【算法分析】

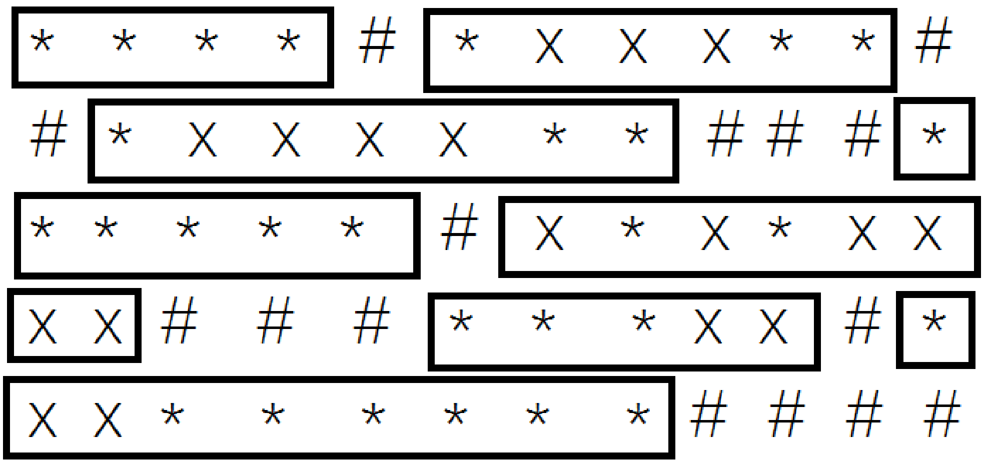
先考虑没有硬石头的情况。

如果没有硬石头，那么问题就相当于软石头上不能放炸弹，求最多能放多少个炸弹，也就相当于有一些二元组(x,y)，要选出这些二元组的一个最大的子集，使得这个子集内任意两个二元组的x不同，y也不同。

可以看出，这是一个显然的二分图匹配模型。对于每个二元组(x,y)，由二分图第一部中的节点x向第二部中的节点y连一条边之后，答案就是最大匹配。

如果有硬石头，我们也使用一样的思想。

先在行上，把连续不包含硬石头的连通块标记出来，如图



在列上也是一样。

那么就相当每个空地有一个二元组(x,y)表示这个空地属于第x个横向的连通块和第y个纵向的连通块。

同样地，对于每个二元组(x,y)，由第一部中点x连向第二部中点y，答案为最大匹配数。

【参考程序】

```
// 陈栉旷
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
using namespace std;
inline int read() {
    int res = 0; bool bo = 0; char c;
    while (((c = getchar()) < '0' || c > '9') && c != '-');
    if (c == '-') bo = 1; else res = c - 48;
    while ((c = getchar()) >= '0' && c <= '9')
        res = (res << 3) + (res << 1) + (c - 48);
    return bo ? ~res + 1 : res;
}
```

```
inline char get() {
    char c; while ((c = getchar()) != '*' && c != 'x' && c != '#');
    return c;
}
const int N = 55, M = 2505;
int n, m, tot, ecnt, nxt[M], adj[M], go[M], my[M], row[N][N], col[N][N];
bool vis[M]; char a[N][N];
void add_edge(int u, int v) {
    nxt[++ecnt] = adj[u]; adj[u] = ecnt; go[ecnt] = v;
}
bool dfs(int u) {
    for (int e = adj[u], v; e; e = nxt[e])
        if (!vis[v = go[e]]) {
            vis[v] = 1;
            if (!my[v] || dfs(my[v])) {
                my[v] = u;
                return 1;
            }
        }
    return 0;
}
int solve() {
    int i, ans = 0;
    for (i = 1; i <= tot; i++) {
        memset(vis, 0, sizeof(vis));
        if (dfs(i)) ans++;
    }
    return ans;
}
int main() {
    int i, j; n = read(); m = read();
    for (i = 1; i <= n; i++) for (j = 1; j <= m; j++)
        a[i][j] = get();
    for (i = 1; i <= n; i++) for (j = 1; j <= m; j++) {
        if (a[i][j] == '#') continue;
        if (j == 1 || a[i][j - 1] == '#') tot++;
        row[i][j] = tot;
    }
    for (j = 1; j <= m; j++) for (i = 1; i <= n; i++) {
        if (a[i][j] == '#') continue;
        if (i == 1 || a[i - 1][j] == '#') tot++;
        col[i][j] = tot;
    }
    for (i = 1; i <= n; i++) for (j = 1; j <= m; j++) {
```

```
        if (a[i][j] != '*') continue;
        add_edge(row[i][j], col[i][j]);
    }
    printf("%d\n", solve());
    return 0;
}
```

【算法分析】

二分答案，问题转变为：限制简称的最大长度，判断是否存在方案。

注意到如果一个串有不少于 n 个不同的子序列，那么无论其它串选择什么简称，该串都能分配到至少一个简称，因此对于一个串的简称方案，我们只需要考虑它至多 n 个的不同子序列。

于是我们搜索出每个串至多 n 个的不同子序列，把每种子序列和每个串分别看做二分图两边的点，若一个串中含有某种子序列，则把这种子序列所代表的点向这个串所代表的点连边，则每个串都能分配到一个简称的条件即为图中的最大匹配数等于 n。

搜索之前先建出每个串的序列自动机，则搜索时只需要在序列自动机走，每走一步都是一种不同的子序列，则每次搜索所有串的子序列的复杂度为 $O(26n^2)$，又因为二分图的点数和边数均为 $O(n^2)$ 级别，若使用匈牙利算法求最大匹配，总的时间复杂度为 $O(n^4 \log n)$，但实际情况下远远达不到这个上界，是可以通过的。

【参考程序】

```
//陈贤
#include <bits/stdc++.h>

template <class T>
inline void read(T &res)
{
    char ch;
    while (ch = getchar(), !isdigit(ch));
    res = ch ^ 48;
    while (ch = getchar(), isdigit(ch))
        res = res * 10 + ch - 48;
}

using std::vector;
using std::string;
const int N = 305;
const int M = 1e5 + 5;

int n, len, lim, E, T, bm, qr, src, tis;
int G[M][26], tag[M], sl[N], vis[M], mth[M];
char s[N];
string sa[N][N], b[M], sans[N];
vector<int> v[M];
```

```
inline bool Find(int x) //匈牙利算法求最大匹配
{
    for (int i = 0, y, im = v[x].size(); i < im; ++i)
        if (vis[y = v[x][i]] != tis)
        {
            vis[y] = tis;
            if (!mth[y] || Find(mth[y]))
                return mth[x] = y, mth[y] = x, true;
        }
    return false;
}

struct node
{
    int ch[26], par;

    inline void Clear()
    {
        memset(ch, 0, sizeof(ch));
        par = 0;
    }
};

struct SAM //序列自动机
{
    node tr[N];
    int T, lst[26];

    inline void Init()
    {
        tr[T = 1].Clear();
        for (int i = 0; i < 26; ++i)
            lst[i] = 1;
    }

    inline void insert(int c) //插入字符
    {
        tr[++T].Clear();
        tr[T].par = lst[c];

        for (int i = 0; i < 26; ++i)
            for (int j = lst[i]; j && !tr[j].ch[c]; j = tr[j].par)
                tr[j].ch[c] = T;
```

```
        lst[c] = T;
    }
}p[N];

inline void Dfs(int id, int x, int k, string a) //搜索串的子序列
{
    if (sl[id] >= n)
        return ;
    if (k > 1)
        sa[id][++sl[id]] = a;
    if (k > lim)
        return ;

    for (int i = 0; i < 26; ++i)
        if (p[id].tr[x].ch[i])
            Dfs(id, p[id].tr[x].ch[i], k + 1, a + (char)(i + 'a'));
}

inline int insert_Trie(const string &a)
{ //需要对搜索出的子序列进行去重，这里用 Trie 来实现
    int x = 1, y;
    for (int i = 0, im = a.size(); i < im; ++i)
    {
        y = a[i] - 'a';
        if (!G[x][y])
        {
            G[x][y] = ++E;
            tag[E] = 0;
            memset(G[E], 0, sizeof(G[E]));
        }
        x = G[x][y];
    }
    if (!tag[x])
        b[tag[x] = ++T] = a;
    return tag[x];
}

inline bool check(int mid)
{
    lim = mid;
    for (int i = 1; i <= n; ++i)
        sl[i] = 0;

    string a = "";
```

```
    for (int i = 1; i <= n; ++i)
        Dfs(i, 1, 1, a);

    memset(G[E = 1], 0, sizeof(G[1]));
    T = n;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= sl[i]; ++j)
            v[i].push_back(insert_Trie(sa[i][j]));
    for (int i = 1; i <= T; ++i)
        mth[i] = 0;
    int cnt = 0;
    for (int i = 1; i <= n; ++i)
    {
        ++tis;
        Find(i) ? ++cnt : 0;
    }

    for (int i = 1; i <= n; ++i)
        v[i].clear();
    if (cnt == n)
    {
        for (int i = 1; i <= n; ++i)
            sans[i] = b[mth[i]];
        return true;
    }
    else
        return false;
}

int main()
{
    freopen("diff.in", "r", stdin);
    freopen("diff.out", "w", stdout);

    read(n);
    int l = 1, r = 0, ans = 0;
    for (int i = 1; i <= n; ++i)
    {
        scanf("%s", s + 1);
        len = strlen(s + 1);
        len > r ? r = len : 0;
        p[i].Init();
        for (int j = 1; j <= len; ++j)
            p[i].insert(s[j] - 'a');
```

```
    }

    while (l <= r)
    {
        int mid = l + r >> 1;
        if (check(mid))
            ans = mid, r = mid - 1;
        else
            l = mid + 1;
    }

    if (ans)
    {
        printf("%d\n", ans);
        for (int i = 1; i <= n; ++i)
            std::cout << sans[i] << std::endl;
    }
    else
        puts("-1");

    fclose(stdin); fclose(stdout);
    return 0;
}
```

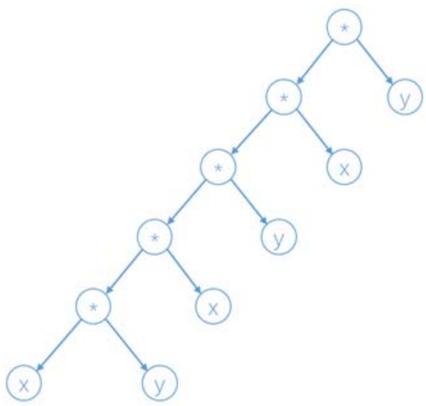
【算法分析】

首先我们知道对于一个表达式$A_1 * A_2 * \ldots * A_m$ （$*$表示任意运算符，$A_1, A_2, \ldots, A_m$表示任意表达式），我们可以任意交换A_i 之间的排列顺序。

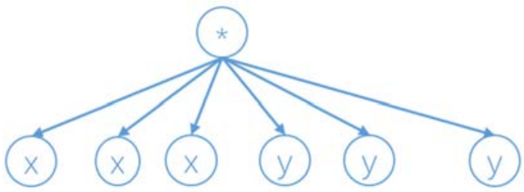
我们考虑利用表达式树这个工具，对于一个中缀表达式 $A * B$，以$*$为根，A与B为左右子树建树，然后将 A, B这两个表达式递归处理。后缀表达式同理。

由于该题的特性，我们可以把树上相邻两个运算符结点合并，将他们的儿子也合并，表示他们的儿子可以以任意顺序排列。

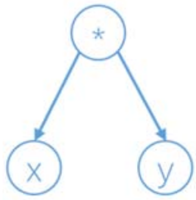
如样例中的 $xy * x * y * x * y *$，表达式树为：



合并后为：



我们发现，一个结点的叶子结点儿子可以并到一起，只计算一次，上图就变成了：



于是这启发我们，将相邻运算符结点合并，同时将每个结点的叶子子结点（即字母子结点）合并，然后就可以方便我们计算答案。

设 $ans[u]$ 表示结点 u 的子树总共要用的最少储存空间，$avl[u][i]$ 表示结点 u 在答案最小的前提下，整个子树的表达式能否以字母 i 开头。

对于一个叶子结点，$ans[i]=1$，$avl[i][x]=1$ 当且仅当这个叶子结点的字符是x。

对于非叶子结点，先递归处理其所有子结点，再考虑表达式顺序。

首先我们考虑直接将所有表达式连起来，那么有：

$$ans[u]=1+|X|+\sum_{v\in son(u)\wedge v\text{不为叶子}} ans[v]$$

其中 $|X|$ 表示的是u 的不同叶子节点的个数。

我们注意到所有非叶子结点的表达式一定以变量开头，以操作符结尾。因此将两个相邻的非叶子结点放在一起一定不能减小答案。唯一的方法是将某个叶子结点放在某个非叶子结点前，并且叶子结点的字母与非叶子结点相同，那么我们通过合并让答案减小 1。因为我们已经将相同叶子结点合并，所以每个结点最多只能参与一次这样的合并。

这就将问题转化为了二分图匹配的模型，若对于一对非叶子结点i 和叶子结点j，满足 $avl[u][v]=1$，则在两点连边，最大匹配数即为答案减少的数。

这样就可以得到 $ans[i]$。现在的问题就是如何得到$avl[u][i]$。

u的所有子结点中，显然叶子结点i 都可以放在开头。那么对于其他的字母 i，想要放在开头，就要在 u 的子结点中找到一个非叶子结点 v，满足 $avl[v][i]=1$，并且 v在二分图中删去后最大匹配数不减小。因为我们令v不和其他叶子结点合并，所以把 v 单独放在开头，可以使字母 i 为整个表达式的开头。

现在的问题就是，求删去一个点后最大匹配数是否减小，这是一个很经典的问题。对于没有匹配边的点 v，删去显然不会减小答案。否则我们就从 v 原先的匹配点出发寻找一条增广路，如果找到了，说明原先的匹配点可以不跟 v 匹配，并且最大匹配不减小。这样我们就不用每次都重新删点后跑最大匹配。

时间复杂度为 $O(26n^2)$，但实际上跑不满这个上界。

【参考程序】

```
//陈煜翔
#include <bits/stdc++.h>

typedefstd::vector<int> vi;
```

```
constintMaxN = 6e3 + 5;

int n;
char s[MaxN];
vi son[MaxN];
int fa[MaxN];
int top, stk[MaxN];
int tag[MaxN], tag_mark;
intans[MaxN], y[MaxN], ty[MaxN];
bool invalid[MaxN];
bool leaf[MaxN][26], avl[MaxN][26];

inline bool is_letter(char c)
{
    return c >= 'a' && c <= 'z';
}

inline bool match(int u, int from)
{
    for (auto v : son[from])
        if (tag[v] != tag_mark&&avl[v][u])
        {
            tag[v] = tag_mark;
            if (y[v] == -1 || match(y[v], from))
            {
                y[v] = u;
                return true;
            }
        }
    return false;
}

inline bool tmp_match(int u, int from)
{
    for (auto v : son[from])
        if (tag[v] != tag_mark&&avl[v][u])
        {
            tag[v] = tag_mark;
            if (y[v] == -1 || tmp_match(y[v], from))
                return true;
        }
    return false;
}
```

```
inline void dfs(int u)
{
    ans[u] = 1;
    for (auto v : son[u])
    {
        dfs(v);
        ans[u] += ans[v];
    }
    for (inti = 0; i< 26; ++i)
        ans[u] += leaf[u][i]; //先计算ans[u]的初值
    tag_mark = 0;
    for (inti = 0; i< 26; ++i)
        if (leaf[u][i])
        {
            ++tag_mark;
            avl[u][i] = true;
            ans[u] -= match(i, u); //将ans[u]减去最大匹配数
        }
    for (auto v : son[u])
    {
        tag[v] = ++tag_mark; //找v的匹配点的增广路时，先标记v不能走
        if (y[v] == -1 || tmp_match(y[v],u))//判断删去v后最大匹配数是否减小
        {
            for (inti = 0; i< 26; ++i)
                avl[u][i] |= avl[v][i];
        }
    }
}

intmain()
{
    freopen("expr.in", "r", stdin);
    freopen("expr.out", "w", stdout);
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for (inti = 1; i<= n; ++i)
    {
        y[i] = -1; //因为我们将字母a转化成序号0，所以这里初值设为-1
        if (is_letter(s[i]))
            stk[++top] = i;
        else
        {
            fa[stk[top]] = i;
            fa[stk[top - 1]] = i; //利用栈建立表达式树
```

```
            stk[--top] = i;
        }
    }
    for (inti = 1; i< n; ++i)
    {
        while (s[fa[i]] == s[fa[fa[i]]]) //合并相同的相邻操作符结点
        {
            invalid[fa[i]] = true; //并且标记原图中无用的点
            fa[i] = fa[fa[i]];
        }
    }
    for (inti = 1; i<= n; ++i)
    {
        if (is_letter(s[i]))
            leaf[fa[i]][s[i] - 'a'] = true; //标记每个点的叶子结点的集合
        else if (!invalid[i])
            son[fa[i]].push_back(i); //son[u]储存u的儿子集合
    }
    dfs(n);
    printf("%d\n", ans[n]);
    return 0;
}
```