

无源汇可行费用流

无源汇可行流

有源汇最大流

有源汇点最小流

数据结构

势能线段树:

动态开点

网络流

二分图

8.2

P1129

P1129

费用流笔记

拆式子拆点

P2053 *

前言:

思路:

做法:

P3159 *

此题算法:费用流

输出方案

数据结构优化建图

一流对区间流问题

限制单点最大值问题

限制单点最小值问题

动态加点问题

费用流最小路径覆盖

P2469

最大权完美匹配

费用流二分

Dilworth定理

限制性匹配问题

JSOI2009 球队收益 / 球队预算

题目描述

P4249 WC2007 剪刀石头布

题目描述

ZJOI2011 营救皮卡丘

洞穴遇险

分层图思想

长度限制性条件

最长不下降子序列

小规模建图思想

对偶图思想

网络流的退流

P3308 *

一选二问题

P1231

P2891

最小化最小割和割边数量

二分答案 网络流

并行思维

最小不相交路径覆盖问题

P2764

题目描述

P2765
P2172
DAG的最小可相交路径覆盖 杂
P5769

二分图多重匹配问题

P3254 P2763

二分图最大独立集

多源点多汇点网络流的特殊情况 ———— 二源点二汇点，指定流向

P3163 *

二分图最大点覆盖集

8.3

二分图最小割

P1361 P2774 虚点

P4174

P2057

P2774

生成树换边

P5039 *

P5934 *

带限制条件的最小割 ———— 回流边的研究

P5039

P6054

最小割树

P4123 P4897 P3329

最小割最大闭合子图模型

P2762 P4174

P2057

拓扑去环

P2805

虚点

P1361 P2774

P1935 对于对立情况进行黑白染色

不好归类，但相当经典

P4177 ?

问题集锦

传递性失效

最小割相同权值产生多割点假象

Hall 定理

1. P3488
2. CF981F Round Marriage
3. Loj6062. 「2017 山东一轮集训 Day2」 Pair
4. CF1009G Allowed Letters
5. [ARC076F] Exhausted?

自动机

1. AC 自动机 ACAM

1.1 算法详解

1.2 fail 树的性质与应用

1.3 应用

1.3.1 结合动态规划

1.3.2 结合矩阵快速幂

1.4 注意点

1.5 例题

I. P3808 【模板】AC 自动机（简单版）

II. [P2292 HNOI2004] L 语言

*III. [P2414 NOI2011] 阿狸的打字机

IV. P5357 【模板】AC 自动机（二次加强版）

- *V. [P4052 JSOI2007]文本生成器
- VI. [P3041 USACO12JAN]Video Game G
- *VII. CF1202E You Are Given Some Strings...
- VIII. CF163E e-Government
- *IX. [P7456 CERC2018] The ABCD Murderer
- X. [P3121 USACO15FEB]Censoring G
- XI. [P3715 BJOI2017]魔法咒语
- XII. CF696D Legen...
- *XIII. [P5840 COCI2015]Divljak
- 2. 后缀自动机 SAM
 - 2.1 基本定义与引理
 - 2.2 关键结论
 - 2.3 构建 SAM
 - 2.4 时间复杂度证明
 - 2.4.1 状态数上界
 - 2.4.2 转移数上界
 - 2.4.3 操作次数上界
 - 2.5 应用
 - 2.5.1 求本质不同子串个数
 - 2.5.2 字符串匹配
 - 2.6 广义 SAM
 - 2.7 常用技巧与结论
 - 2.7.1 线段树合并维护 endpos 集合
 - 2.7.2 桶排确定 dfs 顺序
 - 2.7.3 快速定位子串
 - 2.7.4 其它结论
 - 2.8 注意点总结
 - 2.9 例题
 - I. P3804 【模板】后缀自动机 (SAM)
 - II. [P4070 SDOI2016]生成魔咒
 - *III. [P4022 CTSC2012]熟悉的文章
 - IV. [P5546 POI2000]公共串
 - V. [P3346 ZJOI2015]诸神眷顾的幻想乡
 - VI. [P3181 HAOI2016]找相同字符
 - VII. [P5341 TJOI2019]甲苯先生和大中锋的字符串
 - VIII. [P4341 BJWC2010]外星联络
 - *IX. [P3975 TJOI2015]弦论
 - *X. H1079 退群杯 3rd E.
 - XI. CF316G3 Good Substrings
 - XII. SP8222 NSUBSTR - Substrings
 - XIII. 某模拟赛 一切的开始
 - *XIV. CF1037H Security
 - *XV. CF700E Cool Slogans
 - *XVI. CF666E Forensic Examination
 - 2.10 相关链接与资料
- 3. 回文自动机 PAM

无源汇可行费用流

```
pair<bool, i64> MCFP(vector<array<int, 5>> &e, int n) {  
    int N = n + 2;  
    int s = N - 2, t = s + 1;  
    vector<int> d(n);
```

```

MCFGGraph g(N);
for (auto [u, v, L, U, c] : e) {
    g.addEdge(u, v, U - L, c);
    d[u] -= L;
    d[v] += L;
}
for (int i = 0; i < n; i += 1) {
    if (d[i] > 0) {
        g.addEdge(s, i, d[i], 0);
    } else {
        g.addEdge(i, t, -d[i], 0);
    }
}
auto [flow, cost] = g.flow(s, t);
bool ok = 1;
for (int i = g.r[s]; ~i; i = g.t[i]) {
    ok &= g.e[i].c == 0;
}
for (int i = g.r[t]; ~i; i = g.t[i]) {
    ok &= g.e[i ^ 1].c == 0;
}
return {ok, cost};
}

```

无源汇可行流

[oiwiki](#)

[知乎](#)

[板子题](#)

示例代码

有缘汇点汇点向源点连一条下界为0，上界为无穷大的边即可

```

# include <bits/stdc++.h>
using namespace std;

# ifdef LOCAL
    # include "C:\Users\Kevin\Desktop\demo\save\debug.h"
# else
# define debug(...) 114514
# define ps 114514
# endif

using i64 = long long;

using i128 = __int128_t;

template<class T>
struct MaxFlow {
    int n;
    vector<int> r, t, to, h, cur;
    vector<T> c;
    MaxFlow(int n, int m = 0) {

```

```

        init(n, m);
    }
    void init(int n, int m = 0) {
        this->n = n;
        r.assign(n, -1);
        h.assign(n, -1);
        cur.assign(n, 0);
        t.reserve(2 * m);
        to.reserve(2 * m);
        c.reserve(2 * m);
    }
    void addEdge(int u, int v, T cap) {
        t.push_back(r[u]), to.push_back(v), c.push_back(cap), r[u] = t.size() -
1;
        t.push_back(r[v]), to.push_back(u), c.push_back(0), r[v] = t.size() - 1;
    }
    bool bfs(int s, int e) {
        fill(h.begin(), h.end(), -1);
        queue<int> q;
        h[s] = 0;
        cur[s] = r[s];
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i = r[u]; ~i; i = t[i]) {
                int v = to[i];
                T cap = c[i];
                if (cap > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    cur[v] = r[v];
                    if (v == e) {
                        return true;
                    }
                    q.push(v);
                }
            }
        }
        return false;
    }
    T dfs(int u, int e, T f) {
        if (u == e) {
            return f;
        }
        T r = f;
        for (int &i = cur[u]; ~i; i = t[i]) {
            int v = to[i];
            T cap = c[i];
            if (cap > 0 && h[v] == h[u] + 1) {
                T k = dfs(v, e, min(cap, r));
                if (k == 0) {
                    h[v] = -1;
                }
                c[i] -= k;
                c[i ^ 1] += k;
                r -= k;
            }
        }
    }

```

```

        if (r == 0) {
            return f;
        }
    }
}

return f - r;
}

T flow(int s, int e) {
    T ans = 0;
    while (bfs(s, e)) {
        ans += dfs(s, e, std::numeric_limits<T>::max());
    }
    return ans;
}

std::vector<bool> minCut() {
    std::vector<bool> c(n);
    for (int i = 0; i < n; i++) {
        c[i] = (h[i] != -1);
    }
    return c;
}

struct Edge {
    int from;
    int to;
    T cap;
    T flow;
    friend ostream &operator<<(ostream &cout, Edge u) {
        return cout << '{' << u.from << ", " << u.to << ", " << u.cap << ", "
<< u.flow << "}";
    }
};

vector<Edge> edges() {
    vector<Edge> a;
    for (int i = 0; i < t.size(); i += 2) {
        Edge x;
        x.from = to[i + 1];
        x.to = to[i];
        x.cap = c[i] + c[i + 1];
        x.flow = c[i + 1];
        a.push_back(x);
    }
    return a;
}

};

void solve () {
    int n, m;
    cin >> n >> m;
    MaxFlow<int> g(n + 2);
    int s = n, t = s + 1;
    vector<int> d(n);
    vector<int> ans(m);
    for (int i = 0; i < m; i += 1) {
        int u, v, L, U;

```

```

        cin >> u >> v >> L >> U;
        -- u, -- v;
        g.addEdge(u, v, U - L);
        d[u] -= L;
        d[v] += L;
        ans[i] = L;
    }
    for (int i = 0; i < n; i += 1) {
        if (d[i] > 0) {
            g.addEdge(s, i, d[i]);
        } else {
            g.addEdge(i, t, -d[i]);
        }
    }
    g.flow(s, t);
    bool ok = 1;
    auto adj = g.edges();
    for (auto [u, v, c, f] : adj) {
        if (u == s) {
            ok &= (c == f);
        } else if (v == t) {
            ok &= (c == f);
        }
    }
    cout << (ok ? "YES" : "NO") << '\n';
    if (ok) {
        for (int i = 0; i < m; i += 1) {
            ans[i] += adj[i].flow;
        }
        for (auto u : ans) {
            cout << u << '\n';
        }
    }
}

// 修一下爆没爆int
// 多测

signed main () {
    # ifndef cin
        ios::sync_with_stdio(false);
        cin.tie(nullptr);
    # endif
    i64 _ = 1;
    // cin >> _;
    while (_ --) {
        solve ();
    }
    return 0;
}

```

有源汇最大流

模板题

```
# include <bits/stdc++.h>
using namespace std;

#ifdef LOCAL
    # include "C:\Users\Kevin\Desktop\demo\save\debug.h"
#else
    # define debug(...) 114514
    # define ps 114514
#endif

using i64 = long long;

using i128 = __int128_t;

template<class T>
struct MaxFlow {
    int n;
    vector<int> r, t, to, h, cur;
    vector<T> c;
    MaxFlow(int n, int m = 0) {
        init(n, m);
    }
    void init(int n, int m = 0) {
        this->n = n;
        r.assign(n, -1);
        h.assign(n, -1);
        cur.assign(n, 0);
        t.reserve(2 * m);
        to.reserve(2 * m);
        c.reserve(2 * m);
    }
    void addEdge(int u, int v, T cap) {
        t.push_back(r[u]), to.push_back(v), c.push_back(cap), r[u] = t.size() - 1;
        t.push_back(r[v]), to.push_back(u), c.push_back(0), r[v] = t.size() - 1;
    }
    bool bfs(int s, int e) {
        fill(h.begin(), h.end(), -1);
        queue<int> q;
        h[s] = 0;
        cur[s] = r[s];
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i = r[u]; ~i; i = t[i]) {
                int v = to[i];
                T cap = c[i];
                if (cap > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    cur[v] = r[v];
                }
            }
        }
    }
};
```



```

        if (v == e) {
            return true;
        }
        q.push(v);
    }
}

return false;
}

T dfs(int u, int e, T f) {
    if (u == e) {
        return f;
    }
    T r = f;
    for (int &i = cur[u]; ~i; i = t[i]) {
        int v = to[i];
        T cap = c[i];
        if (cap > 0 && h[v] == h[u] + 1) {
            T k = dfs(v, e, min(cap, r));
            if (k == 0) {
                h[v] = -1;
            }
            c[i] -= k;
            c[i ^ 1] += k;
            r -= k;
            if (r == 0) {
                return f;
            }
        }
    }
    return f - r;
}

T flow(int s, int e) {
    T ans = 0;
    while (bfs(s, e)) {
        ans += dfs(s, e, std::numeric_limits<T>::max());
    }
    return ans;
}

std::vector<bool> minCut() {
    std::vector<bool> c(n);
    for (int i = 0; i < n; i++) {
        c[i] = (h[i] != -1);
    }
    return c;
}

struct Edge {
    int from;
    int to;
    T cap;
    T flow;

    friend ostream &operator<<(ostream &cout, Edge u) {
        return cout << '{' << u.from << ", " << u.to << ", " << u.cap << ", "
        << u.flow << "}";
    }
};

```

```

    }
};

vector<Edge> edges() {
    vector<Edge> a;
    for (int i = 0; i < t.size(); i += 2) {
        Edge x;
        x.from = to[i + 1];
        x.to = to[i];
        x.cap = c[i] + c[i + 1];
        x.flow = c[i + 1];
        a.push_back(x);
    }
    return a;
}

};

void solve () {
    int n, m, s, t;
    cin >> n >> m >> s >> t;
    MaxFlow<i64> g(n + 2);
    -- s, -- t;
    int f = n, e = f + 1;
    vector<i64> d(n);
    for (int i = 0; i < m; i += 1) {
        int u, v, L, U;
        cin >> u >> v >> L >> U;
        -- u, -- v;
        d[u] -= L;
        d[v] += L;
        g.addEdge(u, v, U - L);
    }
    i64 tot = 0;
    for (int i = 0; i < n; i += 1) {
        if (d[i] > 0) {
            g.addEdge(f, i, d[i]);
            tot += d[i];
        } else {
            g.addEdge(i, e, -d[i]);
        }
    }

    // 在加入最后的边之前添加辅助边
    g.addEdge(t, s, 1e9);
    i64 k = g.flow(f, e);
    if (tot != k) {
        cout << "please go home to sleep" << '\n';
    } else {
        tot = g.c.end()[-1]; // 计算可行流的大小
        g.c.end()[-1] = g.c.end()[-2] = 0; // 清空返回边
        cout << tot + g.flow(s, t) << '\n';
    }
}

// 修一下爆没爆int
// 多测

signed main () {

```

```

# ifndef cin
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
# endif
    i64 _ = 1;
    // cin >> _;
    while (_ --) {
        solve ();
    }
    return 0;
}

```

有源汇点最小流

[模板题](#)

```

# include <bits/stdc++.h>
using namespace std;

# ifdef LOCAL
    # include "C:\Users\Kevin\Desktop\demo\save\debug.h"
# else
# define debug(...) 114514
# define ps 114514
# endif

using i64 = long long;

using i128 = __int128_t;

template<class T>
struct MaxFlow {
    int n;
    vector<int> r, t, to, h, cur;
    vector<T> c;
    MaxFlow(int n, int m = 0) {
        init(n, m);
    }
    void init(int n, int m = 0) {
        this->n = n;
        r.assign(n, -1);
        h.assign(n, -1);
        cur.assign(n, 0);
        t.reserve(2 * m);
        to.reserve(2 * m);
        c.reserve(2 * m);
    }
    void addEdge(int u, int v, T cap) {
        t.push_back(r[u]), to.push_back(v), c.push_back(cap), r[u] = t.size() - 1;
        t.push_back(r[v]), to.push_back(u), c.push_back(0), r[v] = t.size() - 1;
    }
    bool bfs(int s, int e) {
        fill(h.begin(), h.end(), -1);
    }

```

```

queue<int> q;
h[s] = 0;
cur[s] = r[s];
q.push(s);
while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = r[u]; ~i; i = t[i]) {
        int v = to[i];
        T cap = c[i];
        if (cap > 0 && h[v] == -1) {
            h[v] = h[u] + 1;
            cur[v] = r[v];
            if (v == e) {
                return true;
            }
            q.push(v);
        }
    }
}
return false;
}

T dfs(int u, int e, T f) {
    if (u == e) {
        return f;
    }
    T r = f;
    for (int &i = cur[u]; ~i; i = t[i]) {
        int v = to[i];
        T cap = c[i];
        if (cap > 0 && h[v] == h[u] + 1) {
            T k = dfs(v, e, min(cap, r));
            if (k == 0) {
                h[v] = -1;
            }
            c[i] -= k;
            c[i ^ 1] += k;
            r -= k;
            if (r == 0) {
                return f;
            }
        }
    }
    return f - r;
}

T flow(int s, int e) {
    T ans = 0;
    while (bfs(s, e)) {
        ans += dfs(s, e, std::numeric_limits<T>::max());
    }
    return ans;
}

std::vector<bool> minCut() {
    std::vector<bool> c(n);
    for (int i = 0; i < n; i++) {

```

```

        c[i] = (h[i] != -1);
    }
    return c;
}

struct Edge {
    int from;
    int to;
    T cap;
    T flow;
    friend ostream &operator<<(ostream &cout, Edge u) {
        return cout << '{' << u.from << ", " << u.to << ", " << u.cap << ", "
<< u.flow << "}";
    }
};

vector<Edge> edges() {
    vector<Edge> a;
    for (int i = 0; i < t.size(); i += 2) {
        Edge x;
        x.from = to[i + 1];
        x.to = to[i];
        x.cap = c[i] + c[i + 1];
        x.flow = c[i + 1];
        a.push_back(x);
    }
    return a;
}

};

void solve () {
    int n, m, s, t;
    cin >> n >> m >> s >> t;
    vector<i64> d(n);
    MaxFlow<i64> g(n + 2);
    -- s, -- t;
    int f = n, e = f + 1;
    for (int i = 0; i < m; i += 1) {
        int u, v, L, U;
        cin >> u >> v >> L >> U;
        -- u, -- v;
        d[u] -= L;
        d[v] += L;
        g.addEdge(u, v, U - L);
    }
    i64 tot = 0;
    for (int i = 0; i < n; i += 1) {
        if (d[i] > 0) {
            tot += d[i];
            g.addEdge(f, i, d[i]);
        } else {
            g.addEdge(i, e, -d[i]);
        }
    }
    g.addEdge(t, s, 1e9);
    if (tot != g.flow(f, e)) {
        cout << "please go home to sleep" << '\n';
    }
}

```

```

    } else {
        tot = g.c.back();
        g.c.end()[-1] = g.c.end()[-2] = 0;
        cout << tot - g.flow(t, s) << '\n';
    }
}

// 修一下爆没爆int
// 多测

signed main () {
    # ifndef cin
        ios::sync_with_stdio(false);
        cin.tie(nullptr);
    # endif
    i64 _ = 1;
    // cin >> _;
    while (_ --) {
        solve ();
    }
    return 0;
}

```

数据结构

1. 循环展开
2. 巴雷特模乘

```

if (val >= x) {
    val = val - (i128(val) * y >> 64) * x;
    if (val >= x) val -= x;
}

```

3. 64 位机子上 64 跑的比 32 快

时间轴一般是顺序，线段树的下标是权值，线段树上二分 `kth`

<https://www.luogu.com.cn/problem/P1972>

<https://www.luogu.com.cn/problem/CF1000F>

<https://www.luogu.com.cn/problem/CF1422F>

四指针树上差分，求第k大，树上差分，以深度前缀链为时间轴

<https://www.luogu.com.cn/problem/P3380>

静态区间，在线单点查询，可以用区间为时间轴

https://www.bilibili.com/video/BV1Zu4y1b7Ki?spm_id_from=333.788.top_right_bar_window_history.content.click

但是在区间查询时，时间轴也可以是权值，而下标是顺序，外面二分 `kth`

<https://www.luogu.com.cn/problem/P2839>

- 覆盖是否影响: <https://www.luogu.com.cn/problem/P1276>
- 双层权值: <https://www.luogu.com.cn/problem/P6186>
- 线段树拆位: <https://www.luogu.com.cn/problem/CF242E> 实际上 (ofast) 就能暴力通过
- 三路归并: <https://www.luogu.com.cn/problem/P6492>
- dfn 序: <https://www.luogu.com.cn/problem/CF620E>
- 考虑拆位, 转化为区间修改问题 <https://codeforces.com/problemset/problem/240/F>
- 区间内区间和 <https://www.luogu.com.cn/problem/P2184>
- 线段树 差分 <https://www.luogu.com.cn/problem/P1438>
- 看到和还有两倍关系就考虑到不超过64次 <https://www.luogu.com.cn/problem/CF992E>
- 区间只出现一次维护最小 last <https://www.luogu.com.cn/problem/CF1000F>
- 括号序, 维护相邻区间差值最小值 <https://www.luogu.com.cn/problem/CF1149C>
- 或运算不超过 64 次 <https://www.luogu.com.cn/problem/CF1004F>
- 递推式转化为差分前缀和矩阵求解 <https://www.luogu.com.cn/problem/CF446C>
- <https://www.luogu.com.cn/problem/P3300>、<https://www.luogu.com.cn/problem/P4121>、

```

    constexpr int inf = 1e9;

    struct Info {
        array<int, 2> max{};
        array<int, 2> min{};
        int sum = 0;
        int ans = 0;
        int rev = 0;
        array<int, 2> res{};
        void show() const {
            # ifdef LOCAL
                cerr << "info: " << sum << ' ' << ans << ' ' << rev << ' ' <<
                max << ' ' << min << ' ' << res;
            # endif
        }
        void apply(const Info &rhs) {
            sum = -sum;
            if (sum == 1) {
                max = {1, 1};
                min = {0, 0};
                ans = 1;
            } else {
                max = {0, 0};
                min = {-1, -1};
                ans = 0;
            }
        }
        void update(const Info &lhs, const Info &rhs, int l, int m, int r) {
            max[0] = std::max(lhs.max[0], lhs.sum + rhs.max[0]);
            max[1] = std::max(rhs.max[1], lhs.max[1] + rhs.sum);
            min[0] = std::min(lhs.min[0], lhs.sum + rhs.min[0]);
            min[1] = std::min(rhs.min[1], lhs.min[1] + rhs.sum);

            res[0] = std::max({lhs.res[0], lhs.sum + rhs.res[0], lhs.rev -
            rhs.min[0]});
        }
    };

```

```

        res[1] = std::max({lhs.max[1] + rhs.rev, lhs.res[1] - rhs.sum,
rhs.res[1]});

        sum = lhs.sum + rhs.sum;
        rev = std::max(lhs.rev - rhs.sum, lhs.sum + rhs.rev);

        ans = std::max({lhs.res[1] - rhs.min[0], lhs.max[1] +
rhs.res[0], lhs.ans, rhs.ans});
    }
    static Info merge(const Info &lhs, const Info &rhs, int l, int m,
int r) {
        Info info = Info();
        info.update(lhs, rhs, l, m, r);
        return info;
    }
};

```

•

势能线段树：

- 在 Info 中加入 cmp 函数再修改为单点修改即可

```

•
template<typename T>
constexpr T power(T x, long long b) {
    T res = 1;
    while (b) {
        if (b & 1) res *= x;
        x *= x;
        b >>= 1;
    }
    return res;
}

template<int P>
struct mod_int {
    int x;
    static int mod;
    constexpr mod_int() : x{} {}
    constexpr mod_int(long long x) : x(norm(x % getMod())) {}

    constexpr int norm(int x) {
        if (x >= getMod()) x -= getMod();
        if (x < 0) x += getMod();
        return x;
    }

    constexpr static void setMod(int x) {
        mod = x;
    }

    constexpr static int getMod() {
        return (P > 0 ? P : mod);
    }

    constexpr mod_int operator-() {
        return -x;
    }
}

```



```

constexpr mod_int &operator+=(mod_int rhs) {
    x = norm(x + rhs.x);
    return *this;
}
constexpr mod_int &operator-=(mod_int rhs) {
    x = norm(x - rhs.x);
    return *this;
}
constexpr mod_int &operator*=(mod_int rhs) {
    x = 111 * x * rhs.x % getMod();
    return *this;
}

constexpr mod_int inv() {
    return power(*this, P - 2);
}
constexpr mod_int &operator/=(mod_int rhs) {
    x = 111 * x * rhs.inv().x % getMod();
    return *this;
}

constexpr friend mod_int operator+(mod_int lhs, mod_int rhs) {
    return lhs += rhs;
}
constexpr friend mod_int operator-(mod_int lhs, mod_int rhs) {
    return lhs -= rhs;
}
constexpr friend mod_int operator*(mod_int lhs, mod_int rhs) {
    return lhs *= rhs;
}
constexpr friend mod_int operator/(mod_int lhs, mod_int rhs) {
    return lhs /= rhs;
}
constexpr friend bool operator==(mod_int lhs, mod_int rhs) {
    return lhs.x == rhs.x;
}
constexpr friend bool operator!=(mod_int lhs, mod_int rhs) {
    return lhs.x != rhs.x;
}

template<class istream>
constexpr friend istream &operator>>(istream &flow, mod_int &rhs) {
    long long x;
    flow >> x;
    rhs = x;
    return flow;
}
template<class ostream>
constexpr friend ostream &operator<<(ostream &flow, mod_int rhs) {
    return flow << rhs.x;
}
};

template<>
int mod_int<0>::mod = 998244353;

```

```

template<int P, int x>
constexpr mod_int<P> invx = mod_int<P>(x).inv();

constexpr int P = 998244353;
using Z = mod_int<P>;

template<class Info, class Tag>
struct LazySegmentTree {
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;
    LazySegmentTree() : n(0) {}
    LazySegmentTree(int n_, Info v_ = Info()) {
        init(n_, v_);
    }
    template<class T>
    LazySegmentTree(std::vector<T> init_) {
        init(init_);
    }
    void init(int n_, Info v_ = Info()) {
        init(std::vector(n_, v_));
    }
    template<class T>
    void init(std::vector<T> init_) {
        n = init_.size();
        info.assign(4 << std::__lg(n), Info());
        tag.assign(4 << std::__lg(n), Tag());
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = init_[l];
                return;
            }
            int m = (l + r) / 2;
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p, l, m, r);
        };
        build(1, 0, n);
    }
    void pull(int p, int l, int m, int r) {
        info[p].update(info[2 * p], info[2 * p + 1], l, m, r);
    }
    void apply(int p, const Tag &v, int l, int r) {
        info[p].apply(v, l, r);
        tag[p].apply(v);
    }
    void push(int p, int l, int m, int r) {
        if (bool(tag[p])) {
            apply(2 * p, tag[p], l, m);
            apply(2 * p + 1, tag[p], m, r);
            tag[p] = Tag();
        }
    }
    void modify(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {

```

```

        info[p] = v;
        return;
    }
    int m = (l + r) / 2;
    push(p, l, m, r);
    if (x < m) {
        modify(2 * p, l, m, x, v);
    } else {
        modify(2 * p + 1, m, r, x, v);
    }
    pull(p, l, m, r);
}

void modify(int p, const Info &v) {
    modify(1, 0, n, p, v);
}

Info rangeQuery(int p, int l, int r, int x, int y) {
    if (l >= x && r <= y) {
        return info[p];
    }
    int m = (l + r) / 2;
    push(p, l, m, r);
    if (m >= y) {
        return rangeQuery(2 * p, l, m, x, y);
    } else if (m <= x) {
        return rangeQuery(2 * p + 1, m, r, x, y);
    } else {
        return Info::merge(rangeQuery(2 * p, l, m, x, y), rangeQuery(2 *
p + 1, m, r, x, y), l, m, r);
    }
}

Info rangeQuery(int l, int r) {
    if (l >= r) return Info();
    return rangeQuery(1, 0, n, l, r);
}

void rangeApply(int p, int l, int r, int x, int y, const Tag &v) {
    if (l >= y || r <= x) {
        return;
    }
    if (l >= x && r <= y && (info[p].cmp(v) || r - l == 1)) {
        apply(p, v, l, r);
        return;
    }
    int m = (l + r) / 2;
    push(p, l, m, r);
    rangeApply(2 * p, l, m, x, y, v);
    rangeApply(2 * p + 1, m, r, x, y, v);
    pull(p, l, m, r);
}

void rangeApply(int l, int r, const Tag &v) {
    return rangeApply(1, 0, n, l, r, v);
}

template<class F>
int findFirst(int p, int l, int r, int x, int y, F pred) {
    if (l >= y || r <= x || !pred(info[p])) {
        return -1;
    }
}

```

```

        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
        push(p, l, m, r);
        int res = findFirst(2 * p, l, m, x, y, pred);
        if (res == -1) {
            res = findFirst(2 * p + 1, m, r, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findFirst(int l, int r, F pred) {
        return findFirst(1, 0, n, l, r, pred);
    }
    template<class F>
    int findLast(int p, int l, int r, int x, int y, F pred) {
        if (l >= y || r <= x || !pred(info[p])) {
            return -1;
        }
        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
        push(p, l, m, r);
        int res = findLast(2 * p + 1, m, r, x, y, pred);
        if (res == -1) {
            res = findLast(2 * p, l, m, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findLast(int l, int r, F pred) {
        return findLast(1, 0, n, l, r, pred);
    }
    void show(int p, int l, int r, int x, int y, int dep = 0) {
        if (l >= y || r <= x) return;
        int m = (l + r) >> 1;
        if (r - l > 1)
            show(p * 2, l, m, x, y, dep + 1);
        for (int i = 0; i < dep; i += 1) {
            cerr << '\t';
        }
        cerr << l << ' ' << r << ' '; info[p].show(), tag[p].show();
        cerr << '\n';
        if (r - l > 1)
            show(p * 2 + 1, m, r, x, y, dep + 1);
    }
    void show(int l, int r) {
        show(1, 0, n, l, r);
    }
};

constexpr int N = 100;

struct Tag {

```

```

Z x = 1; bool flag = true;
void apply(Tag t) {
    x *= t.x;
}
operator bool() {
    return x != 1;
}
void show() {
    cerr << "tag: " << x << ";";
}
};

struct Info {
    Z x = 0;
    bitset<N + 1> power;
    bool cmp(const Tag &t) {
        return !power[t.x.x];
    }
    void apply(const Tag &t, int l, int r) {
        if (t.flag) {
            x *= t.x;
        } else if (power[t.x.x]) {
            x *= t.x - 1;
            power[t.x.x] = 0;
        } else {
            x *= t.x;
        }
    }
    void update(const Info &lhs, const Info &rhs, int l, int m, int r) {
        x = lhs.x + rhs.x;
        power = lhs.power | rhs.power;
    }
    static Info merge(const Info &lhs, const Info &rhs, int l, int m, int r)
    {
        Info info = Info();
        info.update(lhs, rhs, l, m, r);
        return info;
    }
    void show() {
        cerr << "info: " << x << "; ";
    }
};

```

- ICPC 网络赛 第二场 L Euler Function
 - 欧拉函数性质:
 - 若 $i \bmod p = 0$, 其中 p 为质数, 则 $\varphi(i * p) = p * \varphi(i)$, 否则 $\varphi(i * p) = (p - 1) * \varphi(i)$ 。
- 区间最值覆盖问题
 - 维护起来有点麻烦, 最值和次值即可, add 和 mul 时, 先对 tag 进行操作, 再进行最值处理
 - 区间历史最大值
 - push

```

void push(int p, int l, int m, int r) {
    if (bool(tag[p])) {

```

```

        Info fix = Info::merge(info[p * 2], info[p * 2 + 1], l,
m, r);

        Tag t = tag[p];
        if (fix.x[0] != info[2 * p].x[0]) {
            t.add[2] = t.add[0];
            t.add[3] = t.add[1];
        }
        apply(2 * p, t, l, m);
        t = tag[p];
        if (fix.x[0] != info[2 * p + 1].x[0]) {
            t.add[2] = t.add[0];
            t.add[3] = t.add[1];
        }
        apply(2 * p + 1, t, m, r);
        tag[p] = Tag();
    }
}

```

- tag && inf

- ```

constexpr i64 inf = 1e18;

struct Tag {
 array<int, 4> add{}; int set = inf;

 template<typename Info>
 void apply(Tag t, Info i) {
 if (t.set != inf) {
 add[2] -= i.x[0] - t.set;
 } else {
 add[1] = std::max(add[1], add[0] + t.add[1]);
 add[0] += t.add[0];
 add[3] = std::max(add[3], add[2] + t.add[3]);
 add[2] += t.add[2];
 }
 }
 operator bool() {
 return add[0] != 0 || add[1] != 0 || add[2] != 0 ||
add[3] != 0 || set != inf;
 }
 void show() const {
#ifdef LOCAL
 cerr << "tag: " << add << ' ' << set << ";";
#endif
 }
};

struct Info {
 array<int, 3> x{0, -inf, 1};
 i64 sum = 0;
 int hismax = 0;
 constexpr bool cmp1 (const Tag &t) {
 if (t.add[0] || t.add[1] || t.add[2] || t.add[3]) return
false;
 return t.set >= x[0];
 }
}

```

```

constexpr bool cmp2 (const Tag &t) {
 if (t.add[0] || t.add[1] || t.add[2] || t.add[3]) return
true;
 return t.set > x[1];
}
void apply(const Tag &t, int l, int r) {
 if (t.set != inf) {
 sum -= (x[0] - t.set) * x[2];
 x[0] = t.set;
 } else {
 sum += (r - l - x[2]) * t.add[0] + x[2] * t.add[2];
 hismax = std::max(hismax, x[0] + t.add[3]);
 x[1] += t.add[0];
 x[0] += t.add[2];
 }
}
void update(const Info &lhs, const Info &rhs, int l, int m,
int r) {
 hismax = std::max(lhs.hismax, rhs.hismax);
 if (lhs.x[0] > rhs.x[0]) {
 x[0] = lhs.x[0];
 x[1] = std::max(lhs.x[1], rhs.x[0]);
 x[2] = lhs.x[2];
 } else if (lhs.x[0] == rhs.x[0]) {
 x[0] = lhs.x[0];
 x[1] = std::max(lhs.x[1], rhs.x[1]);
 x[2] = lhs.x[2] + rhs.x[2];
 } else {
 x[0] = rhs.x[0];
 x[1] = std::max(lhs.x[0], rhs.x[1]);
 x[2] = rhs.x[2];
 }

 sum = lhs.sum + rhs.sum;
}
static Info merge(const Info &lhs, const Info &rhs, int l,
int m, int r) {
 Info info = Info();
 info.update(lhs, rhs, l, m, r);
 return info;
}
void show() {
#ifdef LOCAL
 cerr << "info: " << x << ' ' << ' ' << hismax << ' ' <<
sum << " ";
#endif
}
};

```

## 动态开点

<https://www.luogu.com.cn/problem/P5459> 线段树上位移操作

# 网络流

---

## 二分图

---

### 8.2

---

#### P1129

---

- 树是一个二分图
- 最大流复杂度为  $O(nm)$
- 二分图复杂度为  $O(n\sqrt{n})$

#### P1129

---

- 简述：
  - 矩阵可交换行列，求是否能使主对角线为 1
- 仅交换一方即可，两方无用

# 费用流笔记

---

## 拆式子拆点

---

套板子

将题意转化为板子

#### P2053 \*

前言：

本蒟蒻觉得这道题出的不错，一开始我只想到了二分答案，根本没有想到如何构造网络流，看了题解后才大概理解的。

思路：

假设我们现在只有一个修车师傅，共有 $A_1, A_2, \dots, A_n$ 这 $n$ 辆车，那么所有人的等待时间就分别为：

$A_1 + A_2 + \dots + A_n, A_2 + A_3 + \dots + A_n, \dots, A_{n-1} + A_n, A_n$ 。

**那所有人的等待时间呢？**

加起来，我们发现所有人的等待时间应该是：

$A_1 * n + A_2 * (n-1) + \dots + A_n$

**那我们对每辆车被等待的时间考虑，发现越后修的被等待的时间越少**

**而且显而易见的，所有车被等待的时间即为所有人等待的时间**

---



## 做法：

那我们从每辆车被等待的时间思考，思路也不是很难了，

将M位师傅拆成 $N \times M$ 个点，第 $(i-1) \times N + j$ 个点表示的是在修第j辆车的第i位师傅，并将这个点连向每辆车，容量为1，边权为 $C_{ki} \times j$ （ $C_i$ 为第i辆车被第j位师傅修所花的时间， $1 \leq k \leq N$ ），

然后再源点朝M个车点建边， $N \times M$ 个师傅点朝汇点建边，都是边权为0，容量为1的边，最后跑最小费用最大流就行了。

同时建反向边应该不用我赘述了吧

注：本来是先修被等的车越多，但是反过来仔细想先也没有什么问题

## P3159 \*

### 此题算法:费用流

题目很简洁，做法很恶心的典型。

因为是网络流题，所以模板就不说了，只考虑加边。

#### 大致思路：

##### 简化问题

记录初始和结束状态，把白棋看作没棋。

把开始结束都有黑棋的格子看作没棋。

如果开始结束时黑棋数不等，-1-1掉。

##### 加边

1.拆点，每个格子有格子xx和格子yy。

控制格子交换次数。

2.ss向每个黑棋格xx连流量11费用00的边。

表示需匹配状态。

3.每个黑棋格yy向tt连流量11费用00的边。

表示匹配状态。

4.每个格子xx向对应yy连流量 $((\text{允许交换数}+2)\div 2)$ 费用00的边。

两次交换只会消耗11的流量。

※.如果格子初始或结束时有黑棋并且允许交换数为奇数，在上面那条边上附上11的流量。

不交换本来就要通过的流量。

5.每个格子yy向八连通的格子xx连流量 $\text{infinf}$ 费用11的边。

用来交换。

然后跑模板就好了，网络流的题都差不多。

# 输出方案

[P2770](#) [P3356](#)

对于每条路径，从起点开始搜索，每搜到一个点，选一条反向边有剩余容量（说明被走过）的临边走过去，并把反向边的剩余容量减去 1，直到走到终点。

```
for (int i = 0; i < n - 1;) {
 cout << names[i] << '\n';
 for (int j = g.r[i + n]; ~j; j = g.t[j]) {
 if (!g.e[j].f && g.e[j ^ 1].c) {
 g.e[j ^ 1].c--;
 i = g.e[j].v;
 break;
 }
 }
}
for (int i = n - 1; i > 0;) {
 cout << names[i] << '\n';
 for (int j = g.r[i]; ~j; j = g.t[j]) {
 if (!g.e[j].f && g.e[j].c) {
 i = g.e[j].v - n;
 break;
 }
 }
}
```

```
for (int i = 0; i < n; i += 1) {
 int x = 0, y = 0;
 int now = id[x][y][0];
 for (; x != p - 1 || y != q - 1;) {
 for (int j = g.r[now ^ 1]; ~j; j = g.t[j]) {
 if (!g.e[j ^ 1].c) {
 continue;
 }
 for (int c = 0; auto [dx, dy] : d) {
 auto [tx, ty] = make_pair(x + dx, y + dy);
 if (leg(tx, ty) && id[tx][ty][0] == g.e[j].v) {
 cout << i + 1 << ' ' << c << '\n';
 x = tx, y = ty;
 now = g.e[j].v;
 break;
 }
 ++ c;
 }
 g.e[j ^ 1].c--;
 break;
 }
 }
}
```

# 数据结构优化建图

---

[P5331](#)

cdq优化建图

## 一流对区间流问题

---

都能用最小费用可行流来做

## 限制单点最大值问题

[P3358](#)

## 限制单点最小值问题

[P3980](#)

每个点向下一个点连  $\inf - w[i], 0$

每个区间的左端点向右端点 + 1 连  $\inf, c$

由于最大流思想，费用流会满足条件

## 动态加点问题

---

[P2053](#)

[P2050](#)

其实根据上面的建模可以发现，好多  $(j, w)$  的点是没有用到的。

根据上述的贪心证明可知：被用到的一定是从  $(j, 1)$  开始的连续的几个层。

那么我们为什么不一边跑流一边加点呢？这样我们就能省去很多无用的点。

我们设第  $j$  个厨师已经加到了第  $\text{top}_j$  层。若再一次跑流之后， $(j, \text{top}_j)$  被用掉了，那么我们直接把  $(j, \text{top}_j + 1)$  加进图里面就好了。

这样点数的规模就降到了  $O(n+m+p)$ ，边数的规模降到了  $O(np)$ 。

## 费用流最小路径覆盖

---

[P2469](#)

对于所有东西都得经过

1. 由源点向第一部分点连容量为1，费用为零的点。
2. 由第二部分点向汇连容量为1，费用为零的点。
3. 由第一部分点向第二部分点连容量为1，费用为耗时。
4. 源点向第二部分的点连容量为1，费用为航行时间。
5. 最后跑一边最小费用最大流，即可求出答案。

# 最大权完美匹配

[P3967](#) 关键边

[P4134](#) 必须选择两个，考虑对称选取答案，拆点

# 费用流二分

[P3705](#) 分数规划

# Dilworth定理


754e546abc34113328406dd91512d1d0

image-20240827154128820

# 限制性匹配问题

## [JSOI2009](#) 球队收益 / 球队预算

### 题目描述

在一个篮球联赛里，有 $n$ 支球队，球队的支出是和他们的胜负场次有关系的，具体来说，第 $i$ 支球队的赛季总支出是 $C_i \times x^2 + D_i \times y^2, D_i \leq C_i$ 。（赢得多，给球员的奖金就多嘛）

其中 $x, y$ 分别表示这只球队本赛季的胜负场次。现在赛季进行到了一半，每只球队分别取得了 $a_i$ 场胜利和 $b_i$ 场失利。而接下来还有 $m$ 场比赛要进行。问联盟球队的最小总支出是多少。

P4307

## P4249 [WC2007](#) 剪刀石头布

### 题目描述

在一些一对一游戏的比赛（如下棋、乒乓球和羽毛球的单打）中，我们经常会遇到  $A$  胜过  $B$ ， $B$  胜过  $C$  而  $C$  又胜过  $A$  的有趣情况，不妨形象的称之为剪刀石头布情况。有的时候，无聊的人们会津津乐道于统计有多少这样的剪刀石头布情况发生，即有多少对无序三元组  $(A, B, C)$ ，满足其中的一个人在比赛中赢了另一个人，另一个人赢了第三个人而第三个人又胜过了第一个人。注意这里无序的意思是说三元组中元素的顺序并不重要，将  $(A, B, C)$ 、 $(A, C, B)$ 、 $(B, A, C)$ 、 $(B, C, A)$ 、 $(C, A, B)$  和  $(C, B, A)$  视为相同的情况。

有  $N$  个人参加一场这样的游戏的比赛，赛程规定任意两个人之间都要进行一场比赛：这样总共有  $\frac{N \times (N-1)}{2}$  场比赛。比赛已经进行了一部分，我们想知道在极端情况下，比赛结束后最多会发生多少剪刀石头布情况。即给出已经发生的比赛结果，而你可以任意安排剩下的比赛的结果，以得到尽量多的剪刀石头布情况。

反向考虑没有三元环的情况

得到差分建图

简化建图

## ZJOI2011 营救皮卡丘

### 题目描述

皮卡丘被火箭队用邪恶的计谋抢走了！这三个坏家伙还给小智留下了赤果果的挑衅！为了皮卡丘，也为了正义，小智和他的朋友们义不容辞的踏上了营救皮卡丘的道路。

火箭队一共有  $N$  个据点，据点之间存在  $M$  条双向道路。据点分别从 1 到  $N$  标号。小智一行  $K$  人从真新镇出发，营救被困在  $N$  号据点的皮卡丘。为了方便起见，我们将真新镇视为 0 号据点，一开始  $K$  个人都在 0 号点。

由于火箭队的重重布防，要想摧毁  $K$  号据点，必须按照顺序先摧毁 1 到  $K - 1$  号据点，并且，如果  $K - 1$  号据点没有被摧毁，由于防御的连锁性，小智一行任何一个人进入据点  $K$ ，都会被发现，并产生严重后果。因此，在  $K - 1$  号据点被摧毁之前，任何人是不能够经过  $K$  号据点的。

为了简化问题，我们忽略战斗环节，小智一行任何一个人经过  $K$  号据点即认为  $K$  号据点被摧毁。被摧毁的据点依然是可以被经过的。

$K$  个人是可以分头行动的，只要有任何一个人在  $K - 1$  号据点被摧毁之后，经过  $K$  号据点， $K$  号据点就被摧毁了。显然的，只要  $N$  号据点被摧毁，皮卡丘就得救了。

野外的道路是不安全的，因此小智一行希望在摧毁  $N$  号据点救出皮卡丘的同时，使得  $K$  个人所经过的道路的长度总和最少。

请你帮助小智设计一个最佳的营救方案吧！

我们尝试将上述模型应用到此题中。首先，每个人都从 0 号点出发，所以 0 可以作为  $K$  条路径的起点，因而它可以与  $K$  个点匹配。所以  $\langle S, 0 \rangle$  这条边的流量我们要改成  $K$ 。此外我们要引进费用的概念，每次匹配就代表使用了这条边，也就要消耗这条边的代价。这样图就建完了，跑费用流即可。

## 洞穴遇险

### 题目描述

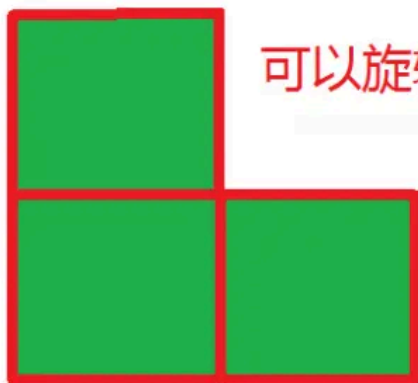
整个洞穴是一个  $N * N$  的方格图，每个格子形如  $(X, Y)$ ,  $1 \leq X, Y \leq N$ 。其中  $X$  表示从上到下的行数， $Y$  表示从左到右的列数。 $(1, 1)$  在左上角， $(1, N)$  在右上角， $(N, 1)$  在左下角， $(N, N)$  在右下角。

满足  $X + Y$  为奇数格子的有一个不稳定性  $V_{X,Y}$ ， $X + Y$  为偶数的格子的不稳定性为 0。

ZRQ 现在手里恰巧有  $M$  个可以支撑洞穴的柱子，柱子的力量可以认为是无穷大。

只要支撑住了一个格子那么这个格子的不稳定性将降为 0。

每个柱子是  $L$  型的，它除了要占据当前的格子外，还需要占据两个相邻的格子（这三个格子形成  $L$  型，可以选择任意方向放置，一共有 4 个方向）。



可以旋转，共4种形态



柱子占据相邻的格子不会降低其不稳定性（换句话说就是柱子只有在拐角处有力量）。

有些格子的顶已经塌下来了，无法在其位置放置柱子了，这些格子也不能被占据了。这样已经塌了的格子有 $K$ 个（他们的不稳定性都为0,即使 $X + Y$ 为奇数，塌下来的格子的不稳定性也会为0）。

ZRQ想问你，在放置一些柱子后，最小的不稳定性之和为多少（可以不将 $M$ 个柱子都放完）。

一般连续选三个的问题都是黑白染色



## 分层图思想

## 长度限制性条件

## 最长不下降子序列

[P2766](#)

[P3308](#)

## 小规模建图思想

在不清楚使用什么模型时，可以使用小规模数据模拟，最后得到一个较好的模型

## 对偶图思想

平面图最小割转对偶图最短路

## 网络流的退流

我们发现，如果是加边的话，可以直接加进去然后直接在原来跑完的基础上继续跑，而难处理的是减边。

这个时候我们就用到优秀的退流啦！假设源点和汇点是  $S$  和  $T$ ，要删去一条边  $(u, v)$ ，那么我们只需要以  $u$  为源点，向  $S$  跑一次最大流，然后再以  $T$  为源点，向  $v$  跑一次最大流即可。最后将  $(u, v)$  这条边以及其反向边的流量设为 0 即可。

### [P3308](#) \*

或者直接手动全部退流，甚至更快doge

## 一选二问题

### [P1231](#)

### [P2891](#)

转换为



## 最小化最小割和割边数量

### [P1344](#)

因为本题既要输出最小割的值又要输出割的边数，前者好求关键是后者如何去求更简单，容易想到我们可以直接建两次图，一次按原边权建图跑最大流求得最小割，再按边权为1建图跑最大流求割的边数，这是一种思路；

当然我们完全可以换种思路用一次最大流搞定，只需建图时将边权  $w = wa + 1$  ( $w$  为本来的边权， $a$  为大于 1000 的数)，这样我们能求得最大流  $ans$ ，则最小割的值为  $ans/a$ ，割的边数为  $ans \% a$ 。这很容易理解，但是还是解释一下：因为最小割的边集中有  $w_1 + w_2 + w_3 \dots + w_n = ans$  (这个  $ans$  为本来的最小割)，所以必然有  $w_1a + w_2a + w_3a \dots + w_na = ansa$ ，于是必然有  $w_1a + 1 + w_2a + 1 + w_3a + 1 \dots + w_na + 1 = ans * a + k$  ( $k$  为最小割的边数， $k \leq m \leq 1000$ )，这样就很明显了，因为边数  $m$  不大于 1000，所以  $k$  的最大值为 1000，我们只要使设定的  $a$  的值大于 1000，那么按上述方法建图，跑出的最大流除以  $a$  就是最小割的值  $ans$ ，最大流模  $a$  就是最小割的边数  $k$ 。

## 二分答案 网络流

### [P2402](#)

二分答案 + 网络流

建图：

- 1、源点  $\rightarrow$  每块田，边权为牛的数量
- 2、每个牛棚  $\rightarrow$  汇点，边权为牛棚最多能容纳的牛
- 3、对中间的边进行二分答案。具体操作如下。

使用 floyd 对两个点之间的最短距离预处理，对最小时间进行二分答案。如果在这个时间内可以从  $ii$  号田地走向  $jj$  号牛棚，就连一条边，容量为  $\infty$ ，最终检验最大流是否等于牛的数量即可。

### [P3425](#)

## 并行思维

[P3153](#)

如果这题换个问法：能不能跳 $a$ 支舞曲

我们来看看

把每个人拆成喜欢和不喜欢两个点

从 $S$ 向每个男生连容量为 $a$ 的边，表示限制 $a$ 支舞曲

再从男生连向喜欢和不喜欢的两个点，

但是这样子没法限制，因为只说了不能和超过 $K$ 个不喜欢的人跳舞

所以可以直接从 $S$ 连向男生喜欢，容量为 $a$

再从男生喜欢连向男生不喜欢连边，容量为 $K$

这样的话就解决了这个问题

接下来就很好办了

男生喜欢连向女生喜欢

男生不喜欢连向女生不喜欢

而女生之间的连边类似于男生

（你就想，如果这个图反过来是一样的，所以怎么连边就很清晰了）

这个时候跑最大流

求出来的就是最大的匹配数

如果最大流恰好等于 $a \cdot n$

也就是恰好 $a \cdot n$ 组匹配，意味着可行

现在再来看这个问题

既然要求最大的 $a$

所以就二分一下

然后每次把图重构一下流量


二分就行了

图论中除了匹配和边覆盖可以做一般图，其它均是  $np$  困难问题

由于带花树算法无用，所以（一般）没有一般图的情形

## 最小不相交路径覆盖问题

往往这样建图

最小路径覆盖模型



## [P2764](#)

### 题目描述

给定有向图  $G = (V, E)$ 。设  $P$  是  $G$  的一个简单路（顶点不相交）的集合。如果  $V$  中每个定点恰好在  $P$  的一条路上，则称  $P$  是  $G$  的一个路径覆盖。 $P$  中路径可以从  $V$  的任何一个定点开始，长度也是任意的，特别地，可以为 0。 $G$  的最小路径覆盖是  $G$  所含路径条数最少的路径覆盖。设计一个有效算法求一个 DAG（有向无环图） $G$  的最小路径覆盖。

## [P2765](#)

还没搞懂正确性

## [P2172](#)

版

### DAG的最小可相交路径覆盖 杂

**算法：**先用floyd求出原图的传递闭包，即如果a到b有路径，那么就加边a->b。然后就转化成了最小不相交路径覆盖问题。

**证明：**为了连通两个点，某条路径可能经过其它路径的中间点。比如1->3->4，2->4->5。但是如果两个点a和b是连通的，只不过中间需要经过其它的点，那么可以在这两个点之间加边，那么a就可以直达b，不必经过中点的，那么就转化成了最小不相交路径覆盖。

[链接](#)

## [P5769](#)

以事件为中心建图，忽略原图，看起来并行的事件转化为最小不相交路径覆盖问题

## 二分图多重匹配问题

---

比较简单

## [P3254](#) [P2763](#)

裸体

## 二分图最大独立集

---

注意不要建立双向边，否则会神秘wa

[P3355](#) [P4304](#)

# 多源点多汇点网络流的特殊情况 —— 二源点二汇点，指定流向

## P3163 \*

交换一方缘汇点跑两次，正确即可

另外：

我们已经介绍了如何求解恰有一个源点和一个汇点的网络流。那么，如果有多个源点和汇点，并且它们都有对应的最大流出容量和流入容量限制时该怎么做呢？答案很简单，只要增加一个超级源点  $s$  和一个超级汇点  $t$ ，从  $s$  向每个源点连一条容量为对应最大流出容量的边，从每个汇点向  $t$  连一条容量为对应最大流入容量的边。不过，如果源和汇之间存在对应关系（从不同源点流出的流要流入指定的汇点）时，是无法这样求解的。这种情况被称为多物网络流问题，尚未有已知的高效算法，这类问题也几乎不会出现在程序设计竞赛当中

## 二分图最大点覆盖集

先说一下什么叫二分图的最小点覆盖，简而言之就是我们希望用尽可能少的点去使所有的边都被选中，这里解释一下：如果说一条边的任意一个顶点被选中那么这条边也被选中。

Konig定理：二分图中，最小点覆盖数 = 最大匹配数。

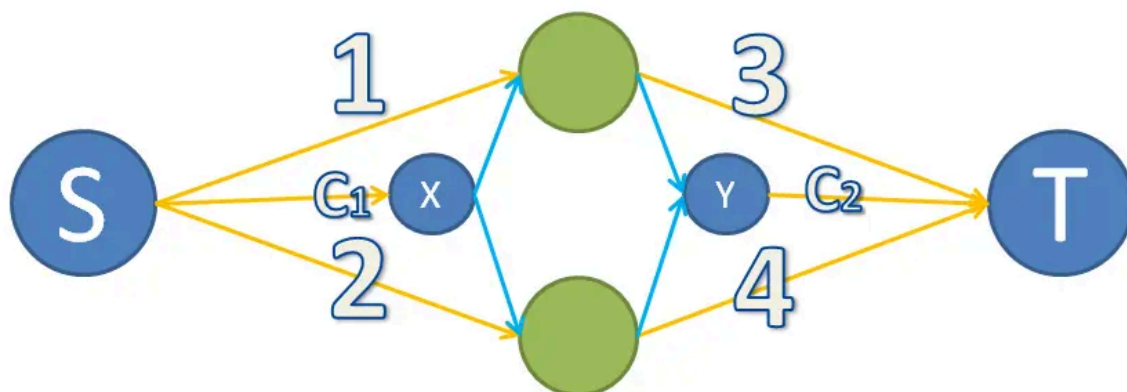
## 8.3

## 二分图最小割

默念一百遍：满足二选一的性质就是二分图

## P1361 P2774 虚点

意见相同利益，构建



## [P4174](#)

将代价提前付出，转化为利益二分图

## [P2057](#)

意见相左利益构建双向边（因为无法判断方向），流量为利益

## [P2774](#)

- 简述：矩阵邻边不能选，选任意数总和最大
- 二分最大独立集
  - 图论中除了匹配和边覆盖可以做一般图，其它均是 np 困难问题
  - 所以独立集询问问题里面一定是二分图

## 生成树换边

---

### [P5039](#) \*

将最小生成树换边的本质和最小割联系起来

### [P5934](#) \*

最小生成树和最大生成树都做

## 带限制条件的最小割 ———— 回流边的研究

---

### [P5039](#)

- 题目描述
  - 经过千辛万苦小 A 得到了一块切糕，切糕的形状是长方体，小 A 打算拦腰将切糕切成两半分给小 B。出于美观考虑，小 A 希望切面能尽量光滑且和谐。于是她找到你，希望你能帮她找出最好的切割方案。
  - 出于简便考虑，我们将切糕视作一个长  $P$ 、宽  $Q$ 、高  $R$  的长方体点阵。我们将位于第  $z$  层中第  $x$  行、第  $y$  列上的点称  $(x, y, z)$ ，它有一个非负的不和谐值  $v(x, y, z)$ 。一个合法的切面满足以下两个条件：
    - 与每个纵轴（一共有  $P \times Q$  个纵轴）有且仅有一个交点。即切面是一个函数  $f(x, y)$ ，对于所有  $(x, y)(x \in [1, P], y \in [1, Q])$ ，我们需指定一个切割点  $f(x, y)$ ，且  $1 \leq f(x, y) \leq R$ 。
    - 切面需要满足一定的光滑性要求，即相邻纵轴上的切割点不能相距太远。对于所有的  $1 \leq x, x' \leq P$  和  $1 \leq y, y' \leq Q$ ，若  $|x - x'| + |y - y'| = 1$ ，则  $|f(x, y) - f(x', y')| \leq D$ ，其中  $D$  是给定的一个非负整数。
- 可能有许多切面  $f$  满足上面的条件，小 A 希望找出总的切割点上的不和谐值最小的那个。

回流边的作用可以反证法证明

 QQ\_1722754239901

## P6054

粗略分析：反向边只能截断作用

 QQ\_1722758844185

粗略分析：正向边强制正向

 QQ\_1722758926763

实际上两者是一样的，只能选顺时针更倾斜的方向，建出两次边时只能选择夹角方向

 QQ\_1722759413891

注意细节

 QQ\_1722760133834

 094cf9a115c2662a94da06734790bd78

 5c05fe3d82e82cbe62adc91fb0ebc9e3

## 最小割树

一个图中只有最多 $n$ 种不同的最小割

建树代码

```
vector<int> a(n);
iota(a.begin(), a.end(), 0);
vector<vector<array<i64, 2>>> adj(n);
auto build = [&] (auto &&build, int l, int r) -> void {
 if (r - l == 0) {
 return;
 }
 int u = a[l], v = a[r];
 i64 cap = g.flow(u, v);
 adj[u].push_back({v, cap});
 adj[v].push_back({u, cap});
 cout << u + 1 << ' ' << v + 1 << ' ' << cap << '\n';
 auto c = g.minCut();
 for (int i = l, j = r; i < j; i += 1) {
 for (; i < j && c[a[i]] == 1 && c[a[j]] == 1; j -= 1);
 if (i < j && c[a[i]] == 1) {
 swap(a[i], a[j]);
 }
 }
 int m = l;
 while (m < r && !c[a[m + 1]]) m++;
 for (int i = 0; i < g.c.size(); i += 2) {
 tie(g.c[i], g.c[i + 1]) = make_pair(g.c[i] + g.c[i + 1], 0);
 }
 build(build, l, m);
 build(build, m + 1, r);
};
build(build, 0, n - 1);
```

[P4123](#) [P4897](#) [P3329](#)

板子

## 最小割最大闭合子图模型

[P2762](#) [P4174](#)

将代价提前支出，转化为虚点经典模型

公式化说明

给定一个有向图，点有点权，选择一个子图，满足子图上**如果选择了一个点就必须选择它后继的所有点**。最大化点权和。

这是一个经典的网络流问题，如果一个点被选择了则后继必须被选择，那么称该图是 **闭合的**，因此该问题叫做**最大权闭合子图问题**。可以使用最小割解决。

具体的建图方法为：

源点向所有正权点连结一条容量为权值的边

保留原图中所有的边，容量为正无穷

所有负权点向汇点连结一条容量为权值绝对值的边

则原图的最大权闭合子图的点权和即为所有正权点权值之和减去建出的网络流图的最小割。

[P2057](#)

多重嵌套

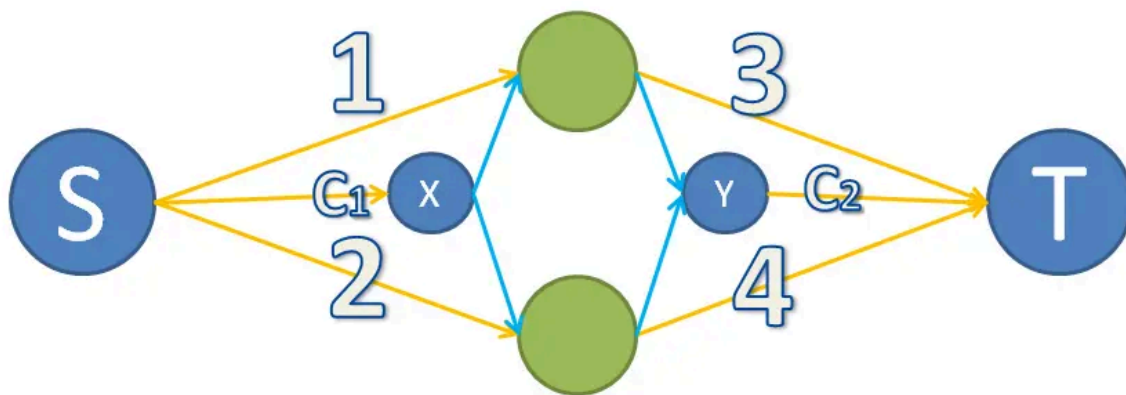
拓扑去环

[P2805](#)

## 虚点

[P1361](#) [P2774](#)

意见相同利益，构建



## [P1935](#) 对于对立情况进行黑白染色

a 与 b 的染色情况对立时，获得利益 c，那么黑白染色建图即可

## 不好归类，但相当经典

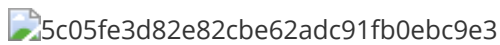
### [P4177](#) ?

## 问题集锦

## 传递性失效

### [P6054](#)

注意细节



## 最小割相同权值产生多割点假象

板子使用示例

```
g.flow(s, t + n);
auto e = g.edges();
auto minCut = g.minCut();
vector<int> ans;
for (auto [u, v, c, f] : e) {
 if (minCut[u] && !minCut[v]) {
 ans.push_back(u);
 }
}

for (auto i : ans) {
 cout << i + 1 << ' ';
}
cout << '\n';
```

`minCut` 表示图中的点在左边还是右边，两边不一样即为最小割

```
2
1 2
1
1
```

# Hall 定理

解决二分图完美匹配问题的充要条件

设二分图左侧有  $x$  个点，右侧有  $y$  个点。不妨设  $x \leq y$ ，则有：

**二分图存在完美匹配**  $\Leftrightarrow \forall k \in [1, x]$ ，均满足从左侧选出  $k$  个点，连向右侧的点集大小不小于  $k$

二分图完美匹配：匹配数为  $\min(x, y)$ 。

正确性十分显然。

最大匹配的一种求法 **推论**：设左侧点集  $S$  连向右侧的点集为  $T$ ，则二分图最大匹配数为  $n - \max(|S| - |T|)$

霍尔定理的关键在于推狮子转化，维护连续区间是否合法

不定一格，寻找性质

## 1. P3488

滑冰俱乐部初始有 1 到  $n$  号码溜冰鞋各  $k$  双，已知  $x$  号脚的人可以穿  $x$  到  $x+d$  号码的鞋子。

现在有  $m$  次操作，每次两个数  $r, x$ ，表示  $r$  号脚的人来了  $x$  个， $x$  为负表示离开。对于每次操作，输出溜冰鞋是否足够。

$r \leq n-d, 1 \leq n, k, m \leq 5 \times 10^5, k \leq 109$ 。

很容易想到将人和溜冰鞋一起建二分图，傻瓜式连边，并对于每次操作都跑一遍最大流，但这时间复杂度显然爆炸！

由于题目只询问是否存在完美匹配，所以考虑使用 Hall 定理。

设  $s_x$  为当前脚码为  $x$  的人数。由 Hall 定理可知，若该二分图存在完美匹配，则有：

$$\forall l, r \in [1, n-d], l \leq r, r \sum_{i=l}^r s_i \leq k \times (r-l+1+d)$$

不妨将右侧常变量分离，则有：

$$r \sum_{i=l}^r s_i \leq k \times d$$

于是我们动态维护所有区间  $\sum s_i - k$  的最大值，每次询问时判断是否大于  $k \times d$  即可。

用线段树可以做到  $O(m \log n)$ 。

## 2. CF981F Round Marriage

$n$  个新郎和  $n$  个新娘围成一个环，长度为  $L$ ，第  $i$  个新郎位置为  $a_i$ ，第  $i$  个新娘位置为  $b_i$ ，需要将他们两两配对，最小化新郎和新娘距离的最大值。

$1 \leq n \leq 2 \times 10^5, 1 \leq L \leq 10^9$

首先根据 "最小值最大"，先二分答案，考虑如何判定「当新郎可以和距离其  $x$  以内的新娘配对时，是否存在完美匹配」。

又观察到了「完美匹配」，考虑使用 Hall 定理。

首先断环成链，设第  $i$  个新郎向左可以匹配到第  $nl_i$  个新娘，向右可以匹配到第  $nr_i$  个新娘。由于单次判定对于每个新郎的  $x$

值相同，故可以利用单调性  $O(n)$  求出每个新郎的  $nl, nr$  值。

那么根据 Hall 定理，有：

$$r-l+1 \leq nr-nl+1$$

观察到该式子只和  $l, r, r$  有关，考虑将  $l, r, r$  拆开：

$$nr-r \geq nl-l$$

于是从左向右扫时，动态更新之前所有  $nl-l$  的最大值，判断是否不大于当前  $nr-r$  即可。

注意断环时将  $a$  拆成两份，将  $b$  拆成四份，才能便捷地在处理中表示所有情况。

时间复杂度  $O(n \log n)$ 。

### 3. [Loj6062. 「2017 山东一轮集训 Day2」 Pair](#)

给出一个长度为  $n$  的数列  $a_i$  和一个长度为  $m$  的数列  $b_i$ ，求  $a_i$  有多少个长度为  $m$  的连续子数列能与  $b_i$  匹配。

两个数列可以匹配，当且仅当存在一种方案，使两个数列中的数可以两两配对，两个数可以配对当且仅当它们的和不少于  $h$ 。

$$1 \leq m \leq n \leq 1.5 \times 10^5, 1 \leq a_i, b_i, h \leq 10^9$$

不妨将  $b$  数组降序排序，则对于每一个  $a_i$ ，对应的连边区间都为  $b$  数组的一个前缀。

所以对于  $a$  中每一个长度为  $m$  的区间，都会在  $b$  中对应  $m$  个前缀。

设区间内值为  $x$  的  $a_i$  的个数为  $p_x$ 。根据 Hall 定理，此时需要满足的约束是：

$$\forall x \in [1, m], x \sum_{i=1}^x p_i \leq x$$

于是考虑使用权值线段树进行维护，初始将线段树第  $i$  个位置的值设为  $-i$ 。对于每一次区间移动，设需要修改的值为  $p_k$ ，则在线段树上修改区间  $[p_k, m]$  即可。

判定时查询线段树最大值是否大于 0，并计数。

时间复杂度  $O(m \log n)$ 。

### 4. [CF1009G Allowed Letters](#)

给定一个长为  $n$  的串，字符集  $a-f$ 。你可以重排这个串，满足指定  $m$  个位置上只能放特定的字符， $m$

个位置以及字符集会给出，求字典序最小的串。

$$1 \leq m \leq n \leq 10^5, \text{ 保证 } m \text{ 个位置互不重复。}$$

*bitmask sosdp* 暴力判断

观察到字符集很小，考虑到在字符集上做文章。

不妨从  $a$  到  $f$  枚举每个位置都填什么字符，对于当前决策字符  $s$ ，若填完  $s$  后，对于后面的位置，满足对于完美匹配的判定，则说明  $s$  合法，可以填  $s$ 。

于是问题就在于，如何高效判定完美匹配。

具体地，根据 Hall 定理，若存在完美匹配，需满足选出字符集的一个子集后，设子集中余下需填字符总数为  $num$ ，剩下位置中还可以填这些字符的位置总数为  $tot$ ，则需要满足  $num \leq tot$ 。但暴力判定这个，单次复杂度仍然是  $O(n)$  的，炸炸炸！



但是注意到，每次填完一个字符之后，对于每个子集的 num 和 tot 来说是好维护的。于是可以考虑作一个简单的状压

dp，将字符集选择状态压起来，每次填写后，动态维护 numS 和 totS，如此便可以做到  $O(1)$  判定了。

故最终时间复杂度为  $O(|S|n^2|S|)$ ，其中  $|S|=6$ 。

## 5. [ARC076F] Exhausted?

$n$  个人， $m$  张椅子，第  $i$  个人可以坐在前  $l_i$  椅子上或者后  $r_i$  个椅子上，每个人只能坐一把椅子。求最少有几个人坐不到椅子。

$1 \leq n, m \leq 2 \times 10^5, 0 \leq l_i < r_i \leq m+1$ 。

首先自然可以想到网络流，使用前后缀优化建图可以拥有较为优秀的复杂度，但是数据范围太大，很难通过本题。

于是考虑使用复杂度更为优秀的类网络流解法，使用 Hall 定理的推论解决本题。

人的集合不好枚举，我们可以考虑枚举枚举前  $i$  个椅子和后  $j$  个椅子构成的区间所完全包含的人数。但这仍然是  $O(m^2)$  的。

观察到这其实是二维数点，每次询问其实就是在查询  $l_i \in [1, l]$  且  $r_i \in [r, m]$  的人数。

于是可以考虑采用扫描线消维的方式，枚举  $l$ ，用线段树维护  $r$  上的最大值，将  $l_i=l$  的所有  $r_i$  扔到线段树里；同时由于  $l$

的向右扩展， $r$  左范围默认向右移动。随着扫描的进行动态维护最大值即可。有一些简单的细节需要处理。

时间复杂度  $O(m \log m)$ 。

# 自动机

基本定义与约定：

- 称字符串  $T$  匹配  $S$  为  $T$  在  $S$  中出现。
- **模式串**：相当于题目给出的 **字典**，用于匹配的字符串。下文也称 **单词**。
- **文本串**：被匹配的字符串。
- 更多约定见 常见字符串算法。

## 1. AC 自动机 ACAM

前置知识：字典树，[KMP](#) 算法与 **动态规划** 思想。

AC 自动机是一类确定有限状态自动机，这说明它有完整的 DFA 五要素，分别是起点  $s$  (Trie 树根节点)，状态集合  $Q$  (Trie 树上所有节点)，接受状态集合  $F$  (所有以某个单词作为后缀的节点)，字符集  $\Sigma$  (题目给定) 和转移函数  $\delta$  (类似 KMP 求解)。

AC 自动机全称 Aho-Corasick Automaton，简称 ACAM。它的用途非常广泛，是重要的字符串算法 (8 级)。

### 1.1 算法详解

AC 自动机用于解决 **多模式串** 匹配问题：给定 **字典**  $s$  和文本串  $t$ ，求每个单词  $s_i$  在  $t$  中出现的次数。当然，它的实际应用十分广泛，远超这一基本问题。ACAM 与 KMP 的不同点在于后者仅有一个模式串，而前者有多个。

朴素的基于 KMP 的暴力时间复杂度为  $|t| \times N + \sum |s_i|$ ，其中  $N$  是单词个数。因为进行一次匹配的时间复杂度为  $|s_i| + |t|$ 。当单词数量  $N$  较大时，无法接受。

多串问题自然首先考虑建出字典树。根据其定义，字典树上任意节点  $q \in Q$  与所有单词的某个前缀 **一一对应**。设节点（节点也称状态） $i$  表示的字符串为  $t_i$ 。

借鉴 KMP 算法的思想，我们考虑对于每个状态  $q$ ，求出其 **失配指针**  $fail_q$ 。类似 KMP 的失配数组  $nxt$ ，失配指针的含义为： $q$  所表示字符串  $t_q$  的 **最长真后缀**  $t_{q[j, |t_q|]}$  ( $2 \leq j \leq |t_q| + 1$ )，使得该后缀作为某个单词的前缀出现。这说明  $t_{q[j, |t_q|]}$  恰好对应了字典树上某个状态，因此一个状态的失配指针指向另一个长度比它短的状态。注意，这样的后缀 **可能不存在**，因此失配指针可能指向表示空串的根节点。

从  $q$  向字符串  $fail_q$  连一条有向边，就得到了 ACAM 的 **fail 树**。

- 例如，当  $s = \{b, ab\}$  时， $ab$  会向  $b$  连边，因为  $ab$  最长的（也是唯一的）在  $s_i$  中作为前缀出现的后缀为  $b$ 。
- 再例如，当  $s = \{aba, baba\}$  时， $ab$  会向  $b$  连边， $bab$  会向  $ab$  连边， $aba$  会向  $ba$  连边，而  $baba$  会向  $aba$  连边。对于每一条有向边  $q \rightarrow fail_q$ ，后者是前者的后缀，也是  $s_i$  的前缀。

考虑用类似 KMP 的算法求解失配指针：首先令  $fail_q \leftarrow fail_{faq}$ 。若当前的  $fail_q$  没有  $faq \rightarrow q$  这条（字典树上的）边所表示的字符  $c$  的转移，则令  $fail_q \leftarrow fail_{fail_q}$ ，否则  $fail_q = trans(fail_q, c)$ ，即字典树上在  $fail_q$  处添加字符  $c$  后到达的状态。若  $fail_q$  已经指向根，但还是没找到出边，则  $fail_q$  最终就指向根。

失配指针已经足够强大，但这并不是 AC 自动机的完全体。我们尝试将每个状态的所有字符转移  $\delta(i, c)$  都封闭在状态集合  $Q$  里面。把 KMP 自动机的转移拎出来观察

$$\delta(i, c) = \{i+1 \mid s_{i+1} = c \text{ 或 } i=0 \text{ (which is root)}\} \cup \{fail_i \mid s_i = c \text{ 且 } i \neq 0\}$$

设字典树的根为节点 0，AC 自动机的转移可类似地写为：

$$\delta(i, c) = \begin{cases} trans(i, c) & \text{if } trans(i, c) \text{ exists} \\ 0 & \text{if } trans(i, c) \text{ doesn't exist and } i=0 \text{ (which is root)} \\ fail_i & \text{if } trans(i, c) \text{ doesn't exist and } i \neq 0 \end{cases}$$

$\delta(i, c)$  表示往状态  $i$  后面添加字符  $c$ ，所得字符串的 **最长的** 与  $s_i$  **前缀** 匹配的 **后缀** 所表示的状态。也可理解为从  $i$  开始跳  $fail$  指针，遇到的第一个有字符  $c$  的转移对应转移到的节点：若  $i$  本身有转移，则  $\delta(i, c)$  就等于  $trans(i, c)$ ，否则向上跳一层  $fail$  指针，等于  $\delta(fail_i, c)$ 。

根据已有信息递推，这是 **动态规划** 的核心思想。即求解  $\delta$  函数的过程本质上是一类 DP。

当  $trans(i, c)$  存在时，设其为  $q$ ，则有  $fail_q = \delta(fail_i, c)$ 。因为根据求  $fail_q$  的方法，我们会先令  $fail_q \leftarrow fail_i$ ，然后跳到第一个有字符  $c$  的位置，令  $fail_q$  等于该位置添加  $c$  转移到的状态。这和  $\delta(fail_i, c)$  的定义等价。

有了这一性质，我们就不需要预先求出失配指针，而是在建造 AC 自动机的同时一并求出。由于我们需要保证在计算一个状态的转移时，其失配指针指向的状态的转移已经计算完毕，又因为失配指针长度小于原串长度，故使用 BFS 建立 AC 自动机。一般形式的 AC 自动机代码如下：

cpp

```
int node, son[N][S], fa[N];
void ins(string s) { // 建出 trie 树
```

```

int p = 0;
for(char it : s) {
 if(!son[p][it - 'a']) son[p][it - 'a'] = ++node;
 p = son[p][it - 'a'];
}
}

void build() { // 建出 AC 自动机
 queue<int> q;
 for(int i = 0; i < S; i++) if(son[0][i]) q.push(son[0][i]); // 对于第一层特判，
 因为 fa[0] = 0，此处即转移的第二种情况
 while(!q.empty()) { // 求得的 son[t][i] 就是文章中的转移函数 delta(t, i)，相当于合并了 trie 和 AC 自动机的转移函数
 int t = q.front(); q.pop();
 for(int i = 0; i < S; i++)
 if(son[t][i]) fa[son[t][i]] = son[fa[t]][i], q.push(son[t][i]); // 转移的第一种情况：原 trie 图有 trans(t, i) 的转移
 else son[t][i] = son[fa[t]][i]; // 转移的第三种情况
 }
}

```

特别的，在 ACAM 上会有一些 **终止节点**  $p$ ，代表一个单词或以一个单词结尾，即  $p$  对应的字符串  $tp$  的某个 **后缀** 在字典  $s$  中作为 **单词** 出现。若状态  $p$  本身表示一个单词，即  $tp \in s$ ，则称为 **单词节点**。所有终止节点  $p$  对应着 DFA 的 **接受状态集合**  $F$ ：ACAM 接受且仅接受以给定词典中的某一个单词结尾的字符串。

总结一下我们使用到的约定和定义：

- 节点也被称为 **状态**。
- 设字典树上状态  $i$  所表示的字符串为  $ti$ 。
- **失配指针**  $fail_q$  的含义为  $q$  所表示字符串  $tq$  的最长真后缀  $tq[j, |tq|]$  ( $2 \leq j \leq |tq| + 1$ ) 使得该后缀作为某个单词的前缀出现。
- $\delta(i, c)$  表示往状态  $i$  后添加字符  $c$ ，所得字符串的 **最长的** 与某个单词的 **前缀** 匹配的 **后缀** 所表示的状态。它也是从  $i$  开始，不断跳失配指针直到遇到一个有字符  $c$  转移的状态  $p$ ，添加字符  $c$  后得到的状态  $trans(p, c)$ 。
- **终止节点**  $p$  代表一个单词，或以一个单词结尾。
- 所有终止节点  $p$  组成的集合对应着 DFA 的 **接受状态集合**  $F$ 。
- 若状态  $p$  本身表示一个单词，即  $tp \in s$ ，则称为 **单词节点**。

## 1.2 fail 树的性质与应用

AC 自动机的核心就在于 fail 树。它有非常好的性质，能够帮我们解决很多问题。

- 性质 0：它是一棵 **有根树**，支持树剖，时间戳拍平，求 LCA 等各种树上路径或子树操作。
- 性质 1：对于节点  $p$  及其对应字符串  $tp$ ，对于其子树内部所有节点  $q \in subtree(p)$ ，都有  $tp$  是  $tq$  的后缀，且  $tp$  是  $tq$  的后缀 **当且仅当**  $q \in subtree(p)$ 。根据失配指针的定义易证。
- 性质 2：若  $p$  是终止节点，则  $p$  的子树全部都是终止节点。根据 fail 指针的定义，容易发现对于在 fail 树上具有祖先 - 后代关系的点对  $p, q$ ， $tp$  是  $tq$  的 Border，这意味着  $tp$  是  $tq$  的后缀。因此，若  $tp$  以某个单词结尾，则  $tq$  也一定以该单词结尾，得证。
- 性质 3：定义  $edp$  表示作为  $tp$  后缀的单词数量。若单词互不相同，则  $edp$  等于 fail 树从  $p$  到根节点上单词节点的数量。若单词可以重复，则  $edp$  等于这些单词节点所对应的单词的出现次数之和。

- 常用结论：一个单词在匹配串  $S$  中出现次数之和，等于它在  $S$  的 **所有前缀中作为后缀出现** 的次数之和。

根据性质 3，有这样一类问题：单词有带修权值，多次询问对于某个给定的字符串  $S$ ，所有单词的权值乘以其在  $S$  中出现次数之和。根据常用结论，问题初步转化为 fail 树上带修点权，并对于  $S$  的每个前缀，查询该前缀所表示的状态到根的权值之和。

通常带修链求和要用到树剖，但查询具有特殊性质：一个端点是根。因此，与其单点修改链求和，不如 **子树修改单点查询**。实时维护每个节点的答案，这样修改一个点相当于更新子树，而查询时只需查单点。转化之前的问题需要树剖 + 数据结构  $\log_2$  维护，但转化后即可时间戳拍平 + 树状数组单  $\log$  小常数解决。

补充：对于普通的链求和，只需差分转化为三个到根链求和也可以使用上述技巧。**链加，单点查询** 也可以通过转化变成 **单点加，子树求和**。只要包含一个单点操作，一个链操作，均可以将链操作转化为子树操作，即可将时间复杂度更大的树剖 BIT 换成普通 BIT。

- 性质 4：把字符串  $t$  放在字典  $s$  的 AC 自动机上跑，得到的状态为  $t$  的最长后缀，满足它是  $s$  的前缀。

## 1.3 应用

大部分时候，我们借助 ACAM 刻画多模式串的匹配关系，求出文本串与字典的 **最长匹配后缀**。但 ACAM 也可以和动态规划结合：在利用动态规划思想构建的自动机上进行 DP，这是 **DP 自动机** 算法。

### 1.3.1 结合动态规划

ACAM 除了能够进行字符串匹配，还常与动态规划相结合，因为它精确刻画了文本串与 **所有** 模式串的匹配情况。同时， $\delta$  函数自然地动态规划的转移指明了方向。因此，当遇到形如“**不能出现若干单词**”的字符串 **计数或最优化** 问题，可以考虑在 ACAM 上 DP，将 ACAM 的状态写进 DP 的一个维度。

例如非常经典的 [\[JSOI2007\]文本生成器](#)。题目要求至少包含一个单词，补集转化相当于求 **不包含任何一个单词** 的长为  $m$  的字符串数量。考虑到我们只关心当前字符串的长度，和它与所有单词的匹配情况，设  $f_{i,j}$  表示长为  $i$  且放到所有单词建出的 ACAM 上能够转移到状态  $j$  的字符串数量。转移即枚举下一个字符  $c$  是什么， $f_{i,j} \rightarrow f_{i+1, \delta(j,c)}$ 。根据限制，需要保证  $j$  和  $\delta(j,c)$  都不是终止节点，最终答案即  $26m - \sum_{q \in Q \wedge q \notin F} f_{m,q}$ 。时间复杂度  $O(nm \sum |s_i|)$ 。

### 1.3.2 结合矩阵快速幂

在上一部分的基础上，若  $\sum |s_i|$  很小而转移轮数非常多，可以将转移写成矩阵的形式。 $\delta(p,c)$  为我们提供了转移矩阵：添加一个字符后，从状态  $p$  转移到  $q$  的方案数为  $\sum c[\delta(p,c)=q]$ ，即  $A_{i,j} = \sum c[\delta(i,c)=j]$ 。

具体转移方式视题目而定。矩阵乘法也可以是广义矩阵乘法，如例 *XII*。

## 1.4 注意点

- 建出字典树后不要忘记调用 `build` 建出 ACAM。
- 注意模式串是否可以重复。
- 在构建 ACAM 的过程中，不要忘记递推每个节点需要的信息。如  $edp$  由  $edfa$  和状态  $p$  所表示的单词数量相加得到。

## 1.5 例题

### I. P3808 【模板】AC 自动机 (简单版)

本题相同编号的串多次出现仅算一次，因此题目相当于求：文本串  $t$  在模式串  $s_i$  建出的 ACAM 上匹配时经过的所有节点到根的路径的并上单词节点的个数。

设当前状态为  $p$ ，每次跳  $p$  的失配指针，加上经过节点表示的单词个数（单词可能相同）并标记，直到遇到标记节点  $q$ ，说明  $q$  到根都已经被考虑到。注意上述过程并不改变  $p$  本身。时间复杂度线性。

cpp

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e6 + 5;
const int S = 26;
int n, node, son[N][S], fa[N], ed[N];
string s;
void ins(string s) {
 int p = 0;
 for(char it : s) {
 if(!son[p][it - 'a']) son[p][it - 'a'] = ++node;
 p = son[p][it - 'a'];
 } ed[p]++;
}
void build() {
 queue <int> q;
 for(int i = 0; i < S; i++) if(son[0][i]) q.push(son[0][i]);
 while(!q.empty()) {
 int t = q.front(); q.pop();
 for(int i = 0; i < S; i++)
 if(son[t][i]) fa[son[t][i]] = t, q.push(son[t][i]);
 else son[t][i] = son[fa[t]][i];
 }
}
int main() {
 cin >> n;
 for(int i = 1; i <= n; i++) cin >> s, ins(s);
 int p = 0, ans = 0; cin >> s, build();
 for(char it : s) {
 int tmp = p = son[p][it - 'a'];
 while(ed[tmp] != -1) ans += ed[tmp], ed[tmp] = -1, tmp = fa[tmp];
 } cout << ans << endl;
 return 0;
}
```

### II. [P2292 HNOI2004] L 语言

首先我们有个显然的 DP：设  $f_i$  表示  $i$  前缀能否理解，那么若 **存在**  $f_j=1 \wedge t[j+1,i] \in D$ ，则  $f_i=1$ 。否则  $f_i=0$ 。对  $D$  建出 ACAM，设  $t[1,i]$  跳到了状态  $p$ ，我们只需要知道  $p$  的哪些长度的后缀是单词，这样就可以  $O(|t| |s|)$  回答单次询问，但不够快。

注意到  $|s| \leq 20$ ，因此考虑状压，设  $mskp$ ：若  $p$  的长度为  $l$  的后缀是单词，则  $mskp$  第  $l$  位为 1。这样，再用  $S$  记录  $f_{i-20} \sim f_{i-1}$  的状态，就可以通过位运算快速得到当前  $f_i$  的结果，并更新  $S$ 。

时间复杂度  $O(n|s| |\Sigma| + m|t|)$ ，其中  $|\Sigma|$  表示字符集大小。

### \*III. [P2414 [NOI2011](#)] 阿狸的打字机

由于删去一个字符和添加一个字符对字典树大小的影响均为 1，因此尽管单词长度之和可能很大，但建出的字典树大小仅有  $m$ 。设第  $i$  个单词在 trie 上的节点为  $f_i$ ，根据应用 1，求  $x$  在  $y$  中的出现次数可以在  $y$  到根的每个节点上打标记，查询  $x$  的子树内有标记的节点个数。

因此将询问离线，按  $y$  从小到大的顺序处理询问（为保证修改标记的总次数线性），套上 BIT 即可。时间复杂度线性对数。[代码](#)。

### IV. P5357 [【模板】AC 自动机（二次加强版）](#)

根据 fail 树的性质 1，文本串  $S$  在 AC 自动机上每经过一个节点就将其权值增加 1，则每个单词  $T_i$  在  $S$  中的出现次数即  $T_i$  在 fail 树上的子树节点权值和。时间复杂度线性对数。

### \*V. [P4052 [JSOI2007](#)] 文本生成器

ACAM 与 DP 相结合的例题。

### VI. [P3041 [USACO12JAN](#)] Video Game G

非常套路的 ACAM 上 DP：设  $f_{i,j}$  表示长度为  $i$  且在 ACAM 上转移到状态  $j$  的字符串的最大权值，有转移  $f_{i,j+\delta(j,c)} \rightarrow f_{i+1,\delta(j,c)}$ 。时间复杂度  $O(nk|s| |\Sigma|)$ 。

### \*VII. [CF1202E You Are Given Some Strings...](#)

还算有趣的一道题目。对于同时与两个字符串相关的问题，考虑 **在拼接处计算贡献**，即求出  $f_i$  表示有多少单词是  $t[1,i]$  的后缀， $g_i$  表示有多少单词是  $t[i,n]$  的前缀。 $f_i$  和  $g_i$  都可以用 ACAM 求出。最终答案为  $\sum_{i=2}^{|t|} f_{i-1} g_i$ ，时间复杂度线性。[代码](#)。

### VIII. [CF163E e-Government](#)

裸题。对  $s$  建出 ACAM，根据应用 1，使用性质 3 部分所给出的技巧：单点修改链上求和转化为子树修改单点求和（前提是一个端点为树根），BIT 维护即可。时间复杂度线性对数。[代码](#)。

### \*IX. [P7456 [CERC2018](#)] The ABCD Murderer

由于单词可以重叠（否则就不可做了），我们只需求出对于每个位置  $i$ ，以  $i$  结尾的最长单词的长度  $L_i$ 。因为对于相同的出现位置，用更短的单词去代替最长单词并不会让答案更优。使用 ACAM 即可求出  $L_i$ 。

最优化问题考虑 DP：设  $f_i$  表示拼出  $s[1,i]$  的最小代价。不难得到转移  $f_i = \min_{j=i-L_i-1}^{i-1} f_j$ 。特别的，若  $L_i$  不存在（即没有单词在  $s$  中以  $i$  为结束位置出现）则  $f_i$  为无穷大。若  $f_n$  为无穷大则无解。可以线段树解决。

如果不想写线段树，还有一种方法：从后往前 DP。这样，每个位置可以转移到的地方是固定的（ $i-L_i \sim i-1$ ），所以用小根堆维护，懒惰删除即可。时间复杂度均为线性对数。

### X. [P3121 [USACO15FEB](#)] Censoring G

非常经典的 AC 自动机题目。对  $t$  建出 SAM 加速匹配，每次加入一个字符，用栈在线维护字符串  $s$  即可。时间复杂度线性。



## XI. [P3715 BJOI2017]魔法咒语

二合一屑题。考虑在 ACAM 上 DP，对于前 50% 的数据，由于  $L$  很小，所以可以暴力 DP，时间复杂度  $O(L \times \sum |s_i| \times \sum |t_i|)$ 。对于后 50% 的数据，由于基本词汇长度  $\leq 2$ ，故直接把  $f_i$  和  $f_{i-1}$  放到矩阵里面递推即可。时间复杂度  $O((\sum |t_i|) 3 \log L)$ 。

## XII. CF696D Legen...

非常套路地设  $f_{i,j}$  表示长度为  $i$  且 ACAM 上状态为  $j$  时的最大贡献，令  $ed_i$  表示状态  $i$  所有后缀对应的所有单词权值之和，即不停跳 fail 到达的所有节点权值之和，一个字典树上节点的权值为其所表示的所有单词权值之和。

显然有转移： $f_{i,j} + ed_{\delta(j,c)} \rightarrow f_{i+1, \delta(j,c)}$ ，使用矩阵快速幂优化即可。时间复杂度  $O((\sum |s_i|) 3 \log L)$ 。[代码](#)。

## \*XIII. [P5840 COCI2015]Divljak

由于  $T$  的形态会改变，所以考虑对  $S$  建出 ACAM。根据 fail 树的性质，问题即转化为对给定节点  $p$  ( $tp = S_x$ ) 求存在多少个  $P \in T$  使得  $p$  的子树内存在  $P$  的每个前缀在 ACAM 上匹配到的节点。这相当于在添加  $P$  时，求出其依次匹配到的节点  $q_1, q_2, \dots, q_{|P|}$ ，在 fail 树上对所有  $q_i$  到根的 **链并** 上的所有节点加 1。

上述经典问题可以通过将  $q_i$  按 dfs 序排序后，对  $q_1$  到根执行链加，然后对于每个  $q_i$  ( $i > 1$ )，对  $q_i$  到  $\text{lca}(q_{i-1}, q_i)$  包含  $q_i$  的儿子执行链加。

考虑使用 1.2 提到的技巧，将链加和单点查询转化为单点修改，子树查询，此时只需对所有  $q_i$  加上 1，所有  $\text{lca}(q_{i-1}, q_i)$  ( $i > 1$ ) 减去 1 即可。时间复杂度线性对数。

## 2. 后缀自动机 SAM

后缀自动机全称 Suffix Automaton，简称 SAM，是一类极其有用但难以真正理解的字符串后缀结构（10 级）。它是笔者一年以前学习的算法，现在进行复习并重构学习笔记，看看能不能悟到一些新的东西。

### 2.1 基本定义与引理

SAM 相关的定义非常多，需要牢记并充分理解它们，否则学习 SAM 会非常吃力，因为符号化的语言相较于直观的图片 and 实例更难以理解。

首先，我们给出 SAM 的定义：一个长为  $n$  的字符串  $s$  的 SAM 是一个接受  $s$  的所有 **后缀** 的 **最小** 的有限状态自动机。具体地，SAM 有 **状态集合**  $Q$ ，每个状态是有向无环图上的一个节点。从每个状态出发有若干条或零条 **转移边**，每条转移边都 **对应一个字符**（因此，一条路径表示一个 **字符串**），且从一个状态出发的转移互不相同。根据 DFA 的定义，SAM 还存在 **终止状态集合**  $F$ ，表示从初始状态  $T$  到任意终止状态的任意一条路径与  $s$  的一个 **后缀** 一一对应。

SAM 最重要，也是最基本的一个性质：从  $T$  到任意状态的所有路径与  $s$  的 **所有** 子串 **一一对应**。我们称状态  $p$  表示字符串  $tp$ ，当且仅当存在一条  $T \rightarrow p$  的路径使得该路径所表示的字符串为  $tp$ 。根据上述性质， $tp$  是  $s$  的子串。

- 定义转移边  $p \rightarrow q$  表示的字符为  $cp, q$ 。
- 定义  $\delta(p, c)$  表示状态  $p$  添加字符  $c$  转移到的状态。
- 定义 **前缀** 状态集合  $P$  由所有前缀  $s[1, i]$  对应的状态组成。
- SAM 的有向无环转移图也是有向无环单词图（DAWG, Directed Acyclic Word Graph）。

- $\text{endpos}(t)$ : 字符串  $t$  在  $s$  中所有出现的 **结束位置** 的 **集合**。例如, 当  $s=\text{"abcab"}$  时,  $\text{endpos}(\text{"ab"})=\{2,5\}$ , 因为  $s[1:2]=s[4:5]=\text{"ab"}$ 。
- $\text{substr}(p)$ : **状态**  $p$  所表示的所有子串的 **集合**。
- $\text{shortest}(p)$ : **状态**  $p$  所表示的所有子串中, 长度 **最短** 的那一个子串。
- $\text{longest}(p)$ : **状态**  $p$  所表示的所有子串中, 长度 **最长** 的那一个子串。
- $\text{minlen}(p)$ : **状态**  $p$  所表示的所有子串中, 长度 **最短** 的那一个子串的 **长度**。  
 $\text{minlen}(i)=|\text{shortest}(i)|$ 。
- $\text{len}(i)$ : **状态**  $p$  所表示的所有子串中, 长度 **最长** 的那一个子串的 **长度**。 $\text{len}(i)=|\text{longest}(i)|$ 。

两个字符串  $t_1, t_2$  的  $\text{endpos}$  可能相等。例如当  $s=\text{"abab"}$  时,  $\text{endpos}(\text{"b"})=\text{endpos}(\text{"ab"})$ 。这样, 我们可以将  $s$  的子串划分为若干 **等价类**, 用一个状态表示。SAM 的每个状态对应若干  $\text{endpos}$  集合相同的子串。换句话说,  $\forall t \in \text{substr}(p), \text{endpos}(t)$  相等。因此, SAM 的状态数等于所有子串的等价类个数 (初始状态对应空串)。

读者应该有这样的直观印象: SAM 的每个状态  $p$  都表示一个独一无二的  $\text{endpos}$  等价类, 它对应着在  $s$  中出现位置相同的一些子串  $\text{substr}(p)$ 。  $\text{shortest}(p), \text{longest}(p), \text{minlen}(p)$  和  $\text{len}(p)$  描述了  $\text{substr}(p)$  最短和最长的子串及其长度。

转移边与  $\text{substr}$  的联系: 任意一条  $T \rightarrow p$  的路径  $P$  所表示的字符串  $t_P \in \text{substr}(p)$ 。

在引出 SAM 的核心定义「**后缀链接**」前, 我们需要证明关于上述概念的一些性质。下列引理的内容部分来自 OI-wiki, 相关链接见 Part 2.4.

**引理 1:** 考虑两个非空子串  $u$  和  $w$  (假设  $|u| \leq |w|$ )。要么  $\text{endpos}(u) \cup \text{endpos}(w) = \emptyset$ , 要么  $\text{endpos}(w) \subseteq \text{endpos}(u)$ , 取决于  $u$  是否为  $w$  的一个后缀:

$\{\text{endpos}(w) \subseteq \text{endpos}(u) \mid u \text{ is a suffix of } w\}$   
 $\text{endpos}(u) \cup \text{endpos}(w) = \emptyset \text{ otherwise}$

证明: 若存在位置  $i$  满足  $i \in \text{endpos}(u)$  且  $i \in \text{endpos}(w)$ , 说明  $u$  和  $w$  以  $i$  为结束位置在  $s$  中出现。由于  $|u| \leq |w|$ , 所以  $u$  必然是  $w$  的后缀, 因此  $w$  出现的位置  $u$  必然以  $w$  的后缀形式出现, 即对于任意  $i \in \text{endpos}(w)$  有  $i \in \text{endpos}(u)$ 。否则不存在这样的位置  $i$ , 即  $\text{endpos}(u) \cup \text{endpos}(w) = \emptyset$ 。

**引理 2:** 考虑一个状态  $p$ 。  $p$  所表示的所有子串长度连续, 且 **较短者总是较长者的后缀**。

证明: 根据引理 1, 若两个子串  $\text{endpos}$  相同 (这也说明它们属于相同状态), 则较短者总是较长者的后缀, 后半部分得证。

对于前半部分考虑反证: 假设  $\text{longest}(p)$  长为  $L$  ( $\text{minlen}(p) < L < \text{len}(p)$ ) 的后缀  $t_L \notin \text{substr}(p)$ 。由于  $t_L$  是  $\text{longest}(p)$  的 **真后缀**, 故  $\text{endpos}(\text{longest}(p)) \subseteq \text{endpos}(t_L)$ 。根据假设,  $\text{endpos}(\text{longest}(p)) \neq \text{endpos}(t_L)$ 。又因为  $\text{shortest}(p)$  是  $t_L$  的 **真后缀**, 故  $\text{endpos}(t_L) \subseteq \text{endpos}(\text{shortest}(p))$ , 因此  $|\text{endpos}(\text{longest}(p))| < |\text{endpos}(t_L)| \leq |\text{endpos}(\text{shortest}(p))|$ , 这与  $\text{endpos}(\text{longest}(p)) = \text{endpos}(\text{shortest}(p))$  矛盾, 证毕。

简单地说, 对于一个子串  $t$  的所有后缀, 其  $\text{endpos}$  集合大小随着后缀长度减小而单调不降。这很好理解: **后缀越长, 在  $s$  中出现的位置就越少**。

**推论 1:** 对于子串  $t$  的所有后缀, 其  $\text{endpos}$  集合大小随后缀长度减小而单调不降, 且 **较小的  $\text{endpos}$  集合包含于较大的  $\text{endpos}$  集合**。

引理 2 是非常重要的性质。有了它, 我们就可以定义后缀链接了。



- 定义状态  $p$  的 **后缀链接**  $\text{link}(p)$  指向  $\text{longest}(p)$  **最长** 的一个后缀  $w$  满足  $w \neq \text{substr}(p)$  所在的状态。换句话说，一个后缀链接  $\text{link}(p)$  连接到对应于  $\text{longest}(p)$  最长的处于另一个  $\text{endpos}$  等价类的后缀所在的状态。根据引理 2,  $\text{minlen}(i) = \text{len}(\text{link}(i)) + 1$ 。

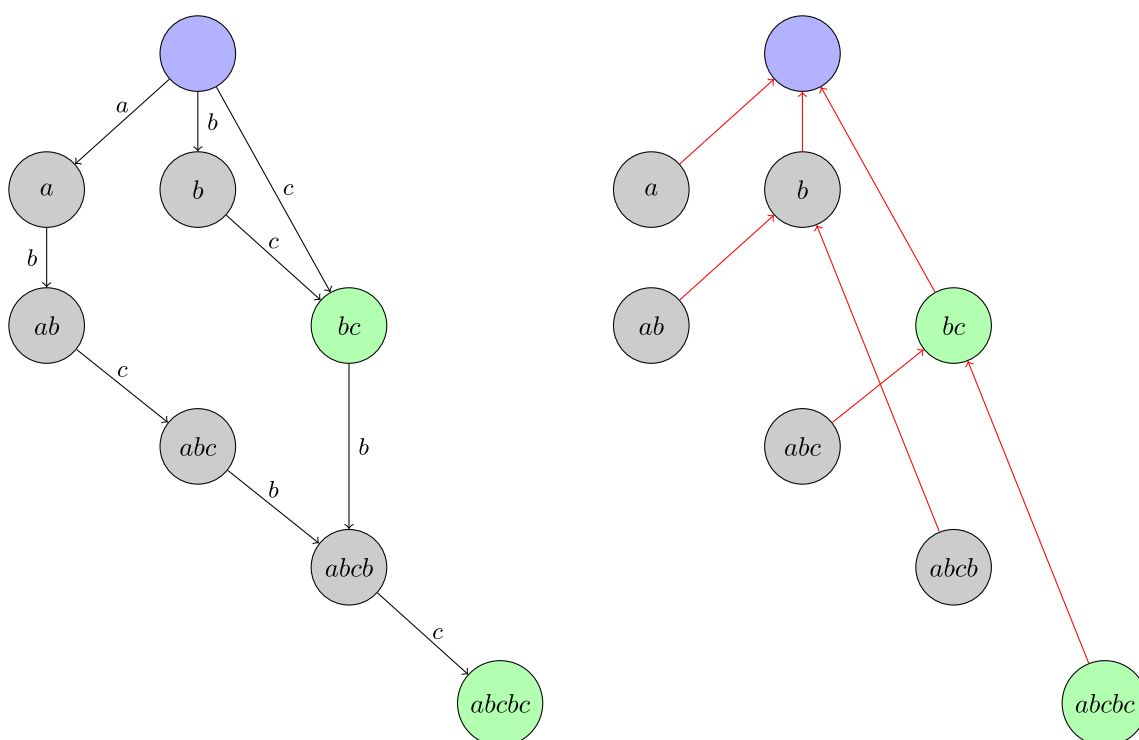
**引理 3:** 所有后缀链接形成一棵以  $T$  为根的树。

证明：对于任意不等于  $T$  的状态，沿着后缀链接移动总能达到一个所表示字符串更短的状态，直到  $T$ 。

- 定义 **后缀路径**  $p \rightarrow q$  表示在后缀链接形成的树上  $p \rightarrow q$  的路径。

**引理 4:** 通过  $\text{endpos}$  集合构造的树（每个子节点的  $\text{subset}$  都包含在父节点的  $\text{subset}$  中）与通过后缀链接  $\text{link}$  构造的树相同。

根据推论 1 与后缀链接的定义容易证明。因此，后缀链接构成的树本质上是  $\text{endpos}$  集合构成的一棵树。



上图图源 OI-wiki。我们给出每个状态的  $\text{endpos}$  集合以便更好理解引理 4:  $\text{endpos}("a") = \{1\}$ ,

$\text{endpos}("ab") = \{2\}$   $\text{endpos}("abcb", "bcb", "cb") = \{4\} \subsetneq \text{endpos}("b") = \{2, 4\}$

$\text{endpos}("abc") = \{3\}$   $\text{endpos}("abcbc", "bcbc", "cbc") = \{5\} \subsetneq \text{endpos}("bc", "c") = \{3, 5\}$

## 2.2 关键结论

我们还需要以下定理确保构建 SAM 的算法的正确性，并使读者对上述定义形成感性的直观的认知。

**结论 1.1:** 从任意状态  $p$  出发跳后缀链接到  $T$  的路径，所有状态  $q \in p \rightarrow T$  的  $[\text{minlen}(q), \text{len}(q)]$  不交，单调递减且并集形成 **连续** 区间  $[0, \text{len}(p)]$ 。

证明：根据后缀链接的性质  $\text{len}(\text{link}(p)) + 1 = \text{minlen}(p)$  即证。

**结论 1.2:** 从任意状态  $p$  出发跳后缀链接到  $T$  的路径, 所有状态  $q \in p \rightarrow T$  的  $\text{substr}(q)$  的并集为  $\text{longest}(p)$  的 **所有后缀**。

证明: 由结论 1.1 和后缀链接的定义易证。

**结论 2.1:**  $\forall tp \in \text{substr}(p)$ , 若存在  $p \rightarrow q$  的 **转移边**, 则  $tp+cp, q \in \text{substr}(q)$ 。

证明: 根据  $\text{substr}$  的定义可得。

**结论 2.2:**  $\forall tq \in \text{substr}(q)$ , 若存在  $p \rightarrow q$  的转移边, 则  $\exists tp \in \text{substr}(p)$  使得  $tp+cp, q = tq$ 。

证明: 结论 2.1 的逆命题。这很好理解, 因为对于任意  $tq \in \text{substr}(q)$ , 若不存在这样的  $tp+cp, q = tq$ , 那么就不存在  $T \rightarrow q$  的路径使得其所表示字符串为  $tp+cp, q$ , 这与  $tq \in \text{substr}(q)$  矛盾。

**结论 3.1:** 考虑状态  $q$ , 不存在转移  $p \rightarrow q$  使得  $\text{len}(p)+1 > \text{len}(q)$ 。

证明: 显然。

**结论 3.2:** 考虑状态  $q$ , **唯一** 存在状态  $p$  和转移  $p \rightarrow q$  使得  $\text{len}(p)+1 = \text{len}(q)$ 。

证明: 考虑反证法, 若不存在这样的  $p$ , 说明  $\forall p, \text{len}(p)+1 < \text{len}(q)$ 。根据结论 2.2,  $\text{substr}(q)$  中最长的一个串的长度为  $\max_{tp \in \text{substr}(p)} |tp|+1$  即  $\max \text{len}(p)+1$ 。根据  $\text{len}$  的定义与  $\text{len}(p)+1 < \text{len}(q)$ , 推得  $\text{len}(q) < \text{len}(q)$ , 矛盾。唯一性不难证明。

简单地说, 若数集  $T$  由若干数集  $S$  的并加上 1 后得到, 那么  $\max_{s \in S} s+1 = \max_{t \in T} t$ 。

**结论 3.3:** 考虑状态  $q$ , **唯一** 存在转移  $p \rightarrow q$  使得  $\text{minlen}(p)+1 = \text{minlen}(q)$ 。

证明: 同理。

- 定义  $\text{maxtrans}(q)$  表示使得  $\text{len}(p)+1 = \text{len}(q)$  且存在转移  $p \rightarrow q$  的唯一的  $p$ 。
- 定义  $\text{mintrans}(q)$  表示使得  $\text{minlen}(p)+1 = \text{minlen}(q)$  且存在转移  $p \rightarrow q$  的唯一的  $p$ 。

**结论 4.1:** 考虑状态  $q$ , 若存在转移  $p \rightarrow q$ , 则  $p$  在后缀链接树上是  $\text{maxtrans}(q)$  或其祖先。

证明: 由于所有  $p$  转移到相同状态  $q$ , 故所有  $p$  的  $\text{substr}(p)$  的并, 短串为长串的后缀。根据 link 树的性质即证。

**结论 4.2:** 考虑状态  $q$ , 若存在转移  $p \rightarrow q$ , 则  $p$  在后缀链接树上是  $\text{mintrans}(q)$  或其子节点。

证明: 同理。

**结论 4.3:** 考虑状态  $q$ , 若存在转移  $p \rightarrow q$ , 则所有这样的  $p$  在 link 树上形成了一条 **深度递减的链**  $\text{maxtrans}(q) \rightarrow \text{mintrans}(q)$ 。

证明: 结合结论 4.1 与结论 4.2 易证。

可以发现上述性质大都与后缀链接有关, 因为后缀链接是 SAM 所提供的最重要的核心信息。我们甚至可以抛弃 SAM 的 DAWG, 仅仅使用后缀链接就可以解决大部分字符串相关问题。

- 扩展定义:  $\text{substr}(p \rightarrow q)$  表示后缀路径  $p \rightarrow q$  上所有状态的  $\text{substr}$  的并。

## 2.3 构建 SAM

铺垫了这么多, 我们终于有足够的性质来建造 SAM 了。之前的长篇大论可能让读者认为它是一个非常复杂的算法: 是, 但不完全是。至少在代码实现方面, 它比同级的 LCT 简单到不知道到哪里去了。

SAM 的构建核心思想是 **增量法**。我们在  $s[1, i-1]$  的 SAM  $A_{i-1}$  的基础上进行更新, 从而得到  $s[1, i]$  的 SAM  $A_i$ 。因此, 该算法是 **在线** 算法。它主要分为三个步骤:

1. 打开 SAM。

2. 把字符插进去。

3. 关上 SAM。

设  $s[1,i-1]$  在  $A_{i-1}$  上的状态为  $las$ ，当前状态数量为  $cnt$ 。 $las$  和  $cnt$  的初始值均为 1，表示初始状态  $T=1$ 。不要忘记初始化  $las$  和  $cnt$ 。

新建初始状态  $cur \leftarrow cnt+1$ ，并令  $cnt$  自增 1 表示状态数量增加 1。 $cur$  即  $s[1,i]$  在  $A_i$  上对应的状态。 $endpos(cur)=\{i\}$ 。令变量  $p \leftarrow las$  防止接下来的操作改变  $las$ 。

接下来我们考虑如何连指向  $cur$  的转移边：由于  $las \rightarrow T$  的后缀路径上的所有状态表示了所有  $s[1,i-1]$  的后缀，因此若  $p$  没有  $s_i$  的转移边，就新建  $p \rightarrow cur$  字符为  $s_i$  的转移，并令  $p \leftarrow link(p)$  表示跳后缀链接。直到遇到路径上第一个有  $s_i$  出边的状态  $p$ ，此时就应该 **停止** 了，因为再连下去  $T \rightarrow p \rightarrow \delta(p, s_i)$  和  $T \rightarrow p \rightarrow cur$  会表示相同字符串，使相同出边指向两个不同节点，与 SAM 的性质相违背。此时需要分三种情况讨论：

---

Case 1：不存在  $p$ 。即后缀路径  $las \rightarrow T$  上的所有状态都没有字符  $s_i$  的转移边。

容易发现这种情况仅在  $s_i$  未在  $s[1:i-1]$  中出现过时发生。我们只需令  $link(cur) \leftarrow T$  即可。

---

Case 2：存在  $p$ ，令  $q = \delta(p, s_i)$  且  $len(p)+1 = len(q)$ 。

令  $link(cur) \leftarrow q$  即可，原因如下：设  $las \rightarrow T$  后缀路径上  $p$  的前一个状态为  $p'$ 。根据操作，可知  $p' \rightarrow cur$  有一条转移边。则此时  $minlen(cur) = minlen(p') + 1 = (len(p)+1) + 1 = len(q) + 1$ ，说明  $q$  恰好与  $cur$  的后缀链接的定义相匹配。

可以证明  $substr(q \rightarrow T)$  是  $s[1,i]$  所有长度  $\leq len(q)$  的后缀：由于  $substr(las \rightarrow T)$  是  $s[1,i-1]$  的所有后缀，又因为  $p$  在  $las \rightarrow T$  上，所以  $longest(p)$  是  $s[1,i-1]$  长为  $len(p)$  的后缀。而  $p \rightarrow q$  存在字符为  $s_i$  的转移边，故  $longest(q)$  是  $s[1,i]$  长为  $len(p)+1 = len(q)$  的后缀。再根据结论 1.2 得证。这同时也证明了  $link(cur) \leftarrow q$  这一操作的正确性。

图源 hihocoder。上图中，在插入  $s_5=a$  时，状态  $p=las=4$  没有字符  $a$  的转移，因此令  $\delta(4,a)=cur=6$ ，然后  $p \leftarrow link(p)=5$ 。状态 5 也没有字符  $a$  的转移，因此令  $\delta(5,a)=6$ ，然后  $p \leftarrow link(p)=T$ ，也就是图中的  $S$ 。

$\delta(T,a)$  存在，此时  $p=T, q=\delta(T,a)=1$ 。因为  $len(T)+1=len(1)$ ，所以令  $link(6) \leftarrow 1$  即可。

注意状态 4,5,6 所表示的子串，可以发现  $(substr(4) \cup substr(5)) + a = substr(6)$ 。这很好地验证了结论 2.1 和结论 2.2。

---

Case 3：存在  $p$ ，令  $q = \delta(p, s_i)$  但  $len(p)+1 \neq len(q)$ 。

此时  $len(p)+1 < len(q)$ ，我们需要将  $q$  **拆成两个状态**  $q_1$  和  $q_2$ ，将  $substr(q)$  分成长度小于等于  $len(p)+1$  和大于  $len(p)+1$  两部分。具体地，先令  $cnt \leftarrow cnt+1$ ，然后新建一个状态  $cl \leftarrow cnt$  表示将  $substr(q)$  长度  $\leq len(p)+1$  的部分丢给  $cl$ ：

- $minlen(cl)$  等于原来的  $minlen(q)$ 。
- $len(cl)$  等于  $len(p)+1$ 。
- 新的  $minlen(q)$  等于  $len(cl)+1$ 。

考虑  $cl$  如何继承  $q$  这一状态：首先， $q$  的所有转移要原封不动地存下来，故对于每个字符  $c$  都要  $\delta(cl, c) \leftarrow \delta(q, c)$ 。此外，由于  $minlen(cl)$  等于原来的  $minlen(q)$ ，因此  $link(cl) \leftarrow$  原来的  $link(q)$ 。同时，新的  $minlen(q)$  等于  $len(cl)+1$  也即  $len(p)+1$ ，所以  $link(q), link(cur) \leftarrow cl$ 。

此外，根据结论 4.3，我们知道后缀路径  $p \rightarrow T$  上转移到  $q$  的状态一定是路径的一段前缀，对于前缀上的所有节点  $p'$ ，我们需要把  $\delta(p', s_i)$  从本来的  $q$  改成  $cl$ ，因为我们把  $\text{substr}(q)$  长度  $\leq \text{len}(p)+1$  的串丢给了状态  $cl$ ，所以对于原本能转移到  $q$  的所有  $\text{len}$  值  $\leq \text{len}(p)$  的状态（显然也是  $p \rightarrow T$  路径的前缀），都需要将字符  $s_i$  的转移 **重定向** 至  $cl$ 。

上图中，我们把  $q=3$  的不大于  $\text{len}(p=T)+1=1$  的所有子串提出来，丢给一个新建的状态  $cl=5$ ，然后  $\text{link}(\text{cur}=4) \leftarrow cl=5$ 。内部  $\text{link}(q=3) \leftarrow cl=5$ ，同时  $\text{link}(cl=5) \leftarrow p=T$ ，即原来的  $\text{link}(q)$ 。

然后，从  $p=T$  往上跳后缀连接直到不存在连向  $q=3$  的路径或到达根节点  $T$ ，表示对于  $p \rightarrow T$  的一段前缀，满足前缀上所有状态添加字符  $s_i$  能够转移到  $q=3$ ，将它们字符为  $s_i$  的转移重定向至  $cl=5$ （当然，上例只有  $T$  一个点，不过并不一定会跳到  $T$ ，因为可能跳到中间的某个状态  $p'$  时就没有转移 ( $p', q=3$ ) 了），即  $(T, 3)$  变为了  $(T, 5)$ 。

上述分类讨论结束后，令  $\text{las} \leftarrow \text{cur}$  表示添加字符  $s_{i+1}$  时  $s[1, i]$  在  $A_i$  对应状态  $\text{cur}$ 。在实现中，我们通常在连接转移边之前执行该操作。构建 SAM 的代码如下：

```
const int N = 1e5 + 5;
const int S = 26;
int cnt = 1, las = 1, son[N][S], fa[N], len[N];
void ins(char s) {
 int it = s - 'a', p = las, cur = ++cnt;
 len[cur] = len[p] + 1, las = cur; // 计算 len[cur], 更新 las
 while(!son[p][it]) son[p][it] = cur, p = fa[p]; // 添加转移边
 if(!p) return fa[cur] = 1, void(); // case 1
 int q = son[p][it];
 if(len[p] + 1 == len[q]) return fa[cur] = q, void(); // case 2
 int cl = ++cnt; cpy(son[cl], son[q], S); // 新建节点, cl 继承 q 的所有转移
 len[cl] = len[p] + 1, fa[cl] = fa[q], fa[q] = fa[cur] = cl; // 计算 len[cl] 以及 cl, q, cur 的后缀链接, 注意 fa[cl] = fa[q] 要在 fa[q] = cl 之前
 while(son[p][it] == q) son[p][it] = cl, p = fa[p]; // 修改后缀路径 p -> T 的一段前缀
}
```

当字符集  $\Sigma$  非常大的时候，时空复杂度均无法接受，因此需要使用平衡树维护每个状态的所有转移边，可以用 `map` 代替。

## 2.4 时间复杂度证明

下设字符串  $s$  长度为  $n$ ，证明大部分摘自 OI wiki。

### 2.4.1 状态数上界

构建后缀自动机的算法本身就已经证明了其 SAM 状态数不超过  $2n-1$ ：插入  $s_1, s_2$  时分别产生一个状态，后续插入每个  $s_i$  时最多产生两个状态，因此当  $n>1$  时状态数不超过  $2n-2$ ，形如  $abb\dots bb$  的字符串达到上界。当  $n=1$  时状态数为  $2n-1$ 。

### 2.4.2 转移数上界

称  $\text{len}(p)+1=\text{len}(q)$  的转移  $(p, q)$  为连续的，显然，从一个非终止状态  $p$  出发 **有且仅有一条** 连续转移  $(p, q)$ ，对于  $q$  也有且仅有一个对应的  $p$ 。因此，连续转移总数不超过  $2n-2$ 。对于不连续的转移，找到从根节点  $T \rightarrow p$  的一条连续路径，设其所表示字符串为  $u$ ；找到从  $q$  到任意一个终止节点  $f \in F$  的一条连续路径，设其所表示字符串为  $v$ 。对于不同的  $p, q$ ， $sp, q=u+cp, q+v$  互不相同：若两个转移  $(p, q)$  和  $(p', q')$  出现  $sp, q=sp', q'$  的情况，由于不同路径所表示字符串不同，因此  $(p, q)$  和  $(p', q')$  在同一条路径，这与

$T \rightarrow p$  和  $q \rightarrow F$  连续矛盾。又因为  $sp, q$  是  $s$  的真后缀 ( $s$  对应的路径转移显然连续)，因此不连续的转移数量不超过  $n-1$ 。这样，我们得到了转移数上界  $3n-3$ 。

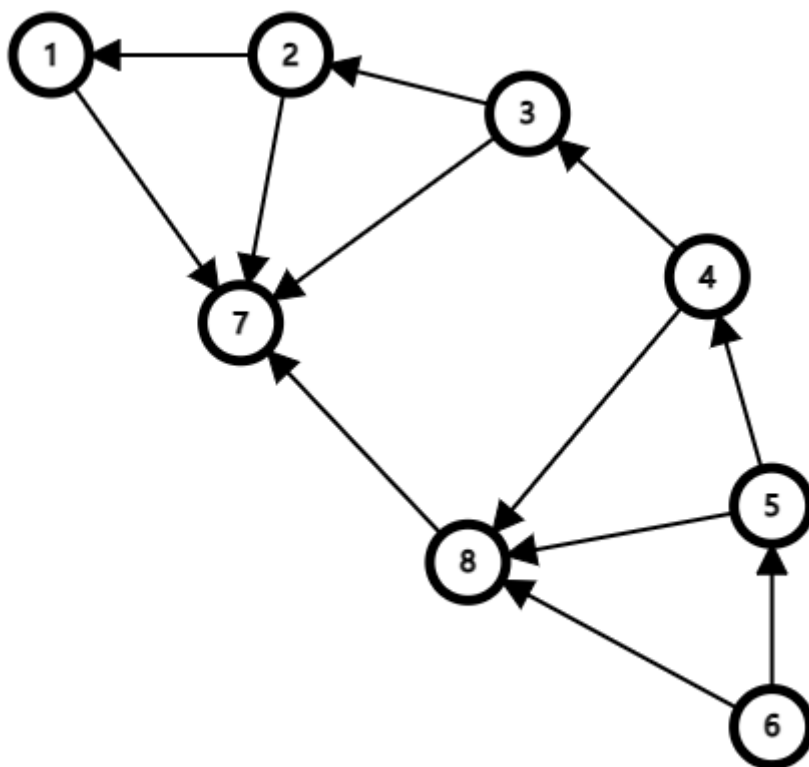
由于最大的状态数量仅在形如  $abb \cdots bb$  的字符串中达到，此时转移数量小于  $3n-3$ 。形如  $abb \cdots bbc$  的字符串达到了  $3n-4$  的上界。

### 2.4.3 操作次数上界

该部分 OI Wiki 上讲得较为简略，因此笔者自行证明了这一结论。在构建 SAM 的过程中，有且仅有将  $p \rightarrow q$  的转移边改为  $p \rightarrow cl$  的操作 **不新建** 转移边。因此，基于 **转移数线性** 这一结论，其它操作的时间复杂度均为线性。

定义  $\text{depth}(p)$  表示  $p$  在 link 树上的 **深度**。引理：若  $p \rightarrow q$  存在转移边，则  $\text{depth}(p) \geq \text{depth}(q)$ 。证明：

- 考虑后缀路径  $q \rightarrow T$  上的任意两个不同状态  $q_1, q_2$  ( $q_1 \neq q_2$ )。设  $p_1$  为任意能转移到  $q_1$  的状态， $p_2$  为任意能转移到  $q_2$  的状态。因为  $\text{substr}(q_1), \text{substr}(q_2)$  均为  $\text{longest}(q)$  的后缀，因此  $\text{substr}(p_1), \text{substr}(p_2)$  均为  $\text{longest}(p)$  的后缀。所以  $p_1, p_2$  均在后缀路径  $p \rightarrow T$  上。
- 若  $p_1 = p_2$ ，则  $p_1$  通过同一字符能转移到不同状态，矛盾。因此  $p_1 \neq p_2$ 。故能转移到  $q \rightarrow T$  上任意状态  $q'$  的所有状态  $p'$  均在  $p \rightarrow T$  上且 **互不相同**。由于对于每个  $q'$  至少存在一个与之对应的  $p'$  (可能存在多个)，因此  $|q \rightarrow T| \leq |p \rightarrow T|$ ，即  $\text{depth}(p) \geq \text{depth}(q)$ 。证毕。
- 可结合下图以更好理解，其中  $i \rightarrow i-1$  的边表示一条后缀链接，其余边表示转移边。



假设我们从  $p$  一直跳到  $p'$ ，并将  $p \rightarrow p'$  路径上所有状态指向  $q$  的转移边改为指向  $cl$ 。设  $q' = \delta(\text{link}(p'), si)$ ，容易证明 **原**  $\text{link}(q)$  即 **现**  $\text{link}(cl) = q'$ 。设  $d = \text{depth}(p) - \text{depth}(p')$ ，即从  $p$  开始跳  $\text{link}$  的次数。根据上述引理，我们有  $\text{depth}(q') \leq \text{depth}(p') = \text{depth}(p) - d \leq \text{depth}(las) - 1 - d$ 。

同时，根据  $\text{link}(cur) = cl$ ， $\text{link}(cl) = q'$  可知  $\text{depth}(cur) - 2 \leq \text{depth}(las) - 1 - d$ ，即  $d \leq \text{depth}(las) - \text{depth}(cur) + 1$ ，这一不等式通过精确分析还可以更紧。因此，该部分操作的总时间复杂度可用  $cur$  相对于  $las$  的 **深度减少量之和** 来估计。同时，若进入 Case 1 或 Case 2，则因为  $las \rightarrow cur$  存在转移边，由引理得  $\text{depth}(cur) \leq \text{depth}(las)$ ，若进入 Case 3，则根据上述不等式有  $\text{depth}(cur) \leq \text{depth}(las) + 1$ 。因此，势能分析得到  $\sum d$  的级别为线性。

## 2.5 应用

### 2.5.1 求本质不同子串个数

根据 SAM 的性质，每个子串唯一对应一个状态，因此答案即  $\sum \text{len}(i) - \text{len}(\text{link}(i))$ 。

### 2.5.2 字符串匹配

用文本串  $t$  在  $s$  的 SAM 上跑匹配时，我们可以得到对于  $t$  的每个前缀  $t[1,i]$ ，其作为  $s$  的子串出现的最长后缀  $Li$ ：若当前状态  $p$ （即  $t[i-Li-1, i-1]$  所表示的状态）不能匹配  $t_i$ （即  $\delta(p, t_i)$  不存在），就跳后缀链接令  $p \leftarrow \text{link}(p)$  并实时更新  $Li = \text{len}(p)$  直到  $p = T$  或  $\delta(p, t_i)$  存在，对于后者令  $p \leftarrow \delta(p, t_i)$ ， $Li$  还需再加上 1。若能匹配，则直接令  $p \leftarrow \delta(p, t_i)$  并令  $Li \leftarrow Li + 1$ 。综合一下，我们得到如下代码：

cpp

```
for(int i = 1, p = 1, L = 0; i <= n; i++) {
 while(p > 1 && !son[p][t[i] - 'a']) L = len[p = fa[p]];
 if(son[p][t[i] - 'a']) L = min(L + 1, len[p = son[p][t[i] - 'a']]);
}
```

## 2.6 广义 SAM

广义 SAM，GSAM，全称 General Suffix Automaton，相对于普通 SAM 它支持对多个字符串进行处理。它可以看做对 trie 建后缀自动机。

一般的写法是每插入一个字符串前将  $las$  指针置为  $T$ ，非常方便。一个细节：构建单串 SAM 时， $\delta(las, s_i)$  一定不存在，但对于多串 SAM 可能存在。这说明当前字符串  $s$  的  $i$  前缀是某个已经添加过的字符串的子串。我们需要进行以下特判，否则会出现这种情况：<https://www.luogu.com.cn/discuss/322224>。

1. 当  $q = \delta(las, s_i)$  存在，且  $\text{len}(las) + 1 = \text{len}(q)$  时，令  $las \leftarrow q$  并直接返回。
2. 当  $q = \delta(las, s_i)$  存在，且  $\text{len}(las) + 1 \neq \text{len}(q)$  时，我们会新建节点  $cl$ ，并进行复制。此时，令  $las \leftarrow cl$  而非  $cur$ 。这是因为  $\text{len}(cur) = \text{len}(las) + 1$  且  $\text{len}(cl) = \text{len}(las) + 1$ ，又因为  $\text{link}(cur) = cl$ ，所以这说明  $\text{substr}(cur) = \emptyset$ ，即 **节点  $cur$  是空壳，真正的信息在  $cl$  上面**。为此，我们舍弃掉这个  $cur$ ，并用  $cl$  代替它。

cpp

```
int ins(int p, int it) {
 if(son[p][it] && len[son[p][it]] == len[p] + 1) return son[p][it]; // 如果节点
 已经存在，且 len 值相对应，即 (p, son[p][it]) 是连续转移，则直接转移。
 int cur = ++cnt, chk = son[p][it]; len[cur] = len[p] + 1;
 while(!son[p][it]) son[p][it] = cur, p = fa[p];
 if(!p) return fa[cur] = 1, cur;
 int q = son[p][it];
 if(len[p] + 1 == len[q]) return fa[cur] = q, cur;
 int cl = ++cnt; cpy(son[cl], son[q], S);
 len[cl] = len[p] + 1, fa[cl] = fa[q], fa[q] = fa[cur] = cl;
 while(son[p][it] == q) son[p][it] = cl, p = fa[p];
 return chk ? cl : cur; // 如果 len[las][it] 存在，则 cur 是空壳，返回 cl 即可
}
```

上述方法本质相当于对匹配串建出 trie 后进行 **dfs** 构建 SAM。部分特殊题目会直接给出 trie 而非模板串，此时模板串长度之和的级别为  $O(|S|^2)$ ，因此只能 **bfs** 构建 SAM：设  $P_p$  表示 trie 树上状态  $p$  在 SAM 上对应的位置，若 **trie 树**  $T$  上的转移  $q = \delta_T(p, c)$  存在，其中  $c$  是  $p \rightarrow q$  所表示字符，那么以  $P_p$  作为  $las$ ，插入字符  $c$  后新的  $las$  即  $P_q$ 。此时 **不需要** 像上面一样特判，因为  $\delta(P_p, c)$  必然不存在，这是由于 bfs 使得  $len(P_p)$  **单调不降**。模板题 [P6139](#) 代码：

cpp

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define cpy(x, y, s) memcpy(x, y, sizeof(x[0]) * (s))

const int N = 2e6 + 5;
const int S = 26;

ll n, ans, cnt = 1;
string s;
int len[N], fa[N], son[N][S];
int ins(int p, int it) {
 int cur = ++cnt; len[cur] = len[p] + 1;
 while(!son[p][it]) son[p][it] = cur, p = fa[p];
 if(!p) return fa[cur] = 1, cur;
 int q = son[p][it];
 if(len[p] + 1 == len[q]) return fa[cur] = q, cur;
 int cl = ++cnt; cpy(son[cl], son[q], S);
 len[cl] = len[p] + 1, fa[cl] = fa[q], fa[q] = fa[cur] = cl;
 while(son[p][it] == q) son[p][it] = cl, p = fa[p];
 return cur;
}

int node = 1, pos[N], tr[N][S];
void ins(string s) {
 int p = 1;
 for(char it : s) {
 if(!tr[p][it - 'a']) tr[p][it - 'a'] = ++node;
 p = tr[p][it - 'a'];
 }
}

void build() {
 queue<int> q; q.push(pos[1] = 1);
 while(!q.empty()) {
 int t = q.front(); q.pop();
 for(int i = 0, p; i < S; i++) if(p = tr[t][i])
 pos[p] = ins(pos[t], i), q.push(p);
 }
}

int main() {
 cin >> n;
 for(int i = 1; i <= n; i++) cin >> s, ins(s);
 build();
 for(int i = 2; i <= cnt; i++) ans += len[i] - len[fa[i]];
 cout << ans << endl;
 return 0;
}
```

```
}
```

## 2.7 常用技巧与结论

### 2.7.1 线段树合并维护 endpos 集合

对于部分题目，我们需要维护每个状态的 endpos 集合，以刻画每个子串在字符串中所有出现位置的信息。

为此，我们在  $s[1,i]$  对应状态的 endpos 集合里插入位置  $i$ ，再根据 endpos 集合构造出来的树本质上就是后缀链接树这一事实，在 link 树上进行 **线段树合并** 即可得到每个状态的 endpos 集合。这是一个非常有用且常见的技巧。

注意，线段树合并时会破坏原有线段树的结构，因此若需要在线段树合并后保留每个状态的 endpos 集合对应的线段树的结构，需要在线段树合并时 **新建节点**。即可 **持久化线段树合并**。SAM 相关问题的线段树合并通常均需要可持久化。

特别的，如果仅为了得到 endpos 集合大小，那么只需求出每个状态在 link 树上的子树有多少个表示  $s$  的前缀的状态。前缀状态即所有曾作为  $cur$  的节点。对此，有两种解决方法：直接建图 dfs，以及 ——

### 2.7.2 桶排确定 dfs 顺序

显然后缀链接树上父亲的  $len$  值一定小于儿子，但千万不能认为编号小的节点  $len$  值也小。因此，对所有节点按照  $len$  值从大到小进行桶排序，然后按顺序合并每个状态及其父亲是正确的，并且常数比建图 + dfs 小不少，代码见例题 1。

### 2.7.3 快速定位子串

给定区间  $[l,r]$ ，求  $s[l,r]$  在 SAM 上的对应状态：在构建 SAM 时容易预处理  $s[1,i]$  所表示的状态  $posi$ 。从  $posr$  开始在 link 树上倍增找到最浅的， $len$  值  $\geq r-l+1$  的状态  $p$  即为所求。

### 2.7.4 其它结论

1. 在 link 树上，若  $p$  是  $q$  的祖先，则  $\text{substr}(p)$  中所有字符串在  $\text{longest}(q)$ （下记为  $s$ ）中出现次数与出现位置相同。具体证明见 [CF700E 题解区](#)。

## 2.8 注意点总结

- 做题时不要忘记初始化  $las$  和  $cnt$ 。
- 第二个 `while` 不要写成 `son[p][it] = cur`，应为 `son[p][it] = cl`。
- SAM 开两倍空间。
- 对于多串 SAM，如果每插入一个新字符串时令  $las \leftarrow T$ ，且插入字符时不特判  $\delta(las, si)$  是否存在，会导致出现空状态，从而父节点的  $len$  值 **不一定严格小于** 子节点，使得桶排失效。对此要格外注意。

## 2.9 例题

### I. [P3804 【模板】后缀自动机 \(SAM\)](#)

对  $s$  建出 SAM，对于每个状态  $p$  求出其 endpos 集合大小。根据题目限制，答案即  $\sum |\text{endpos}(p)| \geq 2len(p) \times |\text{endpos}(p)|$ 。视字符集大小为常数，时间复杂度线性。

cpp

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



```

using namespace std;

#define ll long long
#define cpy(x, y, s) memcpy(x, y, sizeof(x[0]) * (s))

const int N = 2e6 + 5; // 不要忘记开两倍空间
const int S = 26;

char s[N];
int cnt = 1, las = 1;
int son[N][S], len[N], fa[N];
int ed[N], buc[N], id[N];
ll n, ans;
void ins(char s) {
 int it = s - 'a', cur = ++cnt, p = las;
 las = cur, len[cur] = len[p] + 1, ed[cur] = 1;
 while(!son[p][it]) son[p][it] = cur, p = fa[p];
 if(!p) return fa[cur] = 1, void();
 int q = son[p][it];
 if(len[p] + 1 == len[q]) return fa[cur] = q, void();
 int cl = ++cnt; cpy(son[cl], son[q], S);
 len[cl] = len[p] + 1, fa[cl] = fa[q], fa[q] = fa[cur] = cl;
 while(son[p][it] == q) son[p][it] = cur, p = fa[p];
}
int main() {
 scanf("%s", s + 1), n = strlen(s + 1);
 for(int i = 1; i <= n; i++) ins(s[i]);
 for(int i = 1; i <= cnt; i++) buc[len[i]]++;
 for(int i = 1; i <= n; i++) buc[i] += buc[i - 1];
 for(int i = cnt; i; i--) id[buc[len[i]]--] = i;
 for(int i = cnt; i; i--) ed[fa[id[i]]] += ed[id[i]];
 for(int i = 1; i <= cnt; i++) if(ed[i] > 1) ans = max(ans, 1ll * ed[i] *
len[i]);
 cout << ans << endl;
 return 0;
}

```

## II. [P4070 [SDOI2016](#)]生成魔咒

非常裸的 SAM，插入每个字符后新增的子串个数为  $\text{len}(\text{cur}) - \text{len}(\text{link}(\text{cur}))$ ，求和即可。由于字符集太大，需要使用 map 存转移数组。时间复杂度线性对数。

## \*III. [P4022 [CTSC2012](#)]熟悉的文章

首先二分答案  $m$ ，考虑设  $f_i$  表示文章的  $i$  前缀最长的符合限制的匹配长度。根据应用 2.5.2 我们可以求出文章的每个前缀作为字典子串出现的最长后缀长度  $L_i$ ，则  $f_i = \max_{j \in [i-L_i, i-m]} f_j + (i-j)$ 。显然， $L_i \leq L_{i-1} + 1$ ，因此  $i - L_i$  单调不降，故可以使用单调队列优化。时间复杂度线性对数。

## IV. [P5546 [POI2000](#)]公共串

建出 GSAM 后，设  $\text{mski}$  表示  $\text{substr}(i)$  在哪些串中出现过，以状压形式存储，直接在 link 树上合并即可。

## V. [P3346 ZJOI2015]诸神眷顾的幻想乡

由于叶子节点仅有 20 个，因此从每个叶子节点开始，整棵树都会形成一个字典树。将这 20 棵 Trie 树拼在一起求 GSAM 就做完了。

## VI. [P3181 HAOI2016]找相同字符

建出两个串的 GSAM，设  $ed1,i$  表示状态  $i$  关于  $s1$  的  $endpos$  集合大小， $ed2,i$  同理。答案显然为  $\sum ed1,i \times ed2,i \times (\text{len}(i) - \text{len}(\text{link}(i)))$ 。

## VII. [P5341 TJOI2019]甲苯先生和大中锋的字符串

建出  $s$  的 SAM 后容易得到所有出现  $k$  次的子串状态。每个符合题意的状态的字串长度是一段区间，差分即可。时间复杂度线性。

## VIII. [P4341 BJWC2010]外星联络

SAM 的转移函数刻画了一个字符串  $s$  的所有子串，因此直接在该 DAG 上贪心遍历即可。贪心指优先走字符小的出边。

## \*IX. [P3975 TJOI2015]弦论

根据一条路径表示一个子串的性质，考虑求出从每个节点开始的路径条数  $di = 1 + \sum \delta(i,c) d\delta(i,c)$  帮助跳过不可能的分支，然后在 SAM 的 DAG 上模拟跑一遍即可。对于  $t=1$  只需将上式中的 1 改为  $edi$ 。

## \*X. H1079 退群杯 3rd E.

给定字符串  $s$ ，多次询问求  $sc \sim d$  有多少个子串包含  $sa \sim b$ 。  $|s|, q \leq 2 \times 10^5$ 。

设  $L = b - a + 1$ 。我们对每个位置  $p \in [c + L - 1, d]$ ，求出有多少个左端点  $l \geq c$  使得  $sl \sim p$  包含  $sa \sim b$ 。考虑找到  $p$  前面  $sa \sim b$  的最后一次出现位置  $q$ ，则贡献显然为  $\max(0, (q - L + 1) - c + 1)$ 。

转化贡献形式，考虑每个落在  $[c + L - 1, d]$  的  $sa \sim b$  的出现位置  $q$  对答案的贡献。为方便说明，我们不妨假设  $sa \sim b$  在  $d + 1$  处出现。考虑  $sa \sim b$  在  $q$  之后的下一次出现  $q'$ ，则对于  $p \in [q, q' - 1]$ ，贡献均为  $(q - L + 1) - c + 1$ 。注意到  $2 - c - L$  均与询问有关，与  $q$  无关，因此提出。则贡献可写为  $q \times (q' - q)$ 。即每个位置的下标值乘以和下一次出现之间的距离。线段树维护区间出现位置最小值，最大值即可维护该信息。

$2 - c - L$  的贡献次数为  $d - (\min q) + 1$ ，因为所有  $[q, q' - 1]$  的区间并起来形成了区间  $[\min q, d]$ 。对  $endpos$  集合 **可持久化** 线段树合并，再使用 2.7.3 的技巧，即可做到  $\log$  时间内回答每个询问。时间复杂度线性对数。代码。

## XI. CF316G3 Good Substrings

对所有串建出 GSAM，求出每个状态所表示的串在  $s$  和每个模式串中出现了多少次，若合法则统计答案即可。时间复杂度线性。

如果用先建出字典树再建 GSAM 的方法，空间开销会比较大，需要用 `unsigned short` 卡空间。

## XII. SP8222 NSUBSTR - Substrings

这就属于 SAM 超级无敌大水题了吧。

### XIII. 某模拟赛 一切的开始

给定字符串  $s$ ，求其两个 **不相交** 子串的长度乘积最大值，满足其中一个子串为另一个子串的子串。 $|s| \leq 105$ 。

对  $s$  建出 SAM，对于每个状态  $i$ ，我们只关心其第一次出现  $a$  和最后一次出现的位置  $b$ ，因为这样最优，反证法可证。若前者是后者的子串，那么后者显然取满  $[a+1, n]$ ，前者长度即  $L = \min(\text{len}(i), b-a)$ 。若后者是前者的子串，则后者一定尽量长，长度为  $L$ ，那么前者取满  $[1, b-L]$  最优，长度即  $b-L$ 。

综上，答案即  $\max_i L \times \max(n-a, b-L)$ 。时间复杂度线性。

### \*XIV. [CF1037H Security](#)

考虑直接在后缀自动机的 DAWG 上贪心。使用线段树合并判断当前字符串是否作为  $[l, r]$  的子串出现过，时间复杂度  $O(|\Sigma| n \log n)$ 。[代码](#)。

### \*XV. [CF700E Cool Slogans](#)

容易发现  $s_{i-1}$  在  $s_i$  中一定同时以前缀和后缀的形式出现，否则调整法证明可以做到更优。我们使用  $s_{i-1}$  在  $s_i$  中作为后缀的性质，考虑直接在 link 树上 DP。

再根据 2.7.4 的结论一（实际上这个结论是笔者做本题时才遇到的），我们可以设  $fp$  表示  $\text{longest}(p)$  的答案，以及  $gp$  表示  $p$  的祖先中答案取到  $fp$  的深度最小的状态，因为我们要让串长尽可能小，这样出现次数更多。转移即检查  $\text{longest}(\text{glink}(p))$  在  $\text{longest}(p)$  中是否出现了至少两次，这相当于检查  $\text{longest}(\text{glink}(p))$  是否在  $\text{longest}(p)$  的某个出现位置  $pos$  之前的一段区间  $[pos - \text{len}(p) + \text{len}(\text{glink}(p)), pos - 1]$  处出现，容易用线段树合并维护  $\text{endpos}$  集合做到。若是，则令  $fp = \text{flink}(p) + 1$ ， $gp = p$ 。否则  $fp = \text{flink}(p)$ ， $gp = \text{glink}(p)$ 。

$\max fp$  即为答案，时空复杂度线性对数。[代码](#)。

### \*XVI. [CF666E Forensic Examination](#)

SAM 各种常用技巧结合版。首先对  $s$  和  $t_i$  一并建出 GSAM，线段树维护每个节点对应的子串在每个  $t_i$  中出现的次数，即线段树  $T_p$  的位置  $i$  上记录着  $p$  所表示的所有串在  $t_i$  中的出现次数。由于题目还需求最小编号，所以线段树维护区间最大出现次数以及对应最小编号。

使用线段树合并，预处理 link 的倍增数组以快速定位子串，单次询问只需倍增到  $s[pl, pr]$  的对应状态  $p$ ，查询  $T_p$  上  $[l, r]$  的信息即可。时空复杂度均为线性对数。[代码](#)。

## 2.10 相关链接与资料

- [OI wiki: 后缀自动机 \(SAM\)](#)。
- [hihoCoder: 后缀自动机一](#)。
- [hihoCoder: 后缀自动机二](#)。
- [Linshey: 对 SAM 和 PAM 的一点理解](#)。
- [洛谷题单: SA & SAM](#)。
- [辰星凌: 题解 P6139 【模板】广义后缀自动机 \(广义SAM\)](#)。

## 3. 回文自动机 PAM

省选前两周填坑。之所以不是省选之后是因为担心省选考这玩意。