折纸(paper,1s,512MB)

【算法分析】

当我们沿行对折时，每一行折在一起的部分一定是偶回文串，而偶回文串的两半对沿列对折的影响是相同的，所以沿行对折对于沿列对折是没有影响的。

同理，沿列对折对于沿行对折也是没有影响的，行和列相互独立，我们只需要分别对于行和列求出合法的区间数相乘即为最后的答案。

以对列求合法的区间数为例，我们发现一个区间合法的充要条件为：

1. 区间的左端点左边的部分一定存在一种方案能够折到右边去，并且最终折到右边的部分不会超过纸张原有的右边界。

2. 区间的右端点右边的部分一定存在一种方案能够折到左边去，并且最终折到左边的部分不会超过纸张原有的左边界。

该条件的必要性较为显然，考虑证明其充分性。

对于把左端点左边的部分折到右边去或者把右端点右边的部分折到左边去的那一次操作，因为折过去的部分没有超过纸张原有的边界，现在的纸张相当于只剩下了选取区间以及选取区间某一边的某一部分，因此折过去的过程可以不断递归下去，在递归中纸张的宽度不断减小，最后一定能够只剩下选取的区间。

由于合法区间的左右端点相互独立，我们只要求出所有合法的左右端点，用前缀和即可计算出合法的区间数。

考虑合法的左右端点所要满足的条件。先用 manacher 求出每个端点的回文半径，以左端点为例，一个左端点合法当且仅当它的回文半径能够覆盖纸张原有的左边界或回文半径能够覆盖到的范围的左半部分内存在合法的左端点，可以用前缀和快速判断。

时间复杂度 O(nm)。

【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;
typedef long longll;
constint N = 2e6 + 5;
constintMaxn = 0x3f3f3f3f;
vector<char> s[N], t[N]; llans;
int n, m, pre[N], suf[N], r[N], mx_r[N]; char a[N];

template <class T>
```

```
inline void CkMin(T &x, T y) {x > y ? x = y : 0;}
template <class T>
inline T Min(T x, T y) {return x < y ? x : y;}

inline llsolve(int n, int m)
{   //分别计算对行和对列合法的区间数
    for (int j = 1; j <= m; ++j)
        mx_r[j] = Maxn;
    for (inti = 1; i<= n; ++i)
    {   //manacher 求回文半径
        int mx = 0, pos = 0;
        for (int j = 1; j <= m; ++j)
        {
            r[j] = mx >j ?Min(r[(pos<< 1) - j], mx - j) : 1;
            while (s[i][j - r[j]] == s[i][j + r[j]])
                ++r[j];
            if (j + r[j] > mx)
                mx = j + r[j], pos = j;
            CkMin(mx_r[j], r[j]);
        }
    }

    pre[0] = suf[m + 1] = 0;
    for (int j = 1; j <= m; ++j)
    {
        pre[j] = pre[j - 1];
        if ((j & 1) && (j == mx_r[j] || pre[j - 1] - pre[j - mx_r[j]] >
0))
            ++pre[j];
    }
    for (int j = m; j >= 1; --j)
    {
        suf[j] = suf[j + 1];
        if ((j & 1) && (j + mx_r[j] > m || suf[j+1] - suf[j+mx_r[j]] > 0))
            ++suf[j];
    }
    ll res = 0;
    for (int j = m; j >= 1; j -= 2)
        pre[j] -= pre[j - 1];
    for (int j = 1; j + 2 <= m; j += 2)
        res += 1ll * pre[j] * suf[j + 2];
    return res;
}
```

```
intmain()
{
    freopen("paper.in", "r", stdin);
    freopen("paper.out", "w", stdout);

    read(n); read(m);
    for (inti = 1; i<= n; ++i)
        s[i].resize((m + 1 << 1) + 1);
    for (inti = 1; i<= m; ++i)
    {
        t[i].resize((n + 1 << 1) + 1);
        t[i][0] = '&';
    }
    for (inti = 1; i<= n; ++i)
    {
        scanf("%s", a + 1);
        s[i].resize((m + 1 << 1) + 1);

        s[i][0] = '&';
        for (int j = 1; j <= m; ++j)
        {
            s[i][j - 1 << 1 | 1] = '#';
            s[i][j << 1] = a[j];
            t[j][i - 1 << 1 | 1] = '#';
            t[j][i<< 1] = a[j];
        }
        s[i][m << 1 | 1] = '#';
        s[i][m + 1 << 1] = '$';
    }
    for (inti = 1; i<= m; ++i)
    {
        t[i][n << 1 | 1] = '#';
        t[i][n + 1 << 1] = '$';
    }

    ans = solve(n, m << 1 | 1);
    for (inti = 1; i<= m; ++i)
        s[i] = t[i];
    ans = ans * solve(m, n << 1 | 1);
    std::cout<<ans<<std::endl;

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

月宫的符卡序列（A,1s,512MB）

【算法分析】

记以i为中心的最长回文子串的半径为f_i（即以i为中心的最长回文子串的长度为$2\times f_i+1$）。在manacher算法中，只有f_i增大时，才可能出现新的回文子串。因此S中本质不同的回文串只有$O(n)$个。

不妨将所有本质不同的回文子串看作节点，可以在它们之间发现树结构：如果子串$S_{l,r}$是回文串且$r-l+1\geq 3$，那么$S_{l+1,r-1}$必定也是回文串。因此可将$S_{l+1,r-1}$看作$S_{l,r}$的父节点，它们的连边上的字符为S_l（$S_l=S_r$）。特殊地，令树根为空串（1号节点）。若$S_{l,r}$是回文串且$r-l+1\leq 2$，则它的父节点为树根。这样所有本质不同的回文子串就组成了一棵树。

考虑用manacher算法建出这棵树。记当前右端点最大的回文子串中心为k，半径为f_k，记pos_i表示$S_{i-f_i+1,i+f_i-1}$对应树节点的编号，$ch_{u,c}$表示点u走字符c到达的节点。分情况讨论：

1. $k+f_k-1\geq i$：令$j=2k-i$，即i以k为中心的对称点。当$i+f_j\leq k+f_k$时，令$pos_i=pos_j, f_i=f_j$。否则令$f_i=k+f_k-i$，$pos_i=pos_j$的f_j-f_i级祖先。可以用倍增求这个祖先的编号。
2. $k+f_k-1<i$：令$f_i=1$，判断ch_{1,S_i}是否存在，如果不存在则新建节点。接着令$pos_i=ch_{1,S_i}$。

接下来让f_i继续扩展：当$S_{i-f_i}=S_{i+f_i}$时，同样判断$ch_{pos_i,S_{i+f_i}}$是否存在，如果不存在则新建节点，然后令$pos_i=ch_{pos_i,S_{i+f_i}}$，$f_i+1$。

这样我们就把树建出来了，考虑如何计算答案：对于每个i，若$S_{i-f_i+1,i+f_i-1}$对应的节点为x，则将点x的权值异或上i。最后把整棵树dfs一遍，将每个点的权值变为子树的权值异或和。那么现在每个回文子串对应节点的权值就是它的价值了。

时间复杂度$O(n\log n)$，空间复杂度$O(n)$。

【参考程序】 //陈予菲

```
#include <bits/stdc++.h>

usingnamespace std;

constint e = 2e6 + 15;
int n, f[e], g[e], fa[e][20], tot, b[e], ch[e][27], m, pos[e], ans;
char s[e], a[e];

inlineint insert(int u, char s)
{
    int c = s - 'a';
    if (!isalpha(s)) c = 26;
    if (ch[u][c]) returnch[u][c];
    ch[u][c] = ++tot;            //不存在 ch[u][c] 则新建节点
    fa[tot][0] = u;
```

```
    u = ch[u][c];
    for (int i = 0; i <19; i++) fa[u][i + 1] = fa[fa[u][i]][i];
    return u;
}

inlineint jump(int x, int d)         //倍增求 x 的 d 级祖先
{
    for (int i = 19; i >= 0; i--)
    if(d & (1<< i)) x = fa[x][i];
    return x;
}

inlinevoid manacher()
{
    int i, k = 0;
    for (i = 2; i < m; i++)
    {
        if (k + f[k] - 1>= i)    //情况 1
        {
            int j = (k <<1) - i;
            if (i + f[j] <= k + f[k]) pos[i] = pos[j], f[i] = f[j];
            else f[i] = k+f[k]-i, pos[i] = jump(pos[j], f[j]-f[i]);
        }
        else f[i] = 1, pos[i] = insert(1, a[i]); //情况 2
        while(a[i - f[i]] == a[i + f[i]])       //f[i]继续扩展
        {
            f[i]++;
            pos[i] = insert(pos[i], a[i + f[i] - 1]);
        }
        g[pos[i]] ^= b[i]; //i 对应原串的位置 b[i]，将 pos[i]的权值异或上 i
        if (i + f[i] > k + f[k]) k = i;
    }
}

inlinevoid solve()
{
    scanf("%s", s + 1);
      n = strlen(s + 1); m = 0; tot = 1;
      memset(b, 0, sizeof(b));
    memset(f, 0, sizeof(f));
      memset(g, 0, sizeof(g));
      memset(fa, 0, sizeof(fa));
      int i; ans = 0;
      memset(pos, 0, sizeof(pos));
```

```
      memset(ch, 0, sizeof(ch));
    //多组数据
    a[++m] = '.'; a[++m] = '*';
    for (i = 1; i <= n; i++)
    {
        a[++m] = s[i]; b[m] = i - 1;
        a[++m] = '*'; b[m] = i - 1;
    }
    a[++m] = '!';
    manacher();
    for (i = tot; i >= 2; i--)  //将每个点的权值变为子树的权值异或和
    if (i != ch[1][26])
    {
        ans = max(ans, g[i]);
        g[fa[i][0]] ^= g[i];
    }
    cout << ans << endl;
}

int main()
{
    freopen("A.in", "r", stdin);
    freopen("A.out", "w", stdout);
    int T;
    cin >> T;
    while (T--) solve();
    fclose(stdin);
    fclose(stdout);
    return0;
}
```

回文子串(palindrome,4s,256MB)

【算法分析】

看到回文串，可以想到 Manacher，看到区间修改和询问，又可以想到线段树。

考虑线段树上区间$[l,r]$对应的节点维护以下信息：

（1） st：子串$[l,r]$的前$\min(r-l+1,K)$个字符组成的字符串。

（2） ed：子串$[l,r]$的后$\min(r-l+1,K)$个字符组成的字符串。

（3） res：子串$[l,r]$包含多少个长度不超过K的回文子串。

现在要解决的有两个问题：标记的影响与信息合并。

标记的影响比较好处理：一个区间$[l,r]$打上替换为字符c的标记之后，st和ed都为ccccc…c（$\min(r-l+1,K)$个c），且$[l,r]$的所有子串都是回文串，res也可以计算出来。

对于信息合并，设要合并$[l,mid]$和$[mid+1,r]$的信息，对于res$[l,r]$，我们可以把左右子

区间的答案加起来之后，再加上跨越中间的回文子串个数。

考虑如何计算同时包含mid和$mid+1$的回文子串个数：这样的回文子串必然是$s'=ed[l,mid]+st[mid+1,r]$的子串。于是我们可以对s'进行一遍 Manacher 之后，枚举回文中心统计即可，注意回文串必须跨越mid和$mid+1$并且长度不能超过K。

$st[l,r]$即为$st[l,mid]+st[mid+1,r]$的前$\min(r-l+1,K)$个字符。$ed[l,r]$同理。

$O((|S|+Q)K\log|S|)$。

【参考程序】

```
//陈栉旷
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#define p2 p << 1
#define p3 p << 1 | 1

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()) < 'a' || c > 'z');
    return c;
}

template <class T>
inline T Min(const T &a, const T &b) {return a < b ? a : b;}

template <class T>
inline T Max(const T &a, const T &b) {return a > b ? a : b;}

const int N = 5e4 + 5, M = N << 2, E = 55, W = E << 1, Z = W << 1;

char s[N], tag[M], t[W], qt[Z];
int n, k, q, R[Z];
```

```
struct node
{
    char st[E], ed[E];
    intlen, res;

    friend inline node operator + (node a, node b) // 合并信息
    {
        int m = Min(k, a.len) + Min(k, b.len), n = 0, pos = 0, id = 0;
        for (inti = 1; i<= Min(k, a.len); i++)
            t[Min(k, a.len) - i + 1] = a.ed[i];
        for (inti = 1; i<= Min(k, b.len); i++)
            t[Min(k, a.len) + i] = b.st[i];
        qt[0] = '!';
        for (inti = 1; i<= m; i++)
            qt[++n] = '%', qt[++n] = t[i];
        qt[++n] = '%'; qt[n + 1] = '@';
        for (inti = 1; i<= n; i++)
        {
            R[i] = pos>i ?Min(pos - i, R[(id << 1) - i]) : 1;
            while (qt[i - R[i]] == qt[i + R[i]]) R[i]++;
            if (i + R[i] >pos) pos = i + R[i], id = i;
        } //把左子区间的 ed 和右子区间的 st 连接起来做 Manacher
        if (k & 1)
        {
            for (inti = 1; i<= n; i += 2)
                R[i] = Min(R[i], k);
            for (inti = 2; i<= n; i += 2)
                R[i] = Min(R[i], k + 1);
        }
        else
        {
            for (inti = 1; i<= n; i += 2)
                R[i] = Min(R[i], k + 1);
            for (inti = 2; i<= n; i += 2)
                R[i] = Min(R[i], k);
        }  //处理掉长度超过 K 的回文子串
        node res; res.len = a.len + b.len;
        if (a.len>= k) for (inti = 1; i<= k; i++)
            res.st[i] = a.st[i];
        else
        {
            for (inti = 1; i<= a.len; i++)
                res.st[i] = a.st[i];
            for (inti = 1; i<= Min(k - a.len, b.len); i++)
```

```
                res.st[a.len + i] = b.ed[Min(k, b.len) - i + 1];
        }
        if (b.len>= k) for (inti = 1; i<= k; i++)
            res.ed[i] = b.ed[i];
        else
        {
            for (inti = 1; i<= b.len; i++)
                res.ed[i] = b.ed[i];
            for (inti = 1; i<= Min(k - b.len, a.len); i++)
                res.ed[b.len + i] = a.st[Min(k, a.len) - i + 1];
        }  //计算 st 和 ed
        res.res = a.res + b.res + (R[(Min(k, a.len) << 1) + 1] >> 1);
        for (inti = 1; i<= (Min(k, a.len) << 1); i++)
            res.res += Max(0, i + R[i] - 2 - (Min(k, a.len) << 1) >> 1);
        for (inti = (Min(k, a.len) + Min(k, b.len) << 1) + 1;
            i>= (Min(k, a.len) + 1 << 1); i--)
                res.res += Max(0, (Min(k, a.len) << 1) - i + R[i] >> 1);
        return res;
    }
} T[M];

node orz_dyf(int l, int r, char c)  //处理标记的影响
{
    node res; res.len = r - l + 1;
    for (inti = 1; i<= Min(k, r - l + 1); i++)
        res.st[i] = res.ed[i] = c;
    res.res = 0;
    for (inti = 1; i<= Min(k, r - l + 1); i++)
        res.res += r - l + 2 - i;
    return res;
}

void build(int l, int r, int p)      //建树
{
    if (l == r)
    {
        T[p].len = T[p].res = 1;
        T[p].st[1] = T[p].ed[1] = s[l];
        return;
    }
    int mid = l + r >> 1;
    build(l, mid, p2); build(mid + 1, r, p3);
    T[p] = T[p2] + T[p3];
}
```

```
void down(int p)
{
    if (tag[p]) tag[p2] = tag[p], tag[p3] = tag[p], tag[p] = 0;
}

void upt(int l, int r, int p)
{
    int mid = l + r >> 1;
    T[p] = (tag[p2] ?orz_dyf(l, mid, tag[p2]) : T[p2])
        + (tag[p3] ?orz_dyf(mid + 1, r, tag[p3]) : T[p3]);
}

void change(int l, int r, int s, int e, char c, int p) //修改
{
    if (l == s && r == e) return (void) (tag[p] = c);
    int mid = l + r >> 1; down(p);
    if (e <= mid) change(l, mid, s, e, c, p2);
    else if (s >= mid + 1) change(mid + 1, r, s, e, c, p3);
    else change(l, mid, s, mid, c, p2),
        change(mid + 1, r, mid + 1, e, c, p3);
    upt(l, r, p);
}

node ask(int l, int r, int s, int e, int p) //查询
{
    if (l == s && r == e) return tag[p] ?orz_dyf(l, r, tag[p]) : T[p];
    int mid = l + r >> 1; node res; down(p);
    if (e <= mid) res = ask(l, mid, s, e, p2);
    else if (s >= mid + 1) res = ask(mid + 1, r, s, e, p3);
    else res = ask(l, mid, s, mid, p2)
        + ask(mid + 1, r, mid + 1, e, p3);
    return upt(l, r, p), res;
}

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

    int op, l, r; char c;
    scanf("%s", s + 1);
    n = strlen(s + 1);
    k = read(); q = read();
```

```
    build(1, n, 1);
    while (q--)
    {
        op = read(); l = read(); r = read();
        if (op == 1)
        {
            c = get();
            change(1, n, l, r, c, 1);
        }
        else printf("%d\n", ask(1, n, l, r, 1).res);
    }
    return 0;
}
```

简单的字符串（naive，2s，512MB）

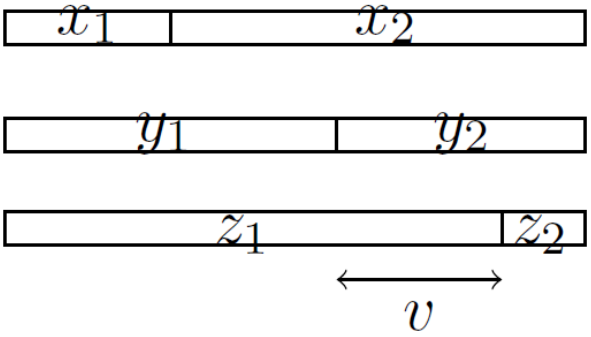
【算法分析】

问题即为求有多少个子区间可以表示成$uvvu$（$|u|\neq 0$，$|v|$可以为0）的形式。

我们发现，如果以两个v作为中心，从中心分别向两边扩展，从这个过程可以得到两个字符串$a=v^Ru^R,b=vu$（u^R表示u的反转），那么$s=a_1b_1a_2b_2\dots a_nb_n$这个串相当于是$v$和$v^R$交错拼起来，$u$和$u^R$交错拼起来，然后再把这两个得到的字符串连起来。那么s这个串相当于两个**偶回文串**拼接而成。

考虑枚举中心，然后得到向两边扩展的字符串a,b，两个字符串都只保留前缀$1\dots\min\{|a|,|b|\}$，然后根据上面的方式得到字符串s。那么我们就是需要统计，s有多少个前缀，能够表示成两个**偶回文串**拼接而成的结果（注意特判只有$|v|=0$）。

Lemma 1：对于一个**双回文串**s（能够表示成两个非空回文串拼接的结果），若$s=x_1x_2=y_1y_2=z_1z_2(|x_1|<|y_1|<|z_1|)$且$x_2,y_1,y_2,z_1$是回文串，则$x_1,z_2$也是回文串。



证明：下面只证x_1是回文串，z_2同理。

如上图，设$z_1=y_1v$，则v是y_2的前缀，v^R是x_2,y_2的后缀，v是x_2的前缀，于是x_1v是z_1的前缀。y_1是z_1的 border，所以$|v|$是z_1的 period，于是$|v|$也是x_1v的 period。所以x_1是v^∞的后缀。v^R是x_1,z_1的前缀，而$|v|$是x_1的 period，所以x_1是$(v^R)^\infty$的前缀。

故x_1是回文串。▫

Lemma 2：对于一个双回文串s，存在一种回文划分$s=ab$（a,b均为回文串且非空），

使得a是s的最长回文前缀，或b是s的最长回文后缀。

可以根据 Lemma 1 加上一些分类得到，这里不详细证明。

Lemma 3：对于一个**双偶回文串**s（能表示成两个偶回文串拼接的结果），存在一种回文划分$s = ab$（a, b均为**偶回文串**且非空），使得a是s的最长**偶**回文前缀，或b是s的最长**偶**回文后缀。

将 Lemma 1 和 Lemma 2 的「回文串」改成「偶回文串」结论仍然成立。

因此我们只需要求出所有前缀的最长偶回文前缀和最长偶回文后缀，并分别判断剩下的部分是否回文。

求所有前缀的最长偶回文后缀，可以从左往右扫，维护一个当前的回文串能延伸右边界rit。每次枚举到新的回文中心i，只需要更新左端点在区间$(rit, i + r_i)$内的信息即可（r_i表示i的回文半径）。正确性显然。（注意因为我们只考虑偶回文串，所以我们只枚举以特殊字符#（Manacher 时插入的特殊字符）为回文中心的贡献）

判断剩下的部分是否回文就直接用中心的回文半径判即可。最长偶回文前缀就枚举的时候顺便维护一下即可。

【参考程序】

```
//陈煜翔
#include <bits/stdc++.h>

template <class T>
inline void read(T &x)
{
    static char ch;
    while (!isdigit(ch = getchar()));
    x = ch - '0';
    while (isdigit(ch = getchar()))
        x = x * 10 + ch - '0';
}

template <class T>
inline void relax(T &x, const T &y)
{
    if (x < y)
        x = y;
}

template <class T>
inline void tense(T &x, const T &y)
{
    if (x > y)
        x = y;
}

template <class T>
```

```
inline T get_abs(const T &x)
{
    return x <0 ? ~x + 1 : x;
}

constintMaxN = 1e4 + 5;

int n, m, ans;
int a[MaxN], s[MaxN];

inline void solve(int *s, int n)
{
    static int t[MaxN], r[MaxN], m;
    static intmax_suf[MaxN];
    m = 0;

    for (inti = 1; i<= n; ++i)
    {
        t[(i<< 1) - 1] = 0;
        t[i<< 1] = s[i];  //避免奇偶讨论，在字符之间加入 0
    }

    m = n << 1 | 1, t[m] = 0;
    t[0] = -1, t[m + 1] = -2;

    intrit = 0, p = 0;
    for (inti = 1; i<= m; ++i) //Manacher 求出每个位置的回文半径

    {
        if (rit>i)
        {
            int j = (p << 1) - i;
            r[i] = std::min(r[j], rit - i);
        }
        else
            r[i] = 1;
        while (t[i - r[i]] == t[i + r[i]])
            ++r[i];
        if (i + r[i] >rit)
        {
            rit = i + r[i];
            p = i;
        }
        max_suf[i] = 0;
```

```
    }

    rit = 1;
    for (inti = 1; i<= m; i += 2)
    {
        for (int j = i + r[i] - 1; j >= rit&& j >i; --j)
            relax(max_suf[j], (j - i)<< 1 | 1);  //更新 j 的最长回文后缀长度
        relax(rit, i + r[i]);
    }

    intcur_pre = 0;
    for (inti = 2; i<= n; i += 2)
    {
        bool flg = false;

        if (r[i + 1] >i)    //如果 1..i 本身就是偶回文串
        {
            flg = true;      //则直接算作合法
            cur_pre = i + 1; //并且 1..i 是当前的最长偶回文前缀
        }
        if (max_suf[i<< 1])  //将最长偶回文后缀去掉后，判断剩下的部分是否回文
        {
            int cur = i - (max_suf[i<< 1] >> 1) - 1;
            if (r[cur + 1] > cur)
                flg = true;
        }
        if (cur_pre&&r[cur_pre + i] >i - cur_pre)
            flg = true; //将最长偶回文前缀去掉后，判断剩下的部分是否回文
        ans += flg;
    }
}

intmain()
{
    freopen("naive.in", "r", stdin);
    freopen("naive.out", "w", stdout);

    read(n);
    for (inti = 1; i<= n; ++i)
        read(a[i]);
    for (inti = 1; i< n; ++i)
    {
        m = 0;
        int l = i, r = i + 1;
```

```
        while (l >= 1 && r <= n)
        {
            s[++m] = a[l];
            s[++m] = a[r]; //枚举中心，再向两边扩展得到字符串
            --l, ++r;
        }

        solve(s, m);
    }
    std::cout<<ans<<std::endl;

    return 0;
}
```