算法模板

Source: https://github.com/upupming/algorithm/blob/master/template.md

二分

始终要记住的一点是，如果新边界出现了 `mid - 1`，就需要在开始将 `mid` 赋值为 `(l + r + 1) >> 1`，多一个 `+ 1`。

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}

作者：yxc
链接：https://www.acwing.com/blog/content/277/
来源：AcWing
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

具体例子：

```
// 查找单调递增序列中 >= x 的最小一个数
while (l < r) {
    int mid = (l + r) >> 1;
    if (a[mid] >= x)
        r = mid;
    else
        l = mid + 1;
}
// 查找单调递增序列中 <= x 的最大一个数
while (l < r) {
    int mid = (l + r + 1) >> 1;
    if (a[mid] <= x)
        l = mid;
```

```
    else
        r = mid - 1;
}
```

高精度

作者：yxc 链接：https://www.acwing.com/blog/content/277/ 来源：AcWing 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```
// 通用构造函数
for (int i = a.size() - 1; i >= 0; i--) A.push_back(a[i] - '0');
// 通用输出函数
void out(const vector<int> &A) {
    for (int i = A.size() - 1; i >= 0; i--) {
        cout << A[i];
    }
}
// 通用比较函数
// A < B, A > 0, B > 0
bool cmp(vector<int> &A, vector<int> &B) {
    if (A.size() != B.size()) {
        return A.size() < B.size();
    }
    for (int i = A.size() - 1; i >= 0; i--) {
        if (A[i] != B[i]) return A[i] < B[i];
    }
    return false;
}
```

高精度加法

模板题 AcWing 791. 高精度加法

```
// O(n)
// C = A + B, A >= 0, B >= 0
vector<int> add(const vector<int> &A, const vector<int> &B) {
    if (A.size() < B.size()) return add(B, A);
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i++) {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if (t) C.push_back(t);
    return C;
}
```

高精度减法

模板题 AcWing 792. 高精度减法

```
// O(n)
// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(const vector<int> &A, const vector<int> &B) {
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i++) {
        // 如果借过位，需要减去 t（t 是借位数量）
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        // < 0  表示从高位借了一位出来了，所以 t = 1
        if (t < 0)
            t = 1;
        else
            t = 0;
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度乘低精度

模板题 AcWing 793. 高精度乘法

```
// O(n)
// C = A * b, A >= 0, b > 0
vector<int> mul(const vector<int> &A, int b) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size() || t; i++) {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t = t / 10;
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度乘高精度

```
// O(nm)
// C = A * B, A >= 0, B >= 0
vector<int> mul(const vector<int> &A, const vector<int> &B) {
    vector<int> C(A.size() + B.size());
    for (int i = 0; i < A.size(); i++) {
```

```
        for (int j = 0; j < B.size(); j++) {
            C[i + j] += A[i] * B[j];
        }
    }
    int t = 0;
    for (int i = 0; i < C.size(); i++) {
        t += C[i];
        C[i] = t % 10;
        t /= 10;
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度除以低精度

模板题 AcWing 794. 高精度除法

```
// O(n)
// A / b = C ... r, A >= 0, b > 0
vector<int> div(const vector<int>& A, int b, int& r) {
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i--) {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r = r % b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

高精度除以高精度

```
// O((n-m)*n)
pair<vector<int>, vector<int>> div(vector<int> A, const vector<int> &B) {
    vector<int> C, R;
    int n = A.size(), m = B.size(), d = n - m;
    C.resize(d + 1, 0);
    // 枚举补 0 的个数
    for (int len = d; len >= 0; len--) {
        vector<int> Bp(len, 0);
        for (int x : B) Bp.push_back(x);

        // A >= Bp
        while (!cmp(A, Bp)) {
            C[len] += 1;
            A = sub(A, Bp);
```

```
        }
    }
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    R = A;
    return make_pair(C, R);
}
```

快速幂

- AcWing 89

```
// 注意：底数可以取模，但是指数不能取模
// 模的性质：先模后乘（加）等于先乘（加）后模
int qpow(int a, int b, int p) {
    int ans = 1 % p;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % p;
        a = 1ll * a * a % p;
        b >>= 1;
    }
    return ans;
}

LL qpow(LL a, LL b, LL p) {
    LL ans = 1ll % p;
    while (b) {
        if (b & 1) ans = 1ll * ans * a % p;
        a = 1ll * a * a % p;
        b >>= 1;
    }
    return ans;
}
```

lowbit 运算

lowbit 运算返回的数是二进制表示下，原来数的最低位的 1 及后边所有的 0 组成的新的数。

```
lowbit(1010_2) = lowbit(10_2)

lowbit(n) = n & (~n + 1) = n & -n

~n == -1 - n => ~n + 1 == -n

// 预计算 H[2^k] = k，Hash 替代 log 运算
for (int i = 0; i <= 20; i++) H[1 << i] = i;
// 找出整数 n 的二进制表示下所有是 1 的位
while (n > 0) {
    cout << H[n & -n] << " ";
    n -= n & -n;
```

```
}
cout << endl;
// 将区间 [1, n] 分为 O(log n) 个小区间，每个区间长度都是当前值的 lowbit
while (x > 0) {
    cout << (x - (x & -x) + 1) << ", "<< x << endl;
    x -= x & -x;
}
```

快速排序

```
void quick_sort(int q[], int l, int r) {
    if (l >= r) return;
    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j) {
        do i++;
        while (q[i] < x);
        do j--;
        while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
```

归并排序

```
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = (l + r) >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

离散化

如果之后还需要进行查询，可以映射成有序数组。如果之后无需查询，直接映射成无序数组就可以了。

```
// 离散化
void discrete() {
    sort(a + 1, a + n + 1);
    for (int i = 1; i <= n; i++) {
        // 也可用 STL 的 unique 函数
        if (i == 1 || a[i] != a[i - 1])
            b[++m] = a[i];
    }
}
// 查询 x 映射为哪个 1-m 之间的整数
int query(int x) {
    return lower_bound(b + 1, b + m + 1, x) - b;
}

// ----------

// 也可使用 map，但是一般会比数组慢一些
unordered_map<int, int> mp;
void discrete() {
    for (int i = 1; i <=n; i++) {
        if (!mp.count(a[i]))
            mp[a[i]] = ++m;
    }
}
int query(int x) {
    return mp[x];
}
```

单调栈

```
int lft[N], rht[N], stk[N], tt;
// 哨兵
h[0] = -1, h[n + 1] = -1;
tt = 0;
stk[++tt] = 0;
for (int i = 1; i <= n; i++) {
    while (h[stk[tt]] >= h[i]) tt--;
    lft[i] = stk[tt];
    stk[++tt] = i;
}
tt = 0;
stk[++tt] = n + 1;
for (int i = n; i >= 1; i--) {
    while (h[stk[tt]] >= h[i]) tt--;
    rht[i] = stk[tt];
    stk[++tt] = i;
}
```

- 直方图中的最大矩形
- 城市游戏
- 最大面积
- LeetCode 795. Number of Subarrays with Bounded Maximum
- LeetCode 1944. Number of Visible People in a Queue (套路比较深的单调栈)

单调队列

```
for (int i = 1; i <= n; i++) {
    cin >> a, sum += a;
    // 队首一直出队直到满足 <=m 条件
    while (!dq.empty() && (i - dq.front().second > m)) {
        dq.pop_front();
    }
    if (dq.empty()) {
        ans = max(ans, sum);
    } else {
        ans = max(ans, sum - dq.front().first);
        // 不断删除队尾的不会优于当前 i 的左端点
        while (!dq.empty() && dq.back().first >= sum) dq.pop_back();
    }
    dq.emplace_back(sum, i);
}
```

- 最大子序和

Trie 树

- AcWing 142 - 144, 1414

```
// 在 Trie 树中，边表示一个字符，节点表示一个前缀，叶子节点表示一个字符串，cnt 用来计数
叶子节点表示的字符串出现的次数
// N 为字符串总长度，26 表示字符可能取值范围大小（对于小写字符串为 26，对于二进制数为
2）
// trie[p][ch] = 0 表示叶子节点，为了语义上的一致性，取 p = 1 作为根节点，初始情况不
插入任何字符串，就已经有一个节点了
// 也可以将 p = 0 作为根节点，++tot 改为 tot++ 即可
int trie[N][26], tot = 1, cnt[N], n, m;
void insert(string s) {
    int len = s.length(), p = 1;
    for (int k = 0; k < len; k++) {
        int ch = s[k] - 'a';
        if (trie[p][ch] == 0) trie[p][ch] = ++tot;
        p = trie[p][ch];
    }
    cnt[p]++;
}
int search(string t) {
    int len = t.length(), p = 1, ans = 0;
```

```
    for (int k = 0; k < len; k++) {
        p = trie[p][t[k] - 'a'];
        if (p == 0) return ans;
        ans += cnt[p];
    }
    return ans;
}

// 另一种在每个节点上都记录 cnt 的写法，支持 insert(a, -1) 删除操作
int trie[N][2], tot = 1, cnt[N];
void insert(int a, int v) {
    int p = 1;
    for (int i = 30; i >= 0; i--) {
        int j = a >> i & 1;
        if (trie[p][j] == 0) {
            trie[p][j] = ++tot;
        }
        p = trie[p][j];
        cnt[p] += v;
    }
}

int query(int a) {
    int ans = 0, p = 1;
    for (int i = 30; i >= 0; i--) {
        int j = a >> i & 1;
        if (cnt[trie[p][!j]]) {
            p = trie[p][!j];
            ans |= 1 << i;
        } else {
            p = trie[p][j];
        }
    }
    return ans;
}
```

邻接表模板

邻接表最简单的方法是直接用 vector 存储，但是用数组存储速度更快，也有一些优点（快速找到反向边）。这里采用的是蓝书的方法，y 总的方法于此略不同，参考我的分享。

- AcWing 257

```
// N, M 分别表示点数和边数，注意如果是无向图的话 M 一定要乘 2，否则会数组越界

// head[x] = m 表示点 x 的邻接表的表头是编号为 m 的边
// ver[m] 表示编号为 m 的边的终点
// edge[m] 表示编号为 m 的边的权值
int head[N], ver[M], edge[M], Next[M], tot;

// 加入有向边 (x, y)，权值为 z
```

```
void add(int x, int y, int z) {
    // 真实数据
    ver[++tot] = y, edge[tot] = z;
    // 在表头 x 处插入
    Next[tot] = head[x], head[x] = tot;
}

// 访问从 x 出发的所有边
for (int i = head[x]; i; i = Next[i]) {
    int y = ver[i], z = edge[i];
    // 找到了一条有向边 (x, y)，权值为 z
}

// 一般来说，memset 初始化就够了
memset(head, 0, sizeof head), tot = 0;

// 多测试用例时，初始化使用循环更好，例如: AcWing 3696
tot = 0;
for (int i = 1; i <= n; i++) {
    head[i] = 0;
    deg[i] = 0;
}
// Next 不需要初始化也可以的，因为每次 add 的时候会对用到的 Next 的改变都是基于 head
的
for (int i = 1; i <= m; i++) {
    Next[i] = 0;
}
```

最大公约数

```
// 欧几里得算法
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
```

另外，最小公倍数 lcm(a, b) = a * b / gcd(a, b)。

扩展欧几里得算法

求 ax + by = gcd(a, b) 的一对整数解 (x, y)。一般形式 ax + by = c 有解，当且仅当 d | c（d 是 a 和 b 的最大公约数）。

```
int extgcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    int d = extgcd(b, a % b, x, y);
    int z = x;
```

```
    x = y;
    y = z - a / b * y;
    return d;
}
```

并查集

```
// sz 表示节点的秩，这里定义为节点的元素个数
int fa[N], sz[N], n;
void init() {
    // 初始化
    for (int i = 0; i <= n; i++) {
        fa[i] = i;
        sz[i] = 1;
    }
}
// Get 操作，使用「路径压缩」+「按秩合并」，时间复杂度为反阿克曼函数，可以认为是常数
int get(int x) {
    if (x == fa[x]) return x;
    // 路径压缩，fa 直接赋值为代表元素
    return fa[x] = get(fa[x]);
}
// Merge 操作，同上，可以认为时间复杂度为常数
void merge(int x, int y) {
    int fx = get(x), fy = get(y);
    if (fx == fy) return;
    if (sz[fx] < sz[fy]) {
        fa[fx] = fy;
        sz[fy] += sz[fx];
    } else {
        fa[fy] = fx;
        sz[fx] += sz[fy];
    }
}

// 一定记得初始化并查集
init();
```

另外还有「边带权」和「拓展域」的并查集，在原有并查集的基础上维护一些具有传递关系的属性。

树状数组

RMQ 常用算法，参见蓝书 0x42。`c[x]` 保存序列 a 的区间 `[x - lowbit(x) + 1, x]` 中所有数的和。`c[x]` 的父节点为 `c[x + lowbit(x)]`

支持「单点增加」和「区间查询」，结合差分可以支持「区间增加」。

```
// 查询前缀和：查询序列 a 第 1~x 个数的和
int ask(int x) {
```

```
    int ans = 0;
    for (; x; x -= x & -x) ans += c[x];
    return ans;
}
// 单点增加：给序列中的一个数 a[x] 加上 y
// 算法：自下而上每个节点都要增加 y
void add(int x, int y) {
    for (; x <= n; x += x & -x) c[x] += y;
}
```

```
// https://www.cnblogs.com/qdscwyy/p/9759220.html

int lowbit(int x) {
    return x & (-x);
}
void updata(int x, int k) {
    while (x <= n) {
        h[x] = k;
        int low = lowbit(x);
        for (int i = 1; i < low; i <<= 1)
            h[x] = max(h[x], h[x - i]);
        x += lowbit(x);
    }
}
// 区间查询 [x, y] 的 max
int query(int x, int y) {
    int ans = 0;
    while (y >= x)
    {
        ans = max(a[y], ans), y -= 1;
        for (; y-lowbit(y) >= x; y -= lowbit(y))
            ans = max(h[y], ans);
    }
    return ans;
}
```

线段树

RMQ 常用算法，树状数组基于区间划分，线段树则是基于分治。

```
struct SegmentTree {
    int l, r;
    int dat;
} tree[N * 4];

// 线段树的建树，时间复杂度：O(N)
// p 表示节点编号，[l, r] 表示节点所代表的区间
void build(int p, int l, int r) {
    tree[p].l = l, tree[p].r = r;
```

```
    // 叶节点，表示单个元素
    if (l == r) {
        tree[p].dat = a[l];
        return;
    }
    int mid = (l + r) >> 1;
    // 左子节点：编号为 2*p，代表区间 [l, mid]
    build(2 * p, l, mid);
    // 右子节点：编号为 2*p+1，代表区间 [mid+1, r]
    build(2 * p + 1, mid + 1, r);
    // 从下往上合并更新信息
    tree[p].dat = max(tree[2 * p].dat, tree[2 * p + 1].dat);
}


// 线段树的单点修改，时间复杂度：O(log N)
// 将 a[x] 的值修改为 v
void change(int p, int x, int v) {
    // 找到叶节点
    if (tree[p].l == tree[p].r) {
        tree[p].dat = v;
        return;
    }
    int mid = (tree[p].l + tree[p].r) >> 1;
    // x 属于左半区间
    if (x <= mid) change(2 * p, x, v);
    // x 属于右半区间
    else
        change(2 * p + 1, x, v);
    // 从下往上合并更新信息
    tree[p].dat = max(tree[2 * p].dat, tree[2 * p + 1].dat);
}

// 线段树的区间查询，时间复杂度：O(log N)
// 查询序列 a 在区间 [l, r] 上的最大值
int ask(int p, int l, int r) {
    // 查询区间 [l, r] 完全包含节点 p 所代表的的区间
    if (l <= tree[p].l && r >= tree[p].r) return tree[p].dat;
    int mid = (tree[p].l + tree[p].r) >> 1;
    // 负无穷大
    int val = -(1 << 30);
    // 左子节点 [tree[p].l, mid] 与查询 [l, r] 有重合
    if (l <= mid) val = max(val, ask(2 * p, l, r));
    // 右子节点 [mid+1, tree[p].r] 与查询 [l, r] 有重合
    if (r >= mid + 1) val = max(val, ask(2 * p + 1, l, r));
    return val;
}

// 调用入口
build(1, 1, n);
change(1, x, v);
```

支持区间修改的线段树（延迟标记）

区间修改的时间复杂度可以通过延迟标记从 O(N) 降为 O(log N)。

```
// 注意：有延迟标记的节点，本身已经完成了数据更新，只是没有传递给子节点
struct SegmentTree {
    int l, r;
    long long dat, lazy;
#define l(x) tree[x].l
#define r(x) tree[x].r
#define dat(x) tree[x].dat
#define lazy(x) tree[x].lazy
} tree[N * 4];
int a[N];

void build(int p, int l, int r) {
    l(p) = l, r(p) = r;
    if (l == r) {
        dat(p) = a[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(2 * p, l, mid);
    build(2 * p + 1, mid + 1, r);
    dat(p) = dat(2 * p) + dat(2 * p + 1);
}
void spread(int p) {
    // 节点 p 有延迟标记的话
    if (lazy(p)) {
        // 更新左子节点信息，延迟值 * 区间长度等于节点的增加量
        dat(2 * p) += lazy(p) * (r(2 * p) - l(2 * p) + 1);
        // 更新右子节点信息
        dat(2 * p + 1) += lazy(p) * (r(2 * p + 1) - l(2 * p + 1) + 1);
        // 给左子节点打延迟标记
        lazy(2 * p) += lazy(p);
        // 给右子节点打延迟标记
        lazy(2 * p + 1) += lazy(p);
        // 清除 p 的标记
        lazy(p) = 0;
    }
}

void change(int p, int l, int r, int d) {
    // 完全覆盖
    if (l <= l(p) && r >= r(p)) {
        // 更新节点信息，每个节点增加量 d * 区间长度 = 节点增加量
        dat(p) += (long long)d * (r(p) - l(p) + 1);
        // 给节点打延迟标记
        lazy(p) += d;
        return;
    }
    // 因为即将访问下面的节点了，必须先下传延迟标记
```

```
    spread(p);
    int mid = (l(p) + r(p)) >> 1;
    // 和左子节点有相交部分
    if (l <= mid) change(2 * p, l, r, d);
    // 和右子节点有相交部分
    if (r >= mid + 1) change(2 * p + 1, l, r, d);
    dat(p) = dat(2 * p) + dat(2 * p + 1);
}

long long ask(int p, int l, int r) {
    if (l <= l(p) && r >= r(p)) return dat(p);
    // 因为即将访问下面的节点了，必须先下传延迟标记
    spread(p);
    int mid = (l + r) >> 1;
    long long val = 0;
    if (l <= mid) val += ask(2 * p, l, r);
    if (r >= mid + 1) val += ask(2 * p + 1, l, r);
    return val;
}
```

- AcWing 243

质数筛选

```
// Eratosthenes 筛法，时间复杂度 O(\sum_{质数 p <= n} n/p) = O(n log log n)
void primes(int n) {
    // 合数标记
    memset(v, 0, sizeof v);
    for (int i = 2; i <= n; i++) {
        if (v[i]) continue;
        cout << i << endl;
        for (int j = i; j <= n/i; j++) v[i*j] = 1;
    }
}
```

```
int primes[N], cnt;
bool st[N];
// 线性筛法
// 每个数只会被自己的最小质因子筛掉，时间复杂度为 O(n)
void get_primes_l(int n) {
    for (int i = 2; i <= n; i++) {
        if (!st[i]) primes[cnt++] = i;
        // j < cnt 没有必要，因为 i 是合数的时候，枚举到最小质因子一定会 break
        // 如果 i 是质数 primes[j] == i 的时候也会停止循环
        for (int j = 0; primes[j] <= n / i; j++) {
            // primes[j] 是 primes[j] * i 的最小质因子
            st[primes[j] * i] = true;
            // primes[j] 一定是 i 的最小质因子（因为 j 是从小到大枚举的）
            // 之所以要 break 是因为后面的 primes[j+1] 不再是 primes[j+1] * i
```

```
的最小质因子了，而是 primes[j]，因为 i % primes[j] == 0
            if (i % primes[j] == 0) break;
        }
    }
}
```

拓扑排序

```
// 拓扑排序模板
void topsort() {
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        if (deg[i] == 0) q.push(i);
    }
    while (q.size()) {
        int x = q.front();
        q.pop();
        a[++cnt] = x;
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i];
            if (--deg[y] == 0) q.push(y);
        }
    }
}
```

Treap

```
// 数组模拟链表
struct Treap {
    // 左右子节点在数组中的下标
    int l, r;
    // 节点的关键码、权值
    int val, dat;
    // 副本数、子树大小
    int cnt, size;
} a[N];
int tot, root, n, INF = 0x7fffffff;

int New(int val) {
    a[++tot].val = val;
    // 随机初始化权值
    a[tot].dat = rand();
    a[tot].cnt = a[tot].size = 1;
    return tot;
}

// 类似线段树自下往上的更新过程
void Update(int p) {
    a[p].size = a[a[p].l].size + a[a[p].r].size + a[p].cnt;
```

```
}

void Build() {
    // 为避免越界，减少边界情况特殊判断，加入哨兵
    New(-INF), New(INF);
    root = 1, a[1].r = 2;
    Update(root);
}

// 把 p 的左子节点绕着 p 向右旋转，注意 p 是引用
void zig(int &p) {
    int q = a[p].l;
    a[p].l = a[q].r, a[q].r = p, p = q;
    Update(a[p].r), Update(p);
}
// 把 p 的右子节点绕着 p 向左旋转，注意 p 是引用
void zag(int &p) {
    int q = a[p].r;
    a[p].r = a[q].l, a[q].l = p, p = q;
    Update(a[p].l), Update(p);
}

// 注意 p 是引用
void Insert(int &p, int val) {
    if (p == 0) {
        p = New(val);
        return;
    }
    // 如果之前已经有相同关键码的节点，只需要 cnt++ 即可
    if (val == a[p].val) {
        a[p].cnt++, Update(p);
        return;
    }
    // 在左子树中插入
    if (val < a[p].val) {
        Insert(a[p].l, val);
        // 不满足堆性质，右旋
        if (a[p].dat < a[a[p].l].dat) zig(p);
    }
    // 在右子树中插入
    else {
        Insert(a[p].r, val);
        // 不满足堆性质，左旋
        if (a[p].dat < a[a[p].r].dat) zag(p);
    }
    Update(p);
}

int GetPre(int val) {
    // a[1].val = -INF
    int ans = 1;
    int p = root;
    // 一直循环直到找到一个关键码为 val 的节点，找到之后会直接 break
    // 找不到也没关系，已经经过的节点中一定包含答案，ans 即为所求
```

```
    while (p) {
        if (val == a[p].val) {
            if (a[p].l > 0) {
                p = a[p].l;
                // 左子树一直往右走
                while (a[p].r > 0) p = a[p].r;
                ans = p;
            }
            // 检索成功之后会 break 掉
            break;
        }
        // 节点 p 是小于 val 的，并且相比 ans 离 val 更近，更新 ans
        if (a[p].val < val && a[p].val > a[ans].val) ans = p;
        // 根据 val 情况往左或者往右走
        p = val < a[p].val ? a[p].l : a[p].r;
    }
    return a[ans].val;
}

int GetNext(int val) {
    // a[2].val == INF
    int ans = 2;
    int p = root;
    while (p) {
        if (val == a[p].val) {
            if (a[p].r > 0) {
                p = a[p].r;
                // 右子树一直往左走
                while (a[p].l > 0) p = a[p].l;
                ans = p;
            }
            break;
        }
        if (a[p].val > val && a[p].val < a[ans].val) ans = p;
        p = val < a[p].val ? a[p].l : a[p].r;
    }
    return a[ans].val;
}
```

- AcWing 253

最短路

Dijkstra

```
// 朴素：O(N^2)，适用于 M 比较大，N 很小的情况
void dijkstra() {
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    // 重复 n - 1 次
    for (int i = 1; i < n; i++) {
```

```
        int x = 0;
        // 找到未标记节点中 dist 最小的
        for (int j = 1; j <= n; j++) {
            if (!v[j] && (x == 0 || d[j] < d[x]))
                x = j;
        }
        v[x] = 1;
        // 用全局最小值点 x 更新其他节点
        for (int y = 1; y <= n; y++)
            d[y] = min(d[y], d[x] + a[x][y]);
    }
}
// 堆优化：O(M log N)，适用于 N 比较大的情况
void dijkstra() {
typedef pair<int, int> PII;
    priority_queue<PII, vector<PII>, greater<PII>> pq;
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    pq.push({0, 1});
    while (pq.size()) {
        // 取出堆顶
        int x = pq.top().second;
        pq.pop();
        if (v[x]) continue;
        v[x] = 1;
        // 扫描所有出边
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i], z = edge[i];
            if (d[y] > d[x] + z) {
                d[y] = d[x] + z;
                pq.push({d[y], y});
            }
        }
    }
}
```

Bellman-Ford & SPFA

```
// Bellman-Ford 算法，时间复杂度 O(NM)
// 在求解有边数限制的最短路问题时，一般选择 Bellman-Ford 算法更好，最外层循环次数就是
边数限制
// 在 Bellman-Ford 算法中，最后 d[n] 虽然是无穷大，但是中间可能被一些负权值更新过，略
小于 0x3f3f3f3f，其实也可以在 relax 的是时候做一个特判，无穷大的边不 relax 就行
// 最后无穷大的判断用 > 0x3f3f3f3f / 2
struct P {
    int x, y, z;
} e[M];
memset(d, 0x3f, sizeof d);
d[1] = 0;
for (int i = 1; i <= k; i++) {
    memcpy(last, d, sizeof d);
```

```
    for (int j = 1; j <= m; j++) {
        // 从上次的 last 距离转移，而不是这次的进行转移，否则边数多于 i 了
        d[e[j].y] = min(d[e[j].y], last[e[j].x] + e[j].z);
    }
}
// SPFA 求最短路，在随机图上时间复杂度为 O(km)，其中 k 是较小的常数，在特殊构造的图上可能退化为 O(nm)
void spfa() {
    queue<int> q;
    memset(d, 0x3f, sizeof d);
    d[1] = 0, v[1] = 1;
    q.push(1);
    while (q.size()) {
        // 取出队头
        int x = q.front();
        q.pop();
        v[x] = 0;
        // 扫描所有出边
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i], z = edge[i];
            if (d[y] > d[x] + z) {
                // 更新，把新的二元组插入堆
                d[y] = d[x] + z;
                if (!v[y]) q.push(y), v[y] = 1;
            }
        }
    }
}
// SPFA 判断负环
bool spfa() {
    // d 数组无需初始化，因为我们最终需要求的不是真正的距离

    // 因为要求所有可能的负环，所以所有点都当做起点加进去
    for (int i = 1; i <= n; i++) {
        v[i] = 1;
        q.push(i);
    }

    while (q.size()) {
        // 取出队头
        int x = q.front();
        q.pop();
        v[x] = 0;
        // 扫描所有出边
        for (int i = head[x]; i; i = Next[i]) {
            int y = ver[i], z = edge[i];
            if (d[y] > d[x] + z) {
                // 更新，把新的二元组插入堆
                d[y] = d[x] + z;
                cnt[y] = cnt[x] + 1;
                if (cnt[y] >= n) return true;
                if (!v[y]) q.push(y), v[y] = 1;
            }
```

```
        }
    }
}
```

Floyd

```
// 跟 Bellman-Ford 算法一样，最后 d[n] 虽然是无穷大，但是中间可能被一些负权值更新过，略小于 0x3f3f3f3f，其实也可以在 relax 的是时候做一个特判，无穷大的边不 relax 就行
// 最后无穷大的判断用 > 0x3f3f3f3f / 2
memset(d, 0x3f, sizeof d);
for (int i = 1; i <= n; i++) d[i][i] = 0;
for (int i = 1; i <= m; i++) {
    cin >> x >> y >> z;
    d[x][y] = min(d[x][y], z);
}
// floyd 求任意两点间最短路径
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

最小生成树

Prim

```
// Prim 采用加点的思想，时间复杂度为 O(N^2)，堆优化版为 O(M log N)，适用于稠密图，一般直接用朴素版即可
// 因为在点少的时候使用，如果点多的话，直接用 Kruskal 就好了，而且 Kruskal 好写很多
void prim() {
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    for (int i = 1; i < n; i++) {
        int x = 0;
        for (int j = 1; j <= n; j++) {
            if (!v[j] && (x == 0 || d[j] < d[x]))
                x = j;
        }
        v[x] = 1;
        for (int y = 1; y <= n; y++) {
            // 注意这里是和 Dijkstra 的区别所在，求的是到集合的距离，不需要加 d[x]了
            // 这里可以求出生成树中 y 的前驱结点是谁
            if (!v[y]) d[y] = min(d[y], a[x][y]);
        }
    }
}
```

Kruskal

```
// Kruskal 是加边的算法，用到了并查集维护生成森林的所有点，时间复杂度为 O(M log M)
struct P {
    int x, y, z;
    bool operator<(const P& b) {
        return z < b.z;
    }
} edge[M];
sort(edge + 1, edge + m + 1);
// 并查集初始化
for (int i = 1; i <= n; i++) fa[i] = i;
// 求最小生成树
for (int i = 1; i <= m; i++) {
    int x = get(edge[i].x);
    int y = get(edge[i].y);
    if (x == y) continue;
    fa[x] = y;
    ans += edge[i].z;
}
// 可以逐一判断是否属于一个集合
// y 总则是看看上面的循环是不是 merge 了 n-1 次
for (int i = 2; i <= n; i++) {
    if (get(i) != get(1)) {
        puts("impossible");
        return 0;
    }
}
```

二分图

染色法判定二分图

```
bool dfs(int x, int color) {
    v[x] = color;
    for (int i = head[x]; i; i = Next[i]) {
        int y = ver[i];
        if (!v[y]) {
            if (!dfs(y, 3 - color)) return false;
        } else {
            if (v[y] == color) return false;
        }
    }
    return true;
}
bool isBipartite() {
    for (int i = 1; i <= n; i++) {
        if (!v[i]) {
```

```
            if (!dfs(i, 1)) return false;
        }
    }
    return true;
}
```

求二分图的最大匹配

```
bool dfs(int x) {
    for (int i = head[x], y; i; i = Next[i]) {
        if (!v[y = ver[i]]) {
            v[y] = 1;
            // 如果 y 正好没有男朋友；或者 y 有男朋友，但是 y 的男朋友 match[y] 可以换一个妹子的话；那么就把 y 的男朋友设置为 x
            if (!match[y] || dfs(match[y])) {
                match[y] = x;
                return true;
            }
        }
    }
    return false;
}
for (int i = 1; i <= n1; i++) {
    memset(v, 0, sizeof v);
    if (dfs(i)) ans++;
}
```

KMP 算法

```
// 求 A 在 B 中的各次出现位置
void calcNext() {
    // KMP 模板，Next[i] 表示「A 中以 i 结尾的非前缀子串」与「A 的前缀」能够匹配的最大长度
    Next[1] = 0;
    // j 的值在 while 循环中不断减小，j = Next[j] 的执行次数不会超过每层 for 循环开始时 j 的值与 while 循环结束时 j 的值之差
    // 每层 for 循环，j 的值至多增加 1，j 始终非负，因此减小幅度总和不会超过增加幅度总和
    // j 的变化次数至多为 2(N +M)，算法时间复杂度为 O(N + M)
    for (int i = 2, j = 0; i <= n; i++) {
        while (j > 0 && a[i] != a[j + 1]) j = Next[j];
        if (a[i] == a[j + 1]) j++;
        Next[i] = j;
    }
}
for (int i = 1, j = 0; i <= m; i++) {
    while (j > 0 && (j == n || b[i] != a[j + 1])) j = Next[j];
    if (b[i] == a[j + 1]) j++;
    // f[i] 表示「B 中以 i 结尾的非前缀子串」与「A 的前缀」能够匹配的最大长度
```

```
        f[i] = j;
        if (j == n) {
            // A 在 B 中某次出现的起始下标
            printf("%d ", i - n);
        }
    }
```

字符串哈希

```
typedef unsigned long long ULL;
const int P = 131;
p[0] = 1;
int n = s.length();
for (int i = 1; i <= n; i++) {
    f[i] = f[i - 1] * P + (s[i - 1] - 'a' + 1);
    p[i] = P * p[i - 1];
}
ULL getHash(int l, int r) {
    return f[r] - f[l - 1] * p[r - l + 1];
}
if (getHash(l1, r1) == getHash(l2, r2)) {
    cout << "Yes" << endl;
} else {
    cout << "No" << endl;
}
```

Manacher 求最长回文子串

注意，「子串」和「子序列」是不同的，「子串」是原字符串中连续的一段，「子序列」原字符串中选一些字符保持原来的顺序构成的新的串。

```
// p[i]表示 Str 中以下标i为回文中心的最大回文半径。
// 如果我们得到了p[i]，那么p[i] - 1就是原串 S 以i为回文中心的最大回文长度
// rt表示已经计算过的回文串能达到的最远右边界的下一个位置,mid表示rt所对应的最左侧的回文中心
// rt=max(j+p[j]),j \in [1,i-1]
// mid + p[mid] == rt

int manacher() {
    n = strlen(s);
    str[0] = '!', str[1] = '#'; /* str[0]为哨兵 */
    for (int i = 0; i < n; i++) {
        str[i * 2 + 2] = s[i];
        str[i * 2 + 3] = '#';
    }
    m = n * 2 + 1;
    str[m + 1] = '@'; /* 哨兵 */

    int rt = 0, mid = 0;
```

```
    int res = 0;
    for (int i = 1; i <= m; i++) {
        p[i] = i < rt ? min(p[2 * mid - i], rt - i) : 1;
        while (str[i + p[i]] == str[i - p[i]]) p[i]++;
        if (i + p[i] > rt) {
            rt = i + p[i];
            mid = i;
        }
        res = max(res, p[i] - 1);
    }
    return res;
}
```


致谢

本模板多数出自《算法竞赛进阶指南》（蓝书）和 AcWing 社区，非常感谢蓝书作者、AcWing 社区的 y 总和其他的小伙伴。