F

> 最长公共子序列问题，要求输出方案。
>
> 记　　　　　　　　　　　　。
>
> 可以　　　　　　　解决。
>
> 关于输出方案，在状态转移的时候记录一下来源，然后递归输出。

也可以不用递归，从最后一位开始往前跳，过程中把答案压到栈里。

```cpp
#include <iostream>
#include <algorithm>
#include <string>

typedef std::string str;
typedef std::pair<int, int> pii;
typedef char chr;

const int maxN = 3000;
const int maxM = 3000;

str s, t;
int f[maxN + 10][maxM + 10];
pii g[maxN + 10][maxM + 10];
chr h[maxN + 10][maxM + 10];

int main() {
        std::cin >> s >> t;
        for (int i = 0; i <= s.size(); i++) {
                for (int j = 0; j <= t.size(); j++) {
                        if (i) {
                                if (f[i][j] < f[i - 1][j]) {
                                        f[i][j] = std::max(f[i][j], f[i - 1][j]);
                                        g[i][j] = std::make_pair(i - 1, j);
                                }
                        }
                        if (j) {
                                if (f[i][j] < f[i][j - 1]) {
                                        f[i][j] = std::max(f[i][j], f[i][j - 1]);
                                        g[i][j] = std::make_pair(i, j - 1);
                                }
                        }
                        if (i && j) {
                                if (f[i][j] < f[i - 1][j - 1]) {
                                        f[i][j] = std::max(f[i][j], f[i - 1][j - 1]);
                                        g[i][j] = std::make_pair(i - 1, j - 1);
                                }
                        }
                        if (i && j && s[i - 1] == t[j - 1]) {
                                if (f[i][j] < f[i - 1][j - 1]+1) {
                                        f[i][j] = std::max(f[i][j], f[i - 1][j - 1]+1);
                                        g[i][j] = std::make_pair(i - 1, j - 1);
                                        h[i][j] = s[i - 1];
                                }
                        }
                }
        }
        int len = - 1;
        pii cur;
        for (int i = 0; i <= s.size(); i++) {
```

```
                for (int j = 0; j <= t.size(); j++) {
                        if (len < f[i][j]) {
                                len = f[i][j];
                                cur = std::make_pair(i, j);
                        }
                }
        }
        str ans = "";
        while(cur != std::make_pair(0,0)) {
                int x = g[cur.first][cur.second].first;
                int y = g[cur.first][cur.second].second;
                if ('a' <= h[cur.first][cur.second] && h[cur.first][cur.second] <= 'z')
                        ans.push_back(h[cur.first][cur.second]);
                }
                cur = std::make_pair(x, y);
        }
        std::reverse(ans.begin(), ans.end());
    std::cout << ans << '\n';
    return 0;
}
```

> “正面朝上的银币数比反面朝上的银币数多” 这一事件包含了很多样本点，尝试统计每个样本点的概率后求和。
>
> 记 表示 次正面向上， 次反面向上的概率，初始时 。
>
> 答案为 。
>
> 还有另一种方法，记 表示前 枚硬币有 次朝上的概率，初始时 。
>
> 答案为 。

这里给出第二种代码实现。

```cpp
#include <iostream>
#include <iomanip>

typedef double dbl;

const int maxN = 3000;

int n;
dbl p[maxN + 10];
dbl f[maxN + 10][maxN + 10];
dbl ans;

int main() {
    std::cin >> n;
    for (int i = 1; i <= n; i++) std::cin >> p[i];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
                for (int j = 0; j <= i - 1; j++) {
                        f[i][j + 1] += f[i - 1][j] * p[i];
                        f[i][j] += f[i - 1][j] * (1.0 - p[i]);
                }
        }
        for (int i = 0; i <= n; i++) if (i > n-i) ans += f[n][i];
    std::cout << std::setiosflags(std::ios::fixed);
    std::cout << std::setprecision(12) << ans << '\n';
    return 0;
}
```

O

> 考虑状压。
>
> 单压一个集合就有 个状态，两个集合都压肯定压不下。
>
> 由于求完备匹配数，一个元素就算现在不匹配早晚都得匹配。
>
> 于是记 表示一个集合用前 个元素匹配了另一个集合的元素，且另一个集合已匹配的元素压缩状态表示为 的方案数。
>
> 枚举 需要 ，一次转移需要 ，总时间复杂度 ，考虑优化。
>
> 当且仅当 时一个状态是合法的，则只需枚举 的 ，符合条件的状态 可以预处理。
>
> 时间复杂度 。

- 若先枚举 ，则只需枚举 的 ，符合条件的状态 可以预处理。
- 若先枚举 ，则无需枚举 ，令 即可。

```
#include <iostream>

const int maxn = 21;
const int mod = 1000000007;

int n;
int a[maxn + 5][maxn + 5];
int f[(1 << (maxn + 1)) + 10];

int main() {
    std::cin >> n;
        for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) std::cin >> a[i][j];
        f[0] = 1;
        for (int i = 0; i < (1 << n); i++) {
                int c = 0;
                for (int j = 0; j < n; j++) if (i & (1 << j)) c++;
                for (int j = 0; j < n; j++) if (~i & (1 << j)) {
                        if (a[c][j]) {
                                f[i + (1 << j)] = (f[i + (1 << j)] + f[i]) % mod;
                        }
                }
        }
    std::cout << f[(1 << n) - 1] << '\n';
        return 0;
}
```

S

> 统计一定范围内符合条件的数字数量显然用数位 DP，但是要求数字是 的倍数不太好搞。
>
> 于是扩展状态表示，把模 的余数也加进去就行了。

这里给出数位 DP 的循环写法，相比于记忆化搜索码量小。

```cpp
#include <iostream>
#include <string>

typedef std::string str;
typedef long long lxl;

const int maxN = 10000;
const int maxM = 100;
const int mod = 1e9 + 7;

str s;
int d;
lxl f[maxN + 10][maxM + 10][2];

int main(){
        std::cin >> s;
        std::cin >> d;
        f[0][0][0] = 1;
        for (int i = 0; i < s.size(); i++){
                int pos = s[i]-'0';
                for (int j = 0; j < d; j++){
                        for (int a = 0; a <= pos;a++){
                                if (a == pos) f[i + 1][(j + a) % d][0] += f[i][j][0];
                                else f[i + 1][(j + a) % d][1] += f[i][j][0];
                        }
                        for (int a = 0; a <= 9; a++) {
                                f[i + 1][(j + a) % d][1] += f[i][j][1];
                        }
                }
        for (int j = 0; j < d; j++) for (int k = 0; k < 2; k++) f[i + 1][j][k] %= mod;
        }
        lxl ans = - 1ll + f[s.size()][0][0] + f[s.size()][0][1];
        std::cout << (ans % mod + mod) % mod << '\n';
    return 0;
}
```