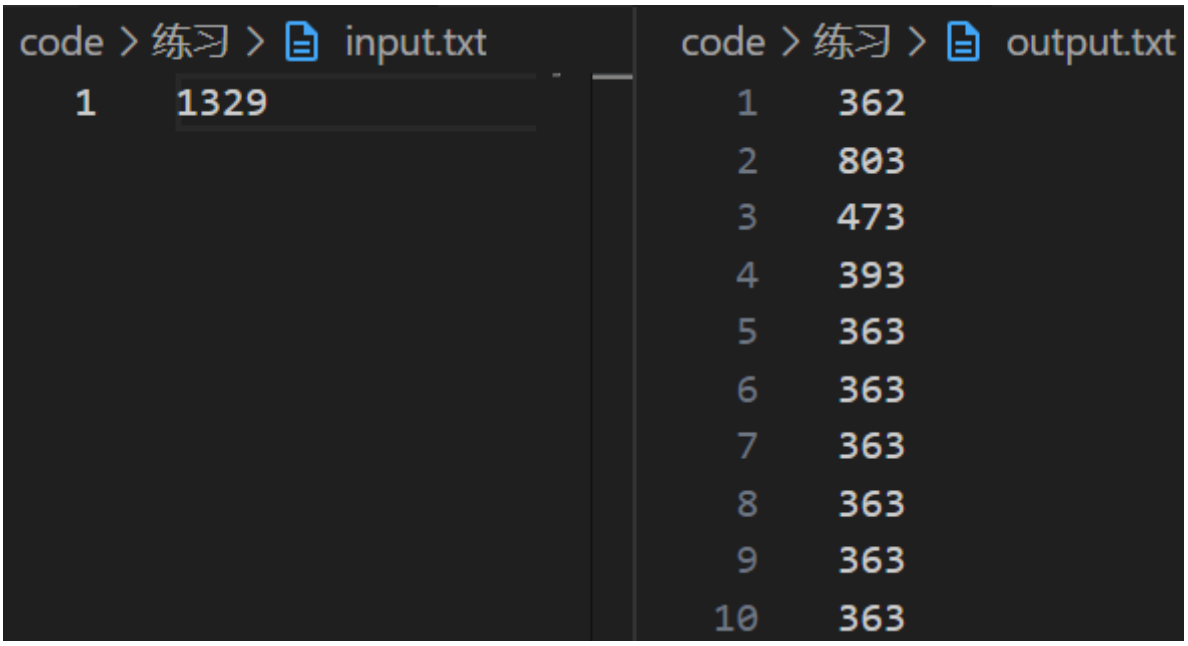
```
    ifs.close();
    ofs.close();

    return 0;
}
```

3.运行结果







4.分析

接下来分析算法的复杂度。

时间复杂度分析：

循环：

```
for (ll i = 1; i <= n; i++) {
    func(i);
}
```

循环从1到n进行遍历，总共执行了n次。

`func`函数的复杂度：

```
void func(ll y) {
    while(y) a[y % 10]++, y /= 10;
}
```

func函数的作用是将数字y分解为它的各个位数，并统计每个数字出现的个数。对于一个数字y，它的位数是$logy$，因此func函数的时间复杂度是$O(logy)$。

因此，总的时间复杂度是：

$$O(1)+O(log2)+O(log3)+\cdots+O(logn)=O(\sum_{i=1}^{n}log(i!))=O(log(n!))$$

根据斯特林公式，当n很大时：

$$n!=\sqrt{2\pi n}(\frac{n}{e})^n$$

故

$$log(n!)=nlogn-nloge+\frac{1}{2}log2\pi+\frac{1}{2}logn$$

因此，总的时间复杂度为：

$$O(nlogn)$$

空间复杂度分析：

数组 `a[10]` 是用来存储每个数字（0-9）出现的次数，固定大小为 10。额外的变量 `n`、`y` 以及用于文件输入输出的流对象都是常量级的内存消耗。因此，空间复杂度为：

$$O(1)$$

总结：

时间复杂度：$O(nlogn)$

空间复杂度：$O(1)$

当n很大时(例如10^9以上)，时间复杂度可能会导致运行时间过长。

II. 数位DP

1.思路

先引入一个概念，什么是`数位`？数位是指把一个数字按照个、十、百、千等等一位一位地拆开，关注它每一位上的数字。如果拆的是十进制数，那么每一位数字都是 0~9，其他进制可类比十进制。

`数位DP的基本原理`：考虑人类计数的方式，最朴素的计数就是从小到大开始依次加一。但我们发现对于位数比较多的数，这样的过程中有许多重复的部分。例如，从 7000 数到 7999、从 8000 数到 8999、和从 9000 数到 9999 的过程非常相似，它们都是后三位从 000 变到 999，不一样的地方只有千位这一位，所以我们可以把这些过程归并起来，将这些过程中产生的计数答案也都存在一个通用的数组里。此数组根据题目具体要求设置状态，用递推或 DP 的方式进行状态转移。

发现对于满i位的数，所有数字出现的次数都是相同的，故设数组$dp[i]$为满i位的数中每个数字出现的次数。

i	$dp[i]$
1	1
2	20
3	300
4	4000

有$dp[i]=10\times dp[i-1]+10^{i-1}$，贡献分为两部分，前$i-1$位数字的贡献，第$i$位数字的贡献。

有了dp数组，我们来考虑如何统计答案。从高位到低位逐位处理，对于第i位，先计算前$i-1$位的贡献，在计算第前位的贡献。

2.代码

```
#include<bits/stdc++.h>
#define endl '\n'
typedef long long ll;
const int N = 13;
using namespace std;
ll n, dp[N], weight[N], ans[N];
int a[N];
```

```
void solve(ll n) {
    ll _n = n;
    int len = 0;
    while(n) a[++len] = n % 10, n /= 10;
    for(int i = len; i >= 1; i--) {
        //前i-1位的贡献
        for(int j = 0; j < 10; j++) ans[j] += dp[i - 1] * a[i];
        //第i位的贡献
        for(int j = 0; j < a[i]; j++) ans[j] += weight[i - 1];
        _n -= weight[i - 1] * a[i];
        ans[a[i]] += _n + 1;
        ans[0] -= weight[i - 1];
    }
}
int main() {
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n;
    weight[0] = 1ll;
    for(int i = 1; i <= 10; ++i) {
        dp[i] = dp[i - 1] * 10 + weight[i - 1];
        weight[i] = weight[i - 1] * 10;
    }
    solve(n);
    for(int i = 0; i < 10; i++) ofs << ans[i] << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

核心代码解释：

```
    for(int i = 1; i <= 10; ++i) {
        dp[i] = dp[i - 1] * 10 + weight[i - 1];
        weight[i] = weight[i - 1] * 10;
    }
```

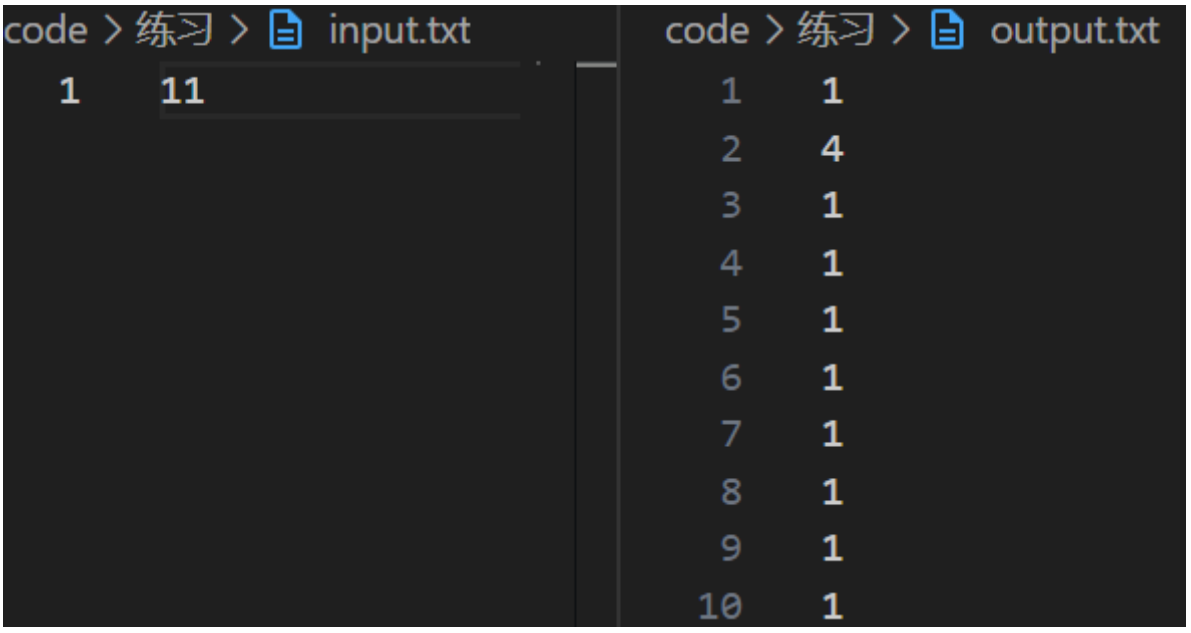
这部分代码用来计算dp数组，其中$weight$表示当前位的权重。$weight[0] = 1$，$weight[1] = 10$，$weight[2] = 100 \cdots$

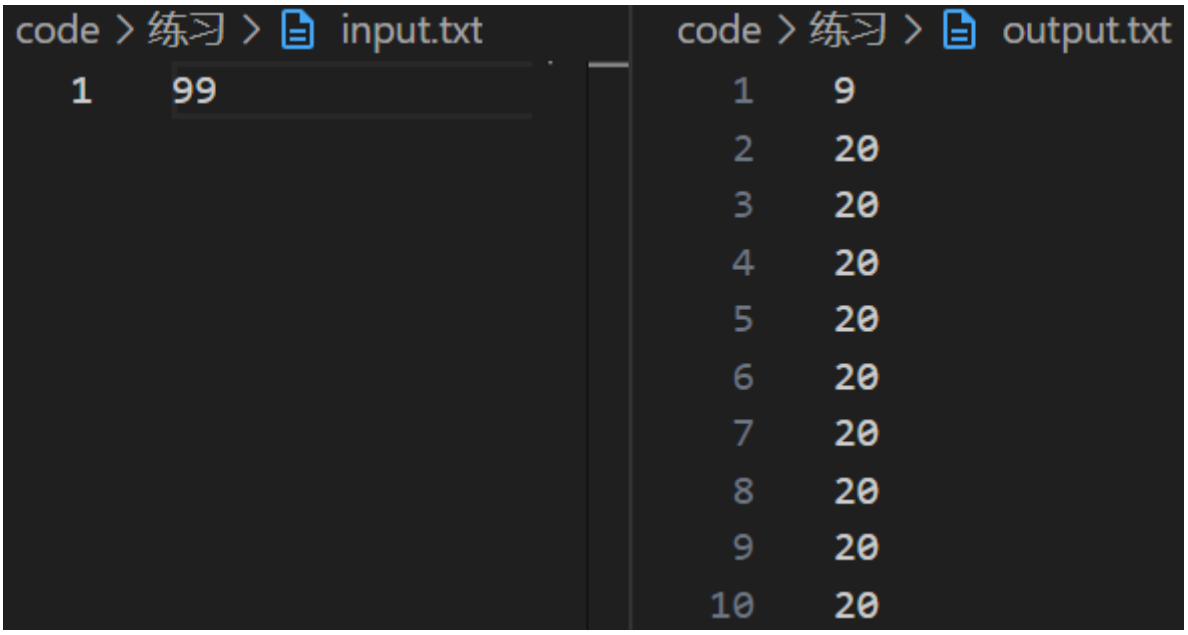

```
void solve(ll n) {
    ll _n = n;
    int len = 0;
    while(n) a[++len] = n % 10, n /= 10;
    for(int i = len; i >= 1; i--) {
        //前i-1位的贡献
        for(int j = 0; j < 10; j++) ans[j] += dp[i - 1] * a[i];
        //第i位的贡献
        for(int j = 0; j < a[i]; j++) ans[j] += weight[i - 1];
        _n -= weight[i - 1] * a[i];
        ans[a[i]] += _n + 1;
        ans[0] -= weight[i - 1];
    }
}
```

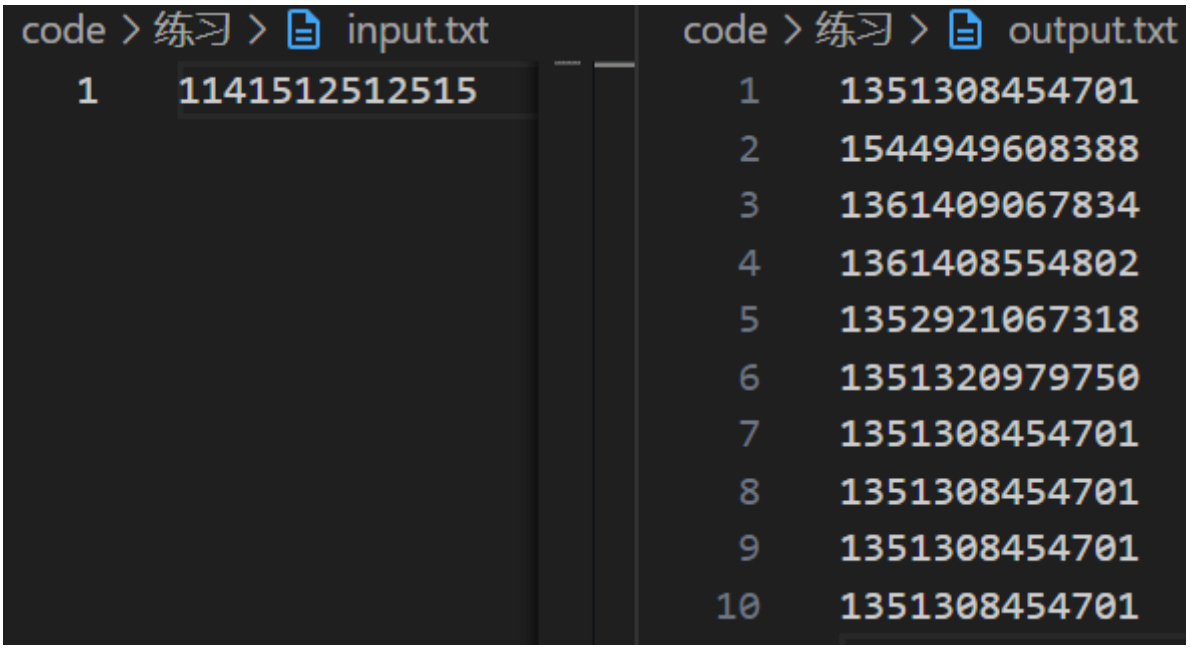
这部分是算法的核心代码：

1. 首先使用`while(n)`循环，将数字n按位拆分，从低位到高位依次存入数组`a`中。
2. 从最高位(`len`)按位处理。
3. 对于第i位，计算前$i-1$的贡献，即`ans[j] = dp[i - 1] * a[i]`。
4. 对于当前位之前的所有数字，累加它们的贡献。即`ans[j] += weight[i - 1]`。
5. 当前位a[i]的贡献需要特别处理，通过`_n -= weight[i - 1] * a[i]`逐步更新剩下的部分，接着对当前位a[i]的数进行加权处理，即`ans[a[i]] += _n + 1`。
6. 最终调整0的贡献，减去权重，即`ans[0] -= weight[i - 1]`。

3.运行结果







4.分析

时间复杂度分析：

分析算法过程，首先通过`while(n)`将数字拆分为各个位数，时间复杂度为$O(L)$，其中L是数字n的位数，约为$logn$。接着是循环进行逐位处理，循环次数位L次，循环体内的操作都是`常数级`的。因此，时间复杂度为：

$$O(logn)$$

最坏情况下，n是10^9，复杂度为常数级别$O(9)$

空间复杂度分析：

数组a用来存放数组的每一位，长度为数字的位数，故空间复杂度为

$$O(logn)$$

总结：

时间复杂度：$O(logn)$

空间复杂度：$O(logn)$

对于$n <= 10^{12}$，复杂度基本是常数级别，算法较为高效。

算法实现题1-2

1-2　字典序问题。

问题描述：在数据加密和数据压缩中常需要对特殊的字符串进行编码。给定的字母表 A 由 26 个小写英文字母组成，即 $A=\{a, b, \cdots, z\}$。该字母表产生的升序字符串是指字符串中字母从左到右出现的次序与字母在字母表中出现的次序相同，且每个字符最多出现 1 次。例如，a、b、ab、bc、xyz 等字符串都是升序字符串。现在对字母表 A 产生的所有长度不超过 6 的升序字符串按照字典序排列并编码如下。

1	2	…	26	27	28	…
a	b	…	z	ab	ac	…

对于任意长度不超过 6 的升序字符串，迅速计算出它在上述字典中的编码。

算法设计：对于给定的长度不超过 6 的升序字符串，计算它在上述字典中的编码。

数据输入：输入数据由文件名为 input.txt 的文本文件提供。文件的第 1 行是一个正整数 k，表示接下来有 k 行。在接下来的 k 行中，每行给出一个字符串。

结果输出：将计算结果输出到文件 output.txt。文件有 k 行，每行对应一个字符串的编码。

输入文件示例	输出文件示例
input.txt	output.txt
2	1
a	2
b	

1.思路

生成字典的过程可以通过递归的方式来实现。递归的思路是：从空字符串开始，每次增加一个比当前字符串最后一个字符大的新字符，直到字符串长度达到6为止。递归生成的字典顺序并非正确，还需要经过一次排序。

2.代码

```
#include<bits/stdc++.h>
#include<windows.h>
#include <synchapi.h>
#define endl '\n'
typedef long long ll;
using namespace std;
vector<string> vt;
int k;
string str;
ll ans;
bool cmp(string a, string b) {
    if(a.size() != b.size())
        return a.size() < b.size();
    else
        return a < b;
}
void generate(string s, char ch) {
    if(s.size() > 6) return;
    if(!s.empty()) vt.push_back(s);
    for(char c = ch + 1; c <= 'z'; ++c)
        generate(s + c, c);
}
int main() {
    generate("", 'a' - 1);
```

```
    sort(vt.begin(), vt.end(), cmp);
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> k;
    for(int i = 0; i < k; ++i) {
        ifs >> str;
        auto it = find(vt.begin(), vt.end(), str);
        if(it != vt.end()) {
            ans = distance(vt.begin(), it) + 1;
        } else {
            ans = -1;
        }
        ofs << ans << endl;
    }
    ////输出所有字典序
    //for(int i = 0; i < vt.size(); ++i) {
    //    ofs << i + 1 << ' ' << vt[i] << endl;
    //}
    // 关闭文件
    ifs.close();
    ofs.close();

    return 0;
}
```

核心代码解释：

```
void generate(string s, char ch) {
    if(s.size() > 6) return;
    if(!s.empty()) vt.push_back(s);
    for(char c = ch + 1; c <= 'z'; ++c)
        generate(s + c, c);
}
```

当前字符串 `s` 的最后一个字符是 `ch`，下一次递归中只允许加入比 `ch` 大的字符。
例如，如果 `s = "a"`，则接下来只允许加入 `'b'` 到 `'z'` 中的字符生成组合。对于每一个新生成的组合（如 `"ab"`），再次递归，继续向其后面添加更大的字符，生成 `"abc"`，`"abd"` 等组合。

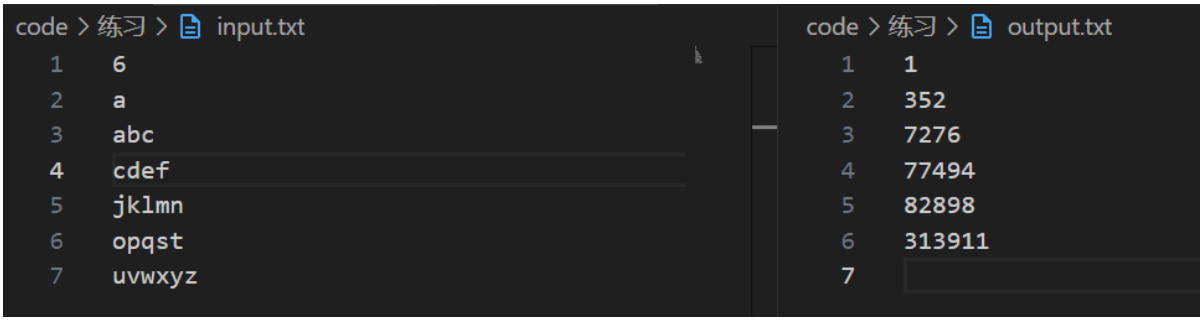
```
bool cmp(string a, string b) {
    if(a.size() != b.size())
        return a.size() < b.size();
    else
        return a < b;
}
```

生成的字典为a，ab，abc，abcd，…，为了实现正确的字典序，需要对生成的字典进行排序。

```
    for(int i = 0; i < k; ++i) {
        ifs >> str;
        auto it = find(vt.begin(), vt.end(), str);
        if(it != vt.end()) {
            ans = distance(vt.begin(), it) + 1;
        } else {
            ans = -1;
        }
        ofs << ans << endl;
    }
```

find()函数查找，找到则返回下标+1，即对应编号。

3.运行结果



4.分析

时间复杂度分析：

递归的时间复杂度取决于生成过程中每一层递归调用的次数和递归深度。可以将递归生成字符串视为一种`组合选择问题`。从26个字母中选择1到6个字母，按字典序排列。对于每个字符，后续递归时可用的字符集会逐步减少。组合的总数为

$$n = C_{26}^1 + C_{26}^2 + C_{26}^3 + C_{26}^4 + C_{26}^5 + C_{26}^6$$

递归每次递进一层时，会生成多个分支，整体递归遍历的次数等同于最终生成的字符串数量。因此，递归调用的次数是**字符串组合数的总和**。故字典生成的时间复杂度为$O(n)$。

在生成字典后，对其进行排序从而得到正确的字典序列，sort()的时间复杂度为$O(nlogn)$。find()的查找过程是一个线性查找，时间复杂度是$O(n)$。

故时间复杂度为：

$$O(nlogn)$$

空间复杂度分析：

空间复杂度主要由递归调用栈和存储字典序列的容器（`vector<string> vt`）决定。

递归调用时，每进入一次函数，都会占用栈空间。递归深度取决于生成的字符串的最大长度（不超过6），因此：递归的最大深度为6。每次递归会占用栈空间用于保存局部变量，包括当前的字符串`s`和字符`ch`。

因此，递归调用栈的空间复杂度为 **O(6)**，这是因为最长的递归深度不超过6，常数级别的栈空间占用。

字典生成后的所有字符串都存储在`vector<string> vt`中，因此，存储所有字典序列所占用的空间为$O(n)$

故空间复杂度为：

$O(n)$

总结：

时间复杂度：$O(nlogn)$

空间复杂度：$O(n)$

其中，n为字符串的数量，$n = C_{26}^1 + C_{26}^2 + C_{26}^3 + C_{26}^4 + C_{26}^5 + C_{26}^6$