# 2017 ACM-ICPC 全国邀请赛(陕西)

## 标准算法模板库


西安交通大学  电子与信息工程学院

计算机科学与技术系

刘海松

# 图论

- 存储结构
  - 链式向前星

- 单源最短路
  - Bellman-Fordmaxm
  - Dijkstra
  - SPFA

- 最小生成树
  - Kruskal
  - Prim

- 并查集
- 拓扑排序
- 网络流
  - Edmonds-Karp
  - Dinic
  - ISAP

- 强连通分量
  - Tarjan

- 欧拉路

# 存储结构 - 链式向前星

定义：

```
#define maxn 10005
#define maxm 100005

struct edge {
    int to, next, val;
    edge(int t = 0, int n = 0, int v = 0):
        to(t), next(n), val(v) {}
}g[maxm * 2];

int np, ne, gsize, head[maxn];
```

添加边：

```
inline void add_edge(int from, int to, int val) {
    g[gsize] = edge(to, head[from], val);
    head[from] = gsize++;
}
```

初始化：

```
memset(head, -1, sizeof head);
```

访问以 `p` 为起点的所有边：

```
for (int i = head[p]; ~i; i = g[i].next)
```

# 最短路 - Bellman-Ford

数据结构使用链式向前星，时间复杂度：$O(NE)$，可以检测负环。

```
int ne, np, ps, pe, sp[maxn], gsize, head[maxn];

int bellman_ford() {
    memset(sp, inf, sizeof sp); sp[ps] = 0;
    for (int i = 1; i < np; i++)
        for (int j = 0; j < np; j++)
            for (int k = head[j]; ~k; k = g[k].next)
                if (sp[g[k].to] > sp[j] + g[k].val)
                    sp[g[k].to] = sp[j] + g[k].val;

    for (int j = 0; j < np; j++)
        for (int k = head[j]; ~k; k = g[k].next)
            if (sp[g[k].to] > sp[j] + g[k].val) {
                printf("Found negative weight cycle.\n");
                return -1;
            }
    return sp[pe];
}
```

# 最短路 - Dijkstra

数据结构使用链式向前星，时间复杂度：$O(ElogN)$

```
int np, ne, ps, gsize, head[maxn], sp[maxn];
typedef pair<int, int> node;

void dijkstra() {
    priority_queue<node, vector<node>, greater<node> > q;
    memset(sp, inf, sizeof sp); sp[ps] = 0;
    q.push(node(0, ps));
    while (!q.empty()) {
        node p = q.top(); q.pop();
        if (sp[p.snd] < p.fst) continue;
        for (int cnt = p.snd, i = head[p.snd]; ~i; i =
g[i].next)
            if (sp[g[i].to] > sp[cnt] + g[i].cost) {
                sp[g[i].to] = sp[cnt] + g[i].cost;
                q.push(node(sp[g[i].to], g[i].to));
            }
    }
}
```

## 最短路 - SPFA

数据结构使用链式向前星，时间复杂度：未知

```
int ps, pe, np, ne, sp[maxn], gsize, head[maxn];
bool inq[maxn];

void SPFA() {
    queue<int> q;
    memset(sp, inf, sizeof sp); sp[ps] = 0;
    q.push(ps); inq[ps] = 1;
    while (!q.empty()) {
        int now = q.front();
        for (int i = head[now]; ~i; i = g[i].next)
            if (sp[g[i].to] > sp[now] + g[i].val) {
                sp[g[i].to] = sp[now] + g[i].val;
                if (!inq[g[i].to]) {
                    q.push(g[i].to); inq[g[i].to] = 1;
                }
            }
        q.pop(); inq[now] = 0;
    }
}
```

# 最小生成树 - Kurskal

时间复杂度：$O(ElogE)$，并查集已加入状态压缩和 `rank` 优化。

```
struct edge {
    int x, y, w;
}g[maxm];

int np, ne, par[maxn], rk[maxn];

bool cmp(edge& a, edge& b) { return a.w < b.w; }

inline int getfa(int x) {
    int fx = x, tmp;
    while (fx != par[fx]) fx = par[fx];
    while (x != fx) {//compress condition
        tmp = par[x];
        par[x] = fx;
        x = tmp;
    }
    return fx;
}

inline void combine(int x, int y) {
    if (rk[x = getfa(x)] > rk[y = getfa(y)])
        par[y] = x;
    else {
        par[x] = y;
        if (rk[x] == rk[y]) rk[y]++;
    }
}

int main() {
    scanf("%d %d", &np, &ne);
    for (int i = 0; i < np; i++) par[i] = i;
```

```
    for (int i = 0; i < ne; i++)
        scanf("%d %d %d", &g[i].x, &g[i].y, &g[i].w);

    sort(g, g + ne, cmp);
    int cnt = 0, all_cost = 0;
    for (int i = 0; i < ne && cnt < np - 1; i++) {
        int fx = getfa(g[i].x), fy = getfa(g[i].y);
        if (fx != fy) {//Add edge (xp, yp)
            combine(fx, fy);
            cnt++;
            all_cost += g[i].w;
        }
    }
    printf("%d\n", all_cost);
    return 0;
}
```

## 最小生成树 - Prim

数据结构使用链式向前星，时间复杂度：$O(ElogN)$

```
typedef pair<int, int> node;
int np, ne, gsize, head[maxn], minc[maxn];
bool vis[maxn];

int prim() {
    int all_cost = 0;
    memset(minc, INF, sizeof minc); minc[1] = 0;
    priority_queue<node, vector<node>, greater<node> > q;
    q.push(node(0, 1));//start from point 1
    while (!q.empty()) {
        node now = q.top(); q.pop();
        if (vis[now.snd] || now.fst > minc[now.snd]) continue;
        vis[now.snd] = 1; all_cost += minc[now.snd];
        for (int i = head[now.snd]; ~i; i = g[i].next)
            if (minc[g[i].to] > g[i].val) {
                minc[g[i].to] = g[i].val;
                q.push(node(g[i].val, g[i].to));
            }
    }
    return all_cost;
}
```

# 并查集

状态压缩 + `rank` 优化

```
int par[maxn], rk[maxn];

void init() {
    memset(rk, 0, sizeof rk);
    for (int i = 0; i < maxn; i++) par[i] = i;
}

inline int getfa(int x) {
    int fx = x, tmp;
    while (fx != par[fx]) fx = par[fx];
    while (x != fx) {//compress condition
        tmp = par[x];
        par[x] = fx;
        x = tmp;
    }
    return fx;
}

inline void combine(int x, int y) {
    if (rk[x = getfa(x)] > rk[y = getfa(y)])
        par[y] = x;
    else {
        par[x] = y;
        if (rk[x] == rk[y]) rk[y]++;
    }
}
```

}

## 拓扑排序

时间复杂度：$O(N + E)$

```
vector<int> g[maxn];
int cnt, np, ne, deg[maxn], order[maxn];

void topologic() {
    queue<int> q;
    for (int i = 0; i < np; i++)
        if (deg[i] == 0) q.push(i);
    while (!q.empty()) {
        int now = q.front(); q.pop();
        order[cnt++] = now;
        for (int i = 0; i < g[now].size(); i++)
            if ((--deg[g[now][i]]) == 0)
                q.push(g[now][i]);
    }
    if (cnt < np) printf("Failed.\n");
}

int main() {
    scanf("%d %d", &np, &ne);
    for (int i = 0; i < ne; i++) {
        int px, py;
        scanf("%d %d", &px, &py);
        g[px].push_back(py);
        deg[py]++;
    }
    topologic();
    for (int i = 0; i < np; i++)
        printf("%d ", order[i]);
    return 0;
}
```

# 网络流 - Edmonds-Karp

时间复杂度： $O(NE^2)$

```
int np, ne, g[maxn][maxn], flow[maxn], path[maxn];

int edmonds_karp(int ps, int pe) {
    int max_flow = 0; queue<int> q;
    while (1) {
        memset(flow, 0, sizeof flow);
        flow[ps] = INF; q.push(ps);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (int v = 1; v <= np; v++)
                if (!flow[v] && g[u][v]) {
                    path[v] = u;
                    q.push(v);
                    flow[v] = min(flow[u], g[u][v]);
                }
        }
        if (flow[pe] == 0) return max_flow;
        for (int p = pe; p != ps; p = path[p]) {
            g[path[p]][p] -= flow[pe];
            g[p][path[p]] += flow[pe];
        }
        max_flow += flow[pe];
    }
}

int main() {
    //g[px][py] += w;
    printf("%d\n", edmonds_karp(1, np));
    return 0;
}
```

# 网络流 - Dinic

时间复杂度: $O(EN^2)$

```
struct edge {
    int from, to, cap, flow;
    edge(int a, int b, int c, int d) {
        from = a; to = b; cap = c; flow = d;
    }
};
vector<edge> edges;
vector<int> g[maxn];
int np, ne, ps, pe, d[maxn], cur[maxn];
//d[i]: 起点到i的距离; cur[i]: 当前弧下标.
bool vis[maxn];

void add_edge(int from ,int to, int cap) {
    edges.push_back(edge(from, to, cap, 0));
    edges.push_back(edge(to, from, 0, 0));
    int m = edges.size();
    g[from].push_back(m - 2);
    g[to].push_back(m - 1);
}

bool BFS() {
    memset(vis, 0, sizeof vis);
    queue<int> q;
    q.push(ps); d[ps] = 0; vis[ps] = 1;
    while (!q.empty()) {
        int now = q.front(); q.pop();
        for (int i = 0; i < g[now].size(); i++) {
            edge& e = edges[g[now][i]];
            if (!vis[e.to] && int(e.cap) > e.flow) {
                q.push(e.to); vis[e.to] = 1;
                d[e.to] = d[now] + 1;
```

```
            }
        }
    }
    return vis[pe];
}

int DFS(int x, int a) {
    if (x == pe || a == 0) return a;
    int flow = 0, f;
    for (int &i = cur[x]; i < g[x].size(); i++) {
        edge& e = edges[g[x][i]];
        if (d[x] + 1 == d[e.to] &&
        (f = DFS(e.to, min(a, e.cap - e.flow))) > 0) {
            e.flow += f;
            edges[g[x][i] ^ 1].flow -= f;
            flow += f; a -= f;
            if (a == 0) break;
        }
    }
    return flow;
}

int max_flow() {
    int flow = 0;
    while (BFS()) {
        memset(cur, 0, sizeof cur);
        flow += DFS(ps, INF);
    }
    return flow;
}

int main() {
    //add_edge(x, y, c);
    ps = 1; pe = np;
    cout << max_flow() << endl;
    return 0;
}
```

# 网络流 - ISAP

时间复杂度：$O(EN^2)$

```
struct edge {
    int from, to, cap, flow;
    edge(int a, int b, int c, int d) {
        from = a; to = b; cap = c; flow = d;
    }
};
vector<edge> edges;
vector<int> g[maxn];
int np, ne, ps, pe;
int d[maxn], cur[maxn], p[maxn], num[maxn];
//d[i]: 起点到i的距离; cur[i]: 当前弧下标;
//p[i]: 可增广路径上的一条弧; num[i]: 距离标号计数
bool vis[maxn];

inline void add_edge(int from, int to, int cap) {
    edges.push_back(edge(from, to, cap, 0));
    edges.push_back(edge(to, from, 0, 0));
    int m = edges.size();
    g[from].push_back(m - 2);
    g[to].push_back(m - 1);
}

inline bool BFS() {
    memset(vis, 0, sizeof vis);
    queue<int> q;
    q.push(pe); d[pe] = 0; vis[pe] = 1;
    while (!q.empty()) {
        int now = q.front();
        q.pop();
        for (int i = 0; i < g[now].size(); i++) {
            edge& e = edges[g[now][i]];
```

```
            if (!vis[e.to] && int(e.cap) > e.flow) {
                vis[e.to] = 1;
                d[e.to] = d[now] + 1;
                q.push(e.to);
            }
        }
    }
    return vis[pe];
}

int augment() {
    int x = pe, a = INF;
    for (; x != ps; x = edges[p[x]].from) {
        edge& e = edges[p[x]];
        a = min(a, e.cap - e.flow);
    }
    for (x = pe; x != ps; x = edges[p[x]].from) {
        edges[p[x]].flow += a;
        edges[p[x] ^ 1].flow -= a;
    }
    return a;
}

inline int max_flow() {
    int flow = 0, x = ps;
    BFS();
    memset(num, 0, sizeof num);
    memset(cur, 0, sizeof cur);
    for (int i = 1; i <= np; i++) num[d[i]]++;
    while (d[ps] < np) {
        if (x == pe) { flow += augment(); x = ps; }
        bool ok = 0;
        for (int& i = cur[x]; i < g[x].size(); i++) {
            edge& e = edges[g[x][i]];
            if (e.cap > e.flow && d[x] == d[e.to] + 1) {
                ok = 1;
                p[e.to] = g[x][i];
                x = e.to;
```

```
                break;
            }
        }
        if (!ok) {
            int m = np - 1;
            for (int i = 0; i < g[x].size(); i++) {
                edge& e = edges[g[x][i]];
                if (e.cap > e.flow)
                    m = min(m, d[e.to]);
            }
            if (--num[d[x]] == 0) break; //gap
            num[d[x] = m + 1]++;
            cur[x] = 0;
            if (x != ps)x = edges[p[x]].from;
        }
    }
    return flow;
}

int main() {
    //add_edge(x, y, c);
    ps = 1; pe = np;
    printf("%d\n", max_flow());
    return 0;
}
```

# 强连通分量 - Tarjan

时间复杂度：$O(N+E)$

```
stack<int> s;
vector<int> g[maxn], gnew[maxn];
bool visited[maxn], instack[maxn];
int DFN[maxn], low[maxn], belong[maxn], ne, np, idx, cnt;

void tarjan(int p) {
    idx++;
    DFN[p] = low[p] = idx;
    s.push(p); visited[p] = instack[p] = 1;

    for (int i = 0; i < g[p].size(); i++)
            if (!visited[g[p][i]]) {
                tarjan(g[p][i]);
                low[p] = min(low[p], low[g[p][i]]);
            } else if (instack[g[p][i]])
                low[p] = min(low[p], DFN[g[p][i]]);

    if (DFN[p] == low[p]) {
        cnt++; int j;
        do {
            j = s.top(); s.pop();
            belong[j] = cnt;
            instack[j] = 0;
        } while (j != p);
    }
}

int main() {
    scanf("%d %d", &np, &ne);
    for (int i = 0; i < ne; i++) {
        int px, py;
```

```
        scanf("%d %d", &px, &py);
        g[px].push_back(py);
    }

    for (int i = 0; i < np; i++)
        if (!visited[i]) tarjan(i);

    for (int i = 0; i < np; i++)
        for (int j = 0; j < g[i].size(); j++)
            if (belong[i] != belong[g[i][j]])
                gnew[belong[i]].push_back(belong[g[i][j]]);
    return 0;
}
```

## 欧拉路

数据结构使用链式向前星，需要添加 `bool psd;` 属性。时间复杂度：$O(E)$

```
int np, ne, head[maxn], gsize;
stack<int> way;

void dfs(int now) {
    for (int i = head[now]; ~i; i = g[i].next)
        if (!g[i].psd) {
            g[i].psd = true;
            dfs(g[i].to);
        }
    way.push(now);
}

int main() {
    scanf("%d %d", &np, &ne);
    memset(head, -1, sizeof head);
    for (int i = 0; i < ne; i++) {
        int px, py;
        scanf("%d %d", &px, &py);
        add_edge(px, py);
        //add_edge(py, px);
    }
    dfs(1);
    while (!way.empty()) {
        printf("%d ", way.top()); way.pop();
    }
    return 0;
}
```

# 数据结构

- 建立二叉树
- 二叉搜索树
- 二叉堆
- 哈夫曼树
- RMQ & RSQ 问题
    - 稀疏表
    - 平方分割
    - 树状数组BIT
    - 线段树

## 建立二叉树

根据前序和中序遍历，或者中序和后序遍历来建立二叉树，时间复杂度 `O(NlogN)` 。

**注意**：每次建树之前，需要初始化 `l` ， `r` ，和 `cnt` 。

```
int pre[maxn], mid[maxn], suc[maxn];
int tree[maxn], l[maxn], r[maxn];
int cnt, n, temp;

void build1(int a, int b, int &idx) {
    tree[++idx] = pre[++cnt];
    int now = idx, t = -1;
    if (a == b) {
        if (mid[a] != tree[idx]) {/*Build failed*/}
        return;
    }
    for (int i = a; i <= b; i++)
        if (mid[i] == tree[idx]) {
            t = i; break;
        }
    if (t == -1) {/*Build failed*/}
    if (t > a) {
        l[now] = idx + 1;
        build1(a, t - 1, idx);
    }
    if (t < b) {
        r[now] = idx + 1;
        build1(t + 1, b, idx);
    }
}

void build2(int a, int b, int &idx) {
    tree[++idx] = suc[--cnt];
```

```
    int now = idx, t = -1;
    if (a == b) {
        if (mid[a] != tree[idx]) {/*Build failed*/}
        return;
    }
    for (int i = a; i <= b; i++)
        if (mid[i] == tree[idx]) {
            t = i; break;
        }
    if (t == -1) {/*Build failed*/}
    if (t < b) {
        r[now] = idx + 1;
        build2(t + 1, b, idx);
    }
    if (t > a) {
        l[now] = idx + 1;
        build2(a, t - 1, idx);
    }
}

int main() {
    memset(l, -1, sizeof l);
    memset(r, -1, sizeof r);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", &suc[i]);
    for (int i = 1; i <= n; i++)
        scanf("%d", &mid[i]);

    cnt = 0;
    build1(1, n, temp = 0);

    cnt = n + 1;
    build2(1, n, temp = 0);

    return 0;
}
```

# 二叉搜索树

插入、查找、删除操作时间复杂度均为$O(logN)$

**注意：**查找操作必须如此调用：

```
node* &ans = find(root, val);
```

```
struct node {
    node *l, *r;
    int val;
    node(int v = 0):val(v), l(0), r(0) {}
};
node* null_node = 0;//Don't use it

void ins(node* &root, int val) {
    if (root == 0) { root = new node(val); return; }
    if (val < root->val) ins(root->l, val);
    if (val > root->val) ins(root->r, val);
}

node* &find(node* &root, int val) {
    if (root == 0) { return null_node;}
    if (val < root->val) return find(root->l, val);
    if (val > root->val) return find(root->r, val);
    return root;
}

node* &getl(node* &root) {
    return (root->l == 0 ? root : getl(root->l));
}

void del(node* &root) {
    if (root->l == 0 && root->r == 0) {
```

```
        delete root; root = 0;
    } else if (root->l == 0) {
        node* tmp = root; root = root->r; delete tmp;
    } else if (root->r == 0) {
        node* tmp = root; root = root->l; delete tmp;
    } else {
        node* &tmp = getl(root->r);
        root->val = tmp->val;
        del(tmp);
    }
}
```

## 二叉堆

大根堆，插入、删除操作时间复杂度均为$O(logN)$。

```
int heap[maxn + 1], size, n;//heap[]: [1, size]

inline void up(int x) {
    int fa = x >> 1, tmp = heap[x];
    while (fa) {
        if (tmp > heap[fa]) heap[x] = heap[fa];//cmp
        else break;
        x = fa; fa = x >> 1;
    }
    heap[x] = tmp;
}

inline void down(int x) {
    int ch = x << 1, tmp = heap[x];
    while (ch <= size) {
        if (ch < size && heap[ch + 1] > heap[ch]) ch++;//cmp
        if (heap[ch] > tmp) heap[x] = heap[ch];//cmp
        else break;
        x = ch; ch = x << 1;
    }
    heap[x] = tmp;
}

inline void push(int val) { heap[++size] = val; up(size); }
inline int top() { return heap[1]; }
inline void pop() { heap[1] = heap[size--]; down(1); }
void build() {
    size = n;
    for (int i = n; i > 0; i--) down(i);
}
```

# 哈夫曼树

建树时间复杂度：$O(NlogN)$

```
struct node {
    int val;
    node *l, *r;
    node(int v = 0): val(v), l(0), r(0) {}
};
struct cmp {
    bool operator() (node* a, node* b) {
        return a->val > b->val;
    }
};
priority_queue<node*, vector<node*>, cmp> q;

node* huffman() {
    node* cur = NULL;
    while (q.size() > 1) {
        cur = new node;
        cur->l = q.top(); q.pop();
        cur->r = q.top(); q.pop();
        cur->val = cur->l->val + cur->r->val;
        q.push(cur);
    }
    return q.top();
}

int main() {
    //q.push(new node(v));
    node *root = huffman();
    return 0;
}
```

# RMQ & RSQ - 平方分割

| 空间 | 预处理 | 查询 | 更新 |
| --- | --- | --- | --- |
| $O(N)$ | $O(N)$ | $O(\sqrt{N})$ | $O(\sqrt{N})$ |

示例：区间加，区间求和。

```
ll sum[450], addv[450], a[200005];
int size, n, m;

inline void maintain(int idx, int k, int v) {
    a[k] += v;
    sum[idx] += v;
}

inline void add(int l, int r, int v) {//0 <= l, r < n
    int idx1 = l / size, idx2 = r / size, k;
    if (idx1 == idx2) {
        for (k = l; k <= r; ++k) maintain(idx1, k, v);
        return;
    }
    for (k = idx1 + 1; k < idx2; ++k) addv[k] += v;
    for (k = l; k < (idx1 + 1) * size; ++k)
        maintain(idx1, k, v);
    for (k = idx2 * size; k <= r; ++k)
        maintain(idx2, k, v);
}

inline ll query(int l, int r) {//0 <= l, r < n
    int idx1 = l / size, idx2 = r / size, k;
    ll ans = 0;
    if (idx1 == idx2) {
        for (k = l; k <= r; ++k) ans += a[k] + addv[idx1];
```

```
        return ans;
    }
    for (k = idx1 + 1; k < idx2; ++k)
        ans += addv[k] * size + sum[k];
    for (k = l; k < (idx1 + 1) * size; ++k)
        ans += a[k] + addv[idx1];
    for (k = idx2 * size; k <= r; ++k)
        ans += a[k] + addv[idx2];
    return ans;
}
```

## RMQ & RSQ - 稀疏表

| 空间 | 预处理 | 查询 | 更新 |
| --- | --- | --- | --- |
| $O(N \log N)$ | $O(N \log N)$ | $O(1)$ | $O(N \log N)$ |

```
#define MAX_LGN 15
int st[MAX_LGN][1 << MAX_LGN], a[maxn], n, m;

void init() {
    for (int j = 0; j < n; j++) st[0][j] = a[j];
    for (int i = 1; (1 << i) <= n; i++)
        for (int j = 0; j <= n - (1 << i); j++)
            st[i][j] = min(st[i - 1][j],
                st[i - 1][j + (1 << (i - 1))]);
}

inline int query(int l, int r) {
    int i = 0;
    while ((1 << (i + 1)) <= r - l + 1) i++;
    return min(st[i][l], st[i][r - (1 << i) + 1]);
}
```

## RMQ & RSQ - 树状数组

| 空间 | 预处理 | 查询 | 更新 |
| --- | --- | --- | --- |
| $O(N)$ | $O(N \log N)$ | $O(\log N)$ | $O(\log N)$ |

```
int tree[maxn], n;

inline int sum(int x) {
    int ans = 0;
    for (; x > 0; x -= (x & -x))
        ans += tree[x];
    return ans;
}

inline void add(int x, int val) {
    for (; x <= n; x += (x & -x))
        tree[x] += val;
}
```

# RMQ & RSQ - 线段树

| 空间 | 预处理 | 查询 | 更新 |
| --- | --- | --- | --- |
| $O(N)$ | $O(N)$ | $O(\log N)$ | $O(\log N)$ |

- 对于建树，首先需要找到一个最小的正整数$n'$，满足：$n' >= n$和$n'$为2的幂。于是$2n' - 1$即为整个线段树的结点数量。其中1号结点为 `root` ，$[1, n')$为非叶子结点，$[n', 2n')$为叶子结点。 实际有效的叶子结点在$[n', n' + n)$中，$[n' + n, 2n')$为无效结点。这些结点的维护信息不能影响到有效结点，比如应该置其 `sumv = 0` ， `maxv = -INF` ， `minv = INF` 。建树时间复杂度$O(n)$。
- 仅在 `flag()` 和 `push_down()` 操作中可以对结点的标记进行修改。如果是双标记或者是多标记线段树，则所有标记都需要考虑在内。 `push_down()` 操作还需要考虑两个子树。
- 仅在 `maintain()` 中更新结点维护的值（比如 `sumv` ， `maxv` ， `minv` 等），维护的信息要能在接近$O(1)$时间内维护完成。要求先将旧值清零，再根据左右子树的维护的值和自身的标记来更新自身维护的值。
- 对于 `update()` 和 `query()` 操作，其维护的区间从0开始还是从1开始并不重要，但 `root` 结点必须为1号点， `query` 操作同理。

  - 若$[a, b]$从0开始，则调用 `update(1, 0, _n - 1, a, b ,v)`
  - 若$[a, b]$从1开始，则调用 `update(1, 1, _n, a, b ,v)`

```
struct node {
    ll setv, sumv;
    node(): sumv(0), setv(-INF) {}
};

int n, m, _n;
node* tree;
```

```
void init() {
    _n = 1;
    while (_n < n) _n <<= 1;
    tree = new node[_n << 1];
    for (int i = _n; i < _n + n; i++) {/*read tree[i]*/}
    for (int p = _n >> 1; p; p >>= 1)
        for (int i = p; i < (p << 1); i++) {
            int lt = i << 1, rt = (i << 1) + 1;
            //tree[i].sumv = tree[lt].sumv + tree[rt].sumv;
        }
}

inline void maintain(int k, int l, int r) {}
inline void push_down(int k) {}
inline void flag(int k, int v) {}

inline void update(int k, int l, int r, int a, int b, int v) {
    int lt = k << 1, rt = (k << 1) + 1;
    if (a <= l && r <= b)
        flag(k, v);
    else {
        push_down(k);
        int mid = (l + r) >> 1;
        if (a <= mid) update(lt, l, mid, a, b, v);
            else maintain(lt, l, mid);
        if (mid < b) update(rt, mid + 1, r, a, b, v);
            else maintain(rt, mid + 1, r);
    }
    maintain(k, l, r);
}

inline int query(int k, int l, int r, int a, int b) {
    if (r < a || l > b) return 0;//or INF, -INF
    //if flag...(e.g. setv)
    if (a <= l && r <= b) {}
    int mid = (r + l) >> 1, lt = k << 1, rt = (k << 1) + 1;
    return query(lt, l, mid, ans) + query(rt, mid + 1, r, ans);
}
```

# 排序

- 快速排序
    - 随机快速排序
    - 中点快速排序

- 归并排序
- 基数排序
- `Kth number`
    - 固定区间单次查询
    - 不固定区间多次查询

## 快速排序

当待排序数据为随机序列时，中点快速排序的执行效率高于随机快速排序。

```
void qsort(int *begin, int *end) {
    if (end - begin <= 1) return;
    int key = *(begin + rand() % (end - begin - 1));
    int *i = begin, *j = end - 1;
    while (i <= j) {
        while (*i < key) i++;
        while (*j > key) j--;
        if (i <= j) swap(*(i++), *(j--));
    }
    qsort(begin, i);
    qsort(i, end);
}

void mid_qsort(int* begin, int* end) {//faster when random
    if (end - begin <= 1) return;
    int key = *(begin + ((end - begin - 1) >> 1));
    int *i = begin, *j = end - 1;
    while (i <= j) {
        while (*i < key) i++;
        while (*j > key) j--;
        if (i <= j) swap(*(i++), *(j--));
    }
    mid_qsort(begin, i);
    mid_qsort(i, end);
 }
```

## 归并排序

```
int temp[maxn];

void merge_sort(int* a, int low, int high) {
    if (low >= high) return;
    int mid = (low + high) >> 1;
    merge_sort(a, low, mid);
    merge_sort(a, mid + 1, high);

    int i = low, j = mid + 1, size = 0;
    for (; (i <= mid) && (j <= high); size++)
        if (a[i] < a[j]) temp[size] = a[i++];
        else temp[size] = a[j++];
    memcpy(temp + size, a + i, (mid - i + 1) << 2);
    memcpy(a + low, temp, (size + mid - i + 1) << 2);
}
```

## 基数排序

选取的基数为65536。

```
#define BASE (1 << 16)
#define maxn 10000005
int tmp[maxn], bkt[BASE + 5];

void radix_sort(int* begin, int* end) {
    int n = end - begin;
    for (int k = BASE - 1, i = 0; i < 2; i++, k <<= 16) {
        for (int j = 0; j < n; j++)
            bkt[((*(begin + j)) & k) >> (i * 16)]++;
        for (int j = 1; j < BASE; j++)
            bkt[j] += bkt[j - 1];
        for (int j = n - 1; j >= 0; j--)
            tmp[--bkt[((*(begin + j)) & k) >> (i * 16)]] =
                *(begin + j);
        memcpy(begin, tmp, n * sizeof(int));
        memset(bkt, 0, sizeof bkt);
    }
}
```

## Kth number

- 若固定区间单次查询，使用基于快速排序的Kth算法。
- 若不固定区间多次查询，使用基于线段树的Kth算法。

```
//find Kth smallest element, k: [1, n]; [begin, end)
int n, m, a[maxn];
int tmp[maxn], xx, yy, cc, kth;//[x, y], <= c
vector<int> tree[(1 << 18)];

void init(int k, int l, int r) {
    if (r - l == 1)
        tree[k].push_back(a[l]);
    else {
        int lc = k << 1, rc = (k << 1) + 1;
        init(lc, l, (l + r) >> 1);
        init(rc, (l + r) >> 1, r);
        tree[k].resize(r - l);
        merge(tree[lc].begin(), tree[lc].end(),
tree[rc].begin(), tree[rc].end(), tree[k].begin());
    }
    if (k == 1) {
        memcpy(tmp, a, sizeof(a));
        sort(tmp, tmp + n);
    }
}

int query(int k, int l, int r) {
    if (l >= yy || r <= xx) return 0;
    else if (xx <= l && r <= yy)
        return upper_bound(tree[k].begin(),
            tree[k].end(), cc) - tree[k].begin();
    else {
        int lc = k << 1, rc = (k << 1) + 1;
```

```
        return query(lc, l, (l + r) >> 1) +
            query(rc, (l + r) >> 1, r);
    }
}

//不固定区间多次查询，基于线段树：初始化O(logn)；查询O((logn)^3)
int st_find(int begin, int end, int k) {//segment tree
    xx = begin; yy = end; kth = k;
    int l = -1, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) >> 1;
        cc = tmp[mid];
        if (query(1, 0, n) >= kth) r = mid;
        else l = mid;
    }
    return tmp[r];
}

//固定区间单次查询，基于快速排序：O(NlogN)
int qfind(int *begin, int *end, int k) {
    if (end - begin <= 1) return *begin;
    int key = *(begin + rand() % (end - begin - 1));
    //中点：int key = *(begin + ((end - begin - 1) >> 1));
    int *i = begin, *j = end - 1;
    while (i <= j) {
        while (*i < key) i++;
        while (*j > key) j--;
        if (i <= j) swap(*(i++), *(j--));
    }
    if (k <= i - begin)
        return qfind(begin, i, k);
    else
        return qfind(i, end, k - (i - begin));
}
```

# 字符串

- 字符串匹配
  - KMP
  - Trie 树
  - AC 自动机

- 字符串循环左移

## 字符串匹配 - KMP

求解 `f` 的过程有一个优化： `f[i] = p[i] == p[j] ? f[j] : j;`

在利用 `f` 数组计算字符串周期时，只能使用原始转移方式： `f[i] = j;`

`t` 为文本串， `p` 为模式串。时间复杂度：$O(N+M)$。

```
int f[maxn];

void getf(char *p) {
    int len = strlen(p);
    memset(f, -1, sizeof f);
    for (int i = 0, j = -1; i < len;) {
        while (~j && p[i] != p[j]) j = f[j];
        i++; j++;
        f[i] = p[i] == p[j] ? f[j] : j;
    }
}

int kmp(char *t, char *p) {
    int len = strlen(t), lenp = strlen(p);
    getf(p);
    for (int i = 0, j = 0; i < len;) {
        while (~j && t[i] != p[j]) j = f[j];
        i++; j++;
        if (j == lenp) return i - j + 1;
    }
    return -1;
}
```

# 字符串匹配 - Trie 树

插入、查找时间复杂度 $O(len(s))$

```
//LA 3942
#define maxn 4005
#define maxl 105
#define maxw 300005

int size, trie[maxn * maxl][30], cnt[maxw];
bool flag[maxn * maxl];
char w[maxw];

void insert(char* s) {
    int p = 0, len = strlen(s);
    for (int i = 0; i < len; i++) {
        int c = s[i] - 'a';
        if (!trie[p][c])
            trie[p][c] = ++size;
        p = trie[p][c];
    }
    flag[p] = 1;
}

int main() {
    int idx = 0;
    while (scanf("%s", w) == 1) {
        size = 0;
        memset(trie, 0, sizeof trie);
        memset(cnt, 0, sizeof cnt);
        memset(flag, 0, sizeof flag);
        int n; char s[maxl]; scanf("%d", &n);
        for (int i = 0; i < n; i++) {
            scanf("%s", s);
            insert(s);
```

```
        }
        int len = strlen(w); cnt[len] = 1;
        for (int i = len - 1; ~i; i--) {
            int p = 0;
            for (int j = i; j < len; j++) {
                int c = w[j] - 'a';
                if (!trie[p][c]) break;
                p = trie[p][c];
                if (flag[p])
                    cnt[i] = (cnt[i] + cnt[j + 1]) % 20071027;
            }
        }
        printf("Case %d: %d\n", ++idx, cnt[0]);
    }
    return 0;
}
```

# 字符串匹配 - AC 自动机

- `flag` 数组仍用来判断结点是否为单词结点，使用 `vector<int>` 而非 `bool` 是为了防止当模式串重复时，后一个会覆盖前一个的情况，否则在统计出现次数时前一个模式串的出现次数将为 0。这样， `vector` 中的内容即为该单词结点对应的编模式串编号（可能有多个编号，对应的模式串均相同）。
- `num` 数组即用来统计：编号为 `i` 的模式串出现次数为 `num[i]` 次。
- 在 `getf()` 函数中，下面这条语句可以将树中不存在的边全部补上，这样在匹配时就不再需要失配函数：

```
if (u == 0) { trie[now][c] = trie[f[now]][c]; continue; }
```

- 在 `find()` 函数中，由于某个单词结点可能对应多个模式串的结尾，发现匹配时需要沿着失配边继续匹配其他模式串。这里有一个 `last` 优化，可以跳过沿路上的非单词结点，效果显著。

```
#define maxn 1005
#define maxl 55
#define maxw 2000010
#define maxnode maxn * maxl
#define sigma_size 26

int size, trie[maxnode][sigma_size], f[maxnode], last[maxnode],
num[maxnode];
vector<int> flag[maxnode];
char p[maxn][maxl], t[maxw];

inline void init() {
    memset(num, 0, sizeof num);
    memset(trie, 0, sizeof trie);
    memset(last, 0, sizeof last);
    for (int i = 0; i < maxnode; i++) flag[i].clear();
```

```
    size = 0;
}

inline int idx(char ch) { return ch - 'A'; }

inline void insert(char *s, int i) {
    int p = 0, len = strlen(s);
    for (int i = 0; i < len; i++) {
        int c = idx(s[i]);
        if (!trie[p][c]) trie[p][c] = ++size;
        p = trie[p][c];
    }
    flag[p].push_back(i);
}

inline void find(char* t) {
    int len = strlen(t);
    for (int i = 0, j = 0; i < len; i++) {
        int c = idx(t[i]);
        if (c >= sigma_size || c < 0) { j = 0; continue; }
        j = trie[j][c];
        for (int tmp = j; tmp; tmp = last[tmp])
            for (int t = 0; t < flag[tmp].size(); t++)
                num[flag[tmp][t]]++;
    }
}

void getf() {
    queue<int> q;
    memset(f, 0, sizeof f);
    for (int c = 0; c < sigma_size; c++)
        if (trie[0][c]) q.push(trie[0][c]);

    while (!q.empty()) {
        int now = q.front(); q.pop();
        for (int c = 0; c < sigma_size; c++) {
            int u = trie[now][c];
            if (u == 0) { trie[now][c] = trie[f[now]][c]; contin
```

```
ue; }
            q.push(u);
            int j = f[now];
            while (j && trie[j][c] == 0) j = f[j];
            f[u] = trie[j][c];
            last[u] = flag[f[u]].size() ? f[u] : last[f[u]];
        }
    }
}

int main() {
    int n;
    while (scanf("%d", &n) == 1) {
        init();
        for (int i = 1; i <= n; i++) {
            scanf("%s", p[i]);
            insert(p[i], i);
        }
        getf();
        scanf("%s", t); find(t);
        for (int i = 1; i <= n; i++)
            if (num[i] > 0) printf("%s: %d\n", p[i], num[i]);
    }
    return 0;
}
```

## 字符串循环左移

时间复杂度：$O(N)$。

```
void rev_str(string &s, int from, int to) {
    while (from < to) {
        char t = s[from];
        s[from++] = s[to];
        s[to--] = t;
    }
}

string left_rotate_str(string s, int m) {
    int n = s.size();
    m %= n;
    rev_str(s, 0, m - 1);
    rev_str(s, m, n - 1);
    rev_str(s, 0, n - 1);
    return s;
}

int main() {
    string a = "abcdef";
    cout << left_rotate_str(a, 21);
}
```

# 数学

- 素数
  - 判断素数
  - 求所有因子
  - 质因数分解
  - 筛法求素数

- 扩展欧几里得
- 快速幂
  - 整数快速幂
  - 矩阵快速幂

- 全排列
  - 递归版
  - 非递归版

- FFT

# 素数

| 判断素数 | 求所有因子 | 质因数分解 | 筛法求素数 |
|---|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(\log N)$ | $O(N\log\log N)$ |

```
ll prime[maxn];
int pnum;
bool notp[maxn];//e.g. notp[2] = 0, notp[4] = 1
bool notp_big[maxn];//for segment sieve

bool judge(ll x) {//判断素数
    for (ll i = 2; i * i <= x; i++)
        if (x % i == 0) return 0;
    return (x != 1) && (x != 0);
}

vector<ll> divisor(ll x) {//求所有因子
    vector<ll> ans;
    for (ll i = 2; i * i <= x; i++)
        if (x % i == 0) {
            ans.push_back(i);
            if (i != x / i) ans.push_back(x / i);
        }
    return ans;
}

map<ll, int> factor(ll x) {//质因数分解
    map<ll, int> ans;
    for (ll i = 2; i * i <= x; i++)
        while (x % i == 0) {
            ans[i]++;
            x /= i;
        }
```

```
    if (x != 1) ans[x]++;
    return ans;
}

int sieve(int x) {//筛法求[2, x)中素数
    memset(notp, 0, sizeof notp);
    pnum = 0;
    notp[0] = notp[1] = 1;
    for (int i = 2; i < x; i++)
        if (!notp[i]) {
            prime[pnum++] = (ll)i;
            for (int j = 2; i * j < x; j++)
                notp[i * j] = 1;
        }
    return pnum;
}

int segment_sieve(ll a, ll b) {//筛法求[a, b)中素数
    pnum = 0;
    memset(notp, 0, sizeof notp);
    memset(notp_big, 0, sizeof notp_big);
    //notp_big[x - a] == 0 -> x is prime
    for (int i = 2; (ll)i * i < b; i++)
        if (!notp[i]) {
            //sieve [2, sqrt(b))
            for (int j = (i << 1); (ll)j * j < b; j += i)
                notp[j] = 1;
            //sieve [a, b)
            for (ll j = max(2LL, (a + i - 1) / i) * i;
j < b; j += i)
                notp_big[j - a] = 1;
        }
    for (ll j = 0; j < b - a; j++)
        if (!notp_big[j])
            prime[pnum++] = a + j;
    return pnum;
}
```

## 扩展欧几里得

- 辗转相除法时间复杂度$O(h)$，`h` 为 `b` 在10进制下的位数。
- 扩展欧几里得算法，求满足 $ax+by=gcd(a,b)$ 的 $x$ 和 $y$ 。（求得结果必定满足 $x \le b$ 和 $y \le a$）

```
int gcd(int a, int b) {
    while (b) {
        int tmp = a % b;
        a = b; b = tmp;
    }
    return a;
}

int extgcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; y = 0; return a;
    } else {
        int r = extgcd(b, a % b, y, x);
        y -= (a / b) * x;
        return r;
    }
}
```

## 快速幂

整数和矩阵快速幂，时间复杂度均为$O(\log N)$。 `main` 中为求斐波那契数列第$n$项的代码。

```
#define MOD 10000007

struct matrix {
    int n, m;
    ll dat[maxn][maxn];//both start from 1
    matrix(int nn = 0, int mm = 0): n(nn), m(mm) {
        memset(dat, 0, sizeof dat);
        for (int i = 1; i <= n; i++) dat[i][i] = 1;
    }
    ll* operator[](const int i) { return dat[i]; }
};

inline matrix mat_mul(matrix &a, matrix &b) {
    matrix ans(a.n, a.n);
    for (int i = 1; i <= a.n; i++)
        for (int j = 1; j <= b.m; j++) {
            ll sum = 0;
            for (int k = 1; k <= a.m; k++)
                sum = (sum + a[i][k] * b[k][j]) % MOD;
            ans[i][j] = sum;
        }
    return ans;
}

matrix qpow_mat(matrix a, ll k) {
    matrix ans(a.n, a.n);
    for (; k; k >>= 1) {
        if (k & 1) ans = mat_mul(ans, a);
        a = mat_mul(a, a);
    }
```

```
    return ans;
}

ll qpow(ll a, ll k) {//(a^k)%MOD
    ll ans = 1;
    for (; k; k >>= 1) {
        if (k & 1) ans = (ans * a) % MOD;
        a = (a * a) % MOD;
    }
    return ans;
}

int main() {
    int n; scanf("%d", &n);
    matrix base(2, 2);
    base[1][1] = 1; base[1][2] = 1;
    base[2][1] = 1; base[2][2] = 2;
    int nn = (n - 1) / 2;
    matrix mm = qpow_mat(base, nn);
    if (n & 1)
        printf("%lld\n", mm[1][1] + mm[1][2]);
    else
        printf("%lld\n", mm[2][1] + mm[2][2]);
    return 0;
}
```

# 全排列 - 递归版

```
#include <stdio.h>
int a[100], n = 12;

void permutation(int x) {
    if (x == n - 1) {
        for (int i = 0; i < n; i++) printf("%d ", a[i]);
        printf("\n"); return;
    }
    int dup[100] = {};
    for (int i = x; i < n; i++) {
        if (dup[a[i]]) continue;
        dup[a[i]] = 1;
        swap(a[i], a[x]);
        permutation(x + 1);
        swap(a[i], a[x]);
    }
}

int main() {
    for (int i = 0; i < n; i++) a[i] = i + 1;
    permutation(0);
    return 0;
}
```

# 全排列 - 非递归版

```
#include <stdio.h>

inline void rev(int* from, int* to) {
    while (from < to) {
        swap(*from, *to);
        from++; to--;
    }
}

inline bool get_next_permutation(int* a, int size) {
    int i = size - 2;
    while ((i >= 0) && (a[i] >= a[i + 1])) i--;
    if (i < 0) return 0;
    int j = size - 1;
    while (a[j] <= a[i]) j--;
    swap(a[j], a[i]);
    rev(a + i + 1, a + size - 1);
    return 1;
}

int main() {
    int a[1005], size = 12;
    for (int i = 0; i < size; i++) a[i] = i + 1;
    do {
        for (int i = 0; i < size; i++) printf("%d ", a[i]);
        printf("\n");
    } while (get_next_permutation(a, size));
    return 0;
}
```

## FFT

快速傅里叶变换，包括多项式乘法、多项式求逆、多项式除法。

时间复杂度 $O(nlogn)$。

```
#define pi M_PI

struct comp {
    double re, im;
    comp(double r = 0, double i = 0): re(r), im(i) {}
    comp operator + (comp x) { return comp(re + x.re,
im + x.im); }
    comp operator - (comp x) { return comp(re - x.re,
im - x.im); }
    comp operator * (comp x) {
        return comp(re * x.re - im * x.im, re * x.im +
 im * x.re);
    }
};

void FFT(comp* a, int* g, int n, int f) {
    for (int i = 0; i < n; i++)
        if (g[i] > i) swap(a[i], a[g[i]]);
    for (int i = 1; i < n; i <<= 1) {
        comp wn1(cos(pi / i), f * sin(pi / i));
        for (int j = 0; j < n; j += (i << 1)) {
            comp wnk(1, 0);
            for (int k = 0; k < i; k++, wnk = wnk * wn
1) {
                comp x = a[j + k], y = wnk * a[j + k +
 i];
                a[j + k] = x + y; a[j + k + i] = x - y
;
            }
```

```
        }
    }
    if (!(~f)) for (int i = 0; i < n; i++) a[i].re /=
n;
}

void mul(comp* a, comp* b, comp* ans, int n) {
    int _n = 1, t = -1;
    while (_n < n) { _n <<= 1; t++; }
    int* g = new int[_n]; g[0] = 0;
    for (int i = 1; i < _n; i++)
        g[i] = (g[i >> 1] >> 1) | ((i & 1) << t);
    FFT(a, g, _n, 1); FFT(b, g, _n, 1);
    for (int i = 0; i < _n; i++) {
        ans[i] = a[i] * b[i];
        b[i] = comp(0, 0);
    }
    FFT(ans, g, _n, -1);
    delete[] g;
}

void inv(comp* a, comp* b, int n) {
    if (n == 1) { b[0].re = 1 / a[0].re; return; }
    inv(a, b, (n + 1) >> 1);
    int _n = 1, t = -1;
    while (_n < n) { _n <<= 1; t++; }
    int* g = new int[_n]; g[0] = 0;
    for (int i = 1; i < _n; i++)
        g[i] = (g[i >> 1] >> 1) | ((i & 1) << t);
    comp* tmp = new comp[_n];
    for (int i = 0; i < n; i++) tmp[i] = a[i];
    FFT(tmp, g, _n, 1); FFT(b, g, _n, 1);
    for (int i = 0; i < _n; i++) {
        tmp[i] = comp(2, 0) - tmp[i] * b[i];
        b[i] = tmp[i] * b[i];
    }
    FFT(b, g, _n, -1);
}
```

```
void div(comp* a, comp* b, comp* d, comp* r, int n, in
t m) {
    int _n = 1, t = -1;
    while (_n < (n - m + 1) << 1) { _n <<= 1; t++; }
    int* g = new int[_n]; g[0] = 0;
    for (int i = 1; i < _n; i++)
        g[i] = (g[i >> 1] >> 1) | ((i & 1) << t);
    for (int i = 0; i < (n >> 1); i++) swap(a[i], a[n
- i - 1]);
    for (int i = 0; i < (m >> 1); i++) swap(b[i], b[m
- i - 1]);
    comp* invb = new comp[_n];
    inv(b, invb, n - m + 2);
    FFT(a, g, _n, 1); FFT(invb, g, _n, 1);
    for (int i = 0; i < _n; i++) d[i] = a[i] * invb[i]
;
    FFT(d, g, _n, -1);
    for (int i = 0; i < ((n - m + 1) >> 1); i++)
        swap(d[i], d[(n - m + 1) - i - 1]);
    for (int i = n - m; ~i; i--) {
        d[i - 1].re += (d[i].re - floor(d[i].re)) * 10
;
        d[i].re = floor(d[i].re);
    }
}
```

# 哈希

- 整数哈希
- 字符串哈希

# 整数哈希

**初始化:**

```
memset(h, -1, sizeof(h));
```

```
#define mod1 1000003
#define mod2 1009
#define hash1(x) ((x & 0x7fffffff) % mod1)
#define hash2(x) ((x & 0x7fffffff) % mod2)
int h[mod1], m, n, t;

inline void ins(int x) {
    int idx, h1 = hash1(x), h2 = hash2(x);
    for (int i = 0;; i++) {
        idx = (h1 + i * h2) % mod1;
        if (h[idx] == x || h[idx] == -1) break;
    }
    h[idx] = x;
}

inline bool query(int x) {
    int idx, h1 = hash1(x), h2 = hash2(x);
    for (int i = 1;; i++) {
        idx = (h1 + i * h2) % mod1;
        if (h[idx] == -1) return false;
        if (h[idx] == x) return true;
    }
}
```

## 字符串哈希

```
const int mod1 = 100003;
char h[mod1][1005];

inline int BKDRHash(const char* s) {
    int ans = 0, len = strlen(s);
    for (int i = 0; i < len; i++)
        ans = ans * 131 + s[i];
    return (ans & 0x7fffffff) % mod1;
}

inline void ins(const char *s) {
    for (int idx = BKDRHash(s);; idx = (idx + 1) % mod1) {
        if (h[idx][0] == 0) { strcpy(h[idx], s); return; }
        if (strcmp(h[idx], s) == 0) return;
    }
}

inline bool query(const char *s) {
    for (int idx = BKDRHash(s);; idx = (idx + 1) % mod1) {
        if (h[idx][0] == 0) return false;
        if (strcmp(h[idx], s) == 0) return true;
    }
}
```

# 动态规划

- 最长不下降序列 - LIS
- 最长公共子序列 - LCS

## 最长不下降序列 - LIS

数组 `l[]` 即为LIS之一，时间复杂度：$O(nlogn)$。

```
int n, a[maxn], l[maxn], pre[maxn], sub[maxn] = {-1};

int LIS() {
    int len = 0;
    for (int i = 0; i < n; i++) {
        int l = 1, r = len;
        while (l <= r) {
            int mid = (l + r + 1) >> 1;
            if (a[sub[mid]] < a[i]) l = mid + 1;
            else r = mid - 1;
        }
        pre[i] = sub[l - 1];
        sub[l] = i;
        if (l > len) len = l;
    }
    return len;
}

void main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
    int len = LIS();
    for (int p = sub[len], cnt = len; ~p; p = pre[p])
        l[--cnt] = a[p];
}
```

## 最长公共子序列 - LCS

求两个字符串 `a` 和 `b` 的最长公共子序列，并统计最长序列的个数。

时间复杂度：$O(NM)$空间复杂度：$O(N+M)$

```
char a[MAXLEN], b[MAXLEN];
int dp[2][MAXLEN], sum[2][MAXLEN];

int main() {
    scanf("%s %s", a + 1, b + 1);
    int len1 = strlen(a + 1), len2 = strlen(b + 1);
    for (int j = 0; j < len2; j++) sum[0][j] = 1;
    sum[1][0] = 1;
    int now = 1, pre = 0;
    for (int i = 1; i < len1; i++) {
        for (int j = 1; j < len2; j++)
            if (a[i] == b[j]) {
                dp[now][j] = dp[pre][j - 1] + 1;
                sum[now][j] = sum[pre][j - 1]
+ (dp[now][j] == dp[pre][j]) * sum[pre][j]
+ (dp[now][j] == dp[now][j - 1]) * sum[now][j - 1];
            } else {
                dp[now][j] = max(dp[pre][j],
                    dp[now][j - 1]);
                sum[now][j] = 0
+ (dp[now][j] == dp[pre][j]) * sum[pre][j]
+ (dp[now][j] == dp[now][j - 1]) * sum[now][j - 1]
- (dp[now][j] == dp[pre][j - 1]) * sum[pre][j - 1];
            }
        swap(now, pre);
    }
    cout << dp[pre][len2 - 1] << sum[pre][len2 - 1];
}
```

# 其他

- 输入外挂
- VIM配置

## 输入外挂

```
char buffer[BufferSize], *head, *tail;

char getch() {
    if(head == tail) {
        int l = fread(buffer, 1, BufferSize, stdin);
        tail = (head = buffer) + l;
    }
    return *head++;
}

int getint() {
    int x = 0, flag = 0; char ch = getch();
    for(; !isdigit(ch); ch = getch())
        if (ch == '-') { flag = 1; break; }
    for(; isdigit(ch); ch = getch())
        x = x * 10 + ch - '0';
    return flag ? -x : x;
}
```

# VIM配置

```
set tabstop=4
set softtabstop=4
set shiftwidth=4
set noexpandtab
set nu
set cindent
set autoindent
set smartindent
set smarttab
set nobackup
set noswapfile
set showmatch
syntax enable
set ruler
set mouse=a
let &termencoding=&encoding
set fileencodings=utf-8,gbk
set clipboard+=unnamed

nmap <F2> :w <CR>
nmap <F3> :!open % <CR>
nmap <F4> :wq <CR>
nmap <F5> :!g++ % <CR>
nmap <F9> :!g++ -g -O2 % && ./a.out <CR>

inoremap ( ()<ESC>i
inoremap ) <c-r>=ClosePair(')')<CR>
inoremap { {}<ESC>i
inoremap } <c-r>=ClosePair('}')<CR>
inoremap <CR> <c-r>=CheckEnter()<CR>

function CheckEnter()
    if getline('.')[col('.') - 1] == '}'
```

```
        return "\<CR>\<CR>\<UP>\<TAB>"
    else
        return "\<CR>"
    endif
endf

function ClosePair(char)
    if getline('.')[col('.') - 1] == a:char
        return "\<Right>"
    else
        return a:char
    endif
endf

vmap <C-x> :!pbcopy<cr>i
vmap <C-c> :w !pbcopy<cr><cr><Esc>
nmap <C-v> :set paste<CR>:r !pbpaste<CR>:set nopaste<CR>i
imap <C-v> <Esc>:set paste<CR>:r !pbpaste<CR>:set nopaste<CR>i
nmap <C-a> ggvG<END>
imap <C-a> <Esc>ggvG<END>
nmap <C-z> u
imap <C-z> <Esc>ui
vmap <BS> <DEL><ESC>i
```