

## 1 前言

本文记录了 *NOIP* 涉及知识点和部分代码实现

## 2 基础算法

主要有模拟、枚举、贪心、二分、分治、倍增  
加粗为重点知识点

### 2.1 模拟、枚举、贪心、分治、倍增

遇到具体题目，具体分析，灵活运用即可

### 2.2 二分

#### 2.2.1 二分答案

一般遇到题目里出现最大值最小，最小值最大这样的字眼，一般都需要使用二分答案

在可能的答案区间里找出问题的答案，大多数情况下用于求解满足某种条件下的最大（小）值，前提是答案具有单调性

模板题目：*Luogu P1873* 砍树

主体代码：

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

具体 `check()` 函数部分根据具体问题编写。而且 `l = ...` 和 `r = ...` 的部分也要根据具体情况灵活更改

### 2.2.2 二分查找

经常使用 C++ STL 内的 `lower_bound()` 和 `upper_bound()` 来完成，复杂度  $O(\log n)$

- `lower_bound(begin, end, num)`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个大于或等于 `num` 的数字，找到返回该数字的地址，不存在则返回 `end`，通过返回的地址减去起始地址 `begin`，得到找到数字在数组中的下标
- `upper_bound(begin, end, num)`: 从数组的 `begin` 位置到 `end-1` 位置二分查找第一个大于 `num` 的数字，找到返回该数字的地址，不存在则返回 `end`，通过返回的地址减去起始地址 `begin`，得到找到数字在数组中的下标
- `lower_bound(begin, end, num, greater<type>())`: 第一个小于或等于 `num`
- `upper_bound(begin, end, num, greater<type>())`: 第一个小于 `num`

实例:

```
int num[6] = {1, 2, 4, 7, 15, 34};
int pos1 = lower_bound(num, num + 6, 7) - num;
//pos1 = 3    num[pos1] = 7
int pos2 = upper_bound(num, num + 6, 7) - num;
//pos2 = 4    num[pos2] = 15
```

## 2.3 离散化

假设 `int` 范围内有 `n` 个整数 `a[1]~a[n]`，去重后有 `m` 个整数；则可以把 `a[i]` 用 `1~m` 间的整数来代替，并保持大小顺序不变。实现了用小整数代替大整数运行算法

```

void discrete() {
    sort(a + 1, a + 1 + n);
    for (int i = 1; i <= n; ++i) {
        if (i == 1 || a[i] != a[i - 1]) {
            b[++m] = a[i];
        }
    }
}

int query(int x) {
    return lower_bound(b + 1, b + 1 + m, x) - b;
}

```

i 代替的数值为 b[i], a[i] 被 query(a[i]) 代替

## 3 动态规划

### 3.1 线性 DP、高维 DP、背包 DP

根据具体问题具体分析，拥有动态规划思想即可

#### 3.1.1 LIS 问题

### 3.2 树形 DP

以图的方式建树，之后按照递归的写法，从根节点初始化到叶节点，再一层一层向上根据状态转移方程更新

```

void DP(int x) {
    该节点初始化
    for (int i = head[x]; i; i = e[i].next) {
        int y = e[i].to;
        DP(y);
        根据状态转移方程更新
    }
}

```

模板题目: *Luogu P1352* 没有上司的舞会  
 还有局域网 192.168.0.123 里山东 1127、1130 题

### 3.3 区间 DP

区间 dp 就是在区间上进行动态规划, 求解一段区间上的最优解。  
 主要是通过合并小区间的最优解进而得出整个大区间上最优解的 dp 算法。

区间长度作为阶段, 左端点作为状态, 转移方程作为决策  
 重点: 分清阶段、状态、决策

经典例题: *Luogu P1880* 石子合并

```
for (int len = 2; len <= n; ++len) { // 阶段
    for (int l = 1; l <= n - len + 1; ++l) { // 状态: 左端点
        int r = l + len - 1; // 状态: 右端点
        for (int k = l; k < r; ++k) { // 决策
            f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r]);
        }
        f[l][r] += sum[r] - sum[l - 1];
    }
}
```

### 3.4 状压 DP

即使用二进制状态压缩来优化 DP  
 没学过, 更没写过。逃...

但我知道例题: *Luogu P1879* 玉米田

## 4 数论

### 4.1 欧几里得算法

```
LL gcd(LL a, LL b) {
```

```

        return b == 0 ? a : gcd(b, a % b);
    }

```

## 4.2 埃式筛法

```

bool vis[maxn];
int prime[maxn];
void getprime(int n) {
    int m = (int)sqrt(n + 0.5), num = 0;
    memset(vis, 0, sizeof(vis));
    vis[0] = vis[1] = 1;
    for (int i = 2; i <= m; ++i) if (!vis[i]) {
        prime[++num] = i;
        for (int j = i * i; j <= n; j += i) vis[j] = 1;
    }
}

```

## 4.3 扩展欧几里得 (线性同余方程)

```

void exgcd(LL a, LL b, LL& d, LL& x, LL& y) {
    if (!b) { d = a; x = 1; y = 0; }
    else { exgcd(b, a % b, d, y, x); y -= x * (a / b); }
}

```

## 4.4 乘法逆元

在大数相除取余时要转换为乘以乘法逆元，求逆元方法：

```

LL inv(LL a, LL n) {
    LL d, x, y;
    exgcd(a, n, d, x, y);
    return d == 1 ? (x + n) % n : -1;
}

```

### 4.5 快速乘，快速幂

```
LL mul_mod(LL a, LL b, LL n){
    LL res = 0;
    while (b > 0) {
        if (b & 1) res = (res + a) % n;
        a = (a + a) % n;
        b >>= 1;
    }
    return res;
}

LL pow_mod(LL a, LL p, LL n){
    int res = 1;
    while (p) {
        if (p & 1) res = 1LL * res * a % n;
        a = 1LL * a * a % n;
        p >>= 1;
    }
    return res;
}
```

## 5 数据结构

### 5.1 单调栈，单调队列

即在普通的栈、队列基础上维护整个栈和队列的单调性

例题：Luogu P3467PLA-Posting

### 5.2 堆

不多说，可以使用 `priority_queue`

### 5.3 并查集

动态维护若干个不重叠的集合，并支持合并与查询

```
int ufs[maxn];
for (int i = 1; i <= n; ++i) ufs[i] = i;
int find(int x) {
    return x == ufs[x] ? x : ufs[x] = find(ufs[x]);
}
void unionn(int x, int y) {
    ufs[get(x)] = get(y);
}
```

带权并查集:

```
int find(int x) {
    if (ufs[x] == x) return x;
    int fx = find(ufs[x]);
    d[x] += d[ufs[x]];
    return ufs[x] = fx;
}
void union(int x, int y) {
    int fx = find(x), fy = find(y);
    ufs[fx] = fy; d[fx] = size[fy];
    size[fy] += size[fx];
}
```

带撤销并查集:

```
int ufs[maxn];
void init() {
    for (int i = 0; i <= n; ++i) {
        ufs[i] = i;
    }
}
inline void sroot(int u) { //换u为根
    for (int i = 0, fa = ufs[u]; u; fa = ufs[u]) {
        ufs[u] = i; i = u; u = fa;
    }
}
```

```

    }
}
void connect(int u, int v) {
    ufs[u] = v;
}
void deleteuv(int u, int v) {
    ufs[v] = 0;
}
void query(int u, int v) {
    for (; v != u && v; v = ufs[v]);
    puts(v == u ? "Yes" : "No");
}

```

具体讲解,例题在我的博客<https://www.luogu.org/blog/tony180116/DataStructure-UFS>

#### 5.4 树状数组

维护前缀和的数据结构, 支持单点增加, 查询前缀和

**lowbit(n)** 操作: 非负整数  $n$  在二进制表示下最低位的 1 及其后边所有的 0 构成的数值

对于一个原序列  $a[]$ , 可以建立一个数组  $tree[]$ , 来保存  $a$  的区间  $[x - \text{lowbit}(x) + 1, x]$  内值的和, 即

$$tree[x] = \sum_{i=x-\text{lowbit}(x)+1}^x a[i]$$

同时  $tree$  数组可以看成是一个树形结构, 并满足以下性质

- 每个节点  $tree[x]$  保存以  $x$  为根的子树中所有叶节点的和
- 每个节点  $tree[x]$  的子节点个数等于  $\text{lowbit}(x)$  的位数
- 除树根外, 每个节点  $tree[x]$  的父亲节点为  $tree[x + \text{lowbit}(x)]$
- 树的深度为  $\log_2 n$



```

int tree[maxn], n, m;
int lowbit(int k) { return k & -k; }
void add(int x, int k) {
    for (; x <= maxn; x += x & -x) tree[x] += y;
}
int query(int x) {
    int ans = 0;
    for (; x; x -= x & -x) ans += tree[x];
    return ans;
}
int init() {
    for (int i = 1; i <= n; ++i) {
        add(i, a[i]);
    }
}

```

模板: *Luogu P3374* 树状数组 1 - 单点修改, 区间查询

*Luogu P3368* 树状数组 2 - 区间修改, 单点查询

*Luogu P3372* 线段树 1 - 区间修改, 区间查询

## 5.5 线段树

基于分治思想的二叉树形数据结构, 可以用于区间上的数据维护

```

struct SegmentTreeNode { // 树上的节点
    int l, r;                // 节点表示区间的左右端点
    long long sum, add;      // 区间和和add的延迟标记
    #define l(x) tree[x].l   // 方便访问
    #define r(x) tree[x].r
    #define sum(x) tree[x].sum
    #define add(x) tree[x].add
} tree[maxn << 2];
int a[maxn], n, m;
void pushup(int p) {
    sum(p) = sum(p<<1) + sum(p<<1|1);
}

```

```

}
void build(int p, int l, int r) { //建树
    l(p) = l, r(p) = r;           //设置左右端点
    if (l == r) { sum(p) = a[l]; return; } //到达叶子节点
    int mid = (l + r) >> 1;
    build(p<<1, l, mid);           //递归构建左右树
    build(p<<1|1, mid + 1, r);
    pushup(p);
}
void pushdown(int p) { //下传延迟标记
    if (add(p)) { //如果有标记
        sum(p<<1) += add(p) * (r(p<<1) - l(p<<1) + 1); //sum传至左儿子
        sum(p<<1|1) += add(p) * (r(p<<1|1) - l(p<<1|1) + 1); //右儿子
        add(p<<1) += add(p);       //延迟标记传至左儿子
        add(p<<1|1) += add(p);     //右儿子
        add(p) = 0; //本节点延迟标记清零
    }
}
void update(int p, int l, int r, int d) { //更新区间内值
    if (l <= l(p) && r >= r(p)) { //完全覆盖
        sum(p) += (long long)d * (r(p) - l(p) + 1); //更新节点信息
        add(p) += d; //打上延迟标记
        return;
    }
    pushdown(p); //下传标记
    int mid = (l(p) + r(p)) >> 1;
    if (l <= mid) update(p<<1, l, r, d); //递归更新左右
    if (r > mid) update(p<<1|1, l, r, d);
    pushup(p);
}
long long query(int p, int l, int r) { //查询操作
    if (l <= l(p) && r >= r(p)) return sum(p); //完全覆盖
    pushdown(p); //下传标记

```

```

    int mid = (l(p) + r(p)) >> 1;
    long long ans = 0;
    if (l <= mid) ans += query(p<<1, l, r);    // 加上左右部分值
    if (r > mid) ans += query(p<<1|1, l, r);
    return ans;
}

```

模板: *Luogu P3372* 线段树 1 - 区间加, 区间查询和

推荐题目: *SPOJ GSS17* Can you answer these queries

<https://www.luogu.org/blog/tony180116/DataStructure-SegmentTree>

## 5.6 ST 表

求静态区间最大值, 查询复杂度  $O(1)$

模板题目: *Luogu P3865* ST 表

```

Log[0] = -1;
for (int i = 1; i <= n; ++i) {
    f[i][0] = a[i];
    Log[i] = Log[i >> 1] + 1;
}
for (int j = 1; j <= LogN; ++j) {
    for (int i = 1; i + (1 << j) - 1 <= n; ++i)
        f[i][j] = max(f[i][j - 1], f[i + (1 << j - 1)][j - 1]);
}
while (m--) {
    scanf("%d %d", &l, &r);
    int s = Log[r - l + 1];
    printf("%d\n", max(f[l][s], f[r - (1 << s) + 1][s]));
}

```

## 5.7 哈希表

模板题目: *Luogu P4305* 不重复数字

```
#define p 100003
```

```

#define hash(a) a%p
int h[p], t, n, x;
int find(int x) {
    int y;
    if (x < 0) y = hash(-x);
    else y = hash(x);
    while (h[y] && h[y] != x) y = hash(++y);
    return y;
}
void push(int x) {
    h[find(x)] = x;
}
bool check(int x) {
    return h[find(x)] == x;
}

```

## 5.8 Trie 字典树

没见过几道题

```

struct Trie {
    int ch[maxn][26], sz, val[maxn];
    Trie() {
        sz = 1;
        memset(ch[0], 0, sizeof(ch[0]));
        memset(val, 0, sizeof(val));
    }
    int idx(char c) { return c - 'a'; }
    void insert(char *s, int v) {
        int u = 0, n = strlen(s);
        for (int i = 0; i < n; ++i) {
            int c = idx(s[i]);
            if (!ch[u][c]) {
                memset(ch[sz], 0, sizeof(ch[sz]));

```

```

        val[sz] = 0;
        ch[u][c] = sz++;
    }
    u = ch[u][c];
}
val[u] = v;
}
int search(char *s) {
    int u = 0, n = strlen(s);
    for (int i = 0; i < n; ++i) {
        int c = idx(s[i]);
        if (!ch[u][c]) return -1;
        u = ch[u][c];
    }
    return val[u];
}
};

```

## 6 图论

由于我习惯用 `vector<>` 建图，所以这里模板均是 `vector` 写的

### 6.1 最短路 (Dijkstra+Heap, SPFA, Floyd)

团队博客里有两篇相关文章

#### 6.1.1 Dijkstra 堆优化

不能含负权边

```

struct Edge {
    int from, to, dis;
    Edge(int u, int v, int w): from(u), to(v), dis(w) {}
};
vector<Edge> edges;

```

```
vector<int> G[maxn];
void add(int u, int v, int w) { //单向边
    edges.push_back(Edge(u, v, w));
    int mm = edges.size();
    G[u].push_back(mm - 1);
}

struct heap {
    int u, d;
    bool operator < (const heap& a) const {
        return d > a.d;
    }
};

int n, m, d[maxn];

void Dijkstra(int s) {
    priority_queue<heap> q;
    memset(d, 0x3f, sizeof(d));
    d[s] = 0;
    q.push((heap){s, 0});
    while (!q.empty()) {
        heap top = q.top(); q.pop();
        int u = top.u, td = top.d;
        if (td != d[u]) continue;
        for (int i = 0; i < G[u].size(); ++i) {
            Edge& e = edges[G[u][i]];
            if (d[e.to] > d[u] + e.dis) {
                d[e.to] = d[u] + e.dis;
                q.push((heap){e.to, d[e.to]});
            }
        }
    }
}
```

```
}
```

### 6.1.2 SPFA 及负环

反正比堆优化的 Dijkstra 慢就是了

```
struct Edge {
    int from, to, val;
    Edge(int u, int v, int w): from(u), to(v), val(w) {}
};
vector<Edge> edges;
vector<int> G[maxn];
void add(int u, int v, int w) {
    edges.push_back(Edge(u, v, w));
    int mm = edges.size();
    G[u].push_back(mm - 1);
}

int dist[maxn], vis[maxn];
int n, m, w;

void SPFA(int s) {
    queue<int> q;
    q.push(s);
    dist[s] = 0; vis[s] = 1;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        vis[u] = 0;
        for (int i = 0; i < G[u].size(); ++i) {
            Edge& e = edges[G[u][i]];
            if (dist[e.to] > dist[u] + e.val) {
                dist[e.to] = dist[u] + e.val;
                if (!vis[e.to]) {
                    vis[e.to] = 1;
                    q.push(e.to);
                }
            }
        }
    }
}
```

```

        }
    }
}
}
}

```

如果任意一条边被修改大于  $n$  次 (执行  $n$  次松弛操作), 这个图内一定存在至少一个负环

## 6.2 最小生成树 Kruskal

```

struct Edge {
    int from, to, val;
    Edge(int u, int v, int w): from(u), to(v), val(w) {}
};
vector<Edge> edges;
vector<int> G[maxn];
void add(int u, int v, int w) {
    edges.push_back(Edge(u, v, w));
    int mm = edges.size();
    G[u].push_back(mm - 1);
}
bool cmp(Edge a, Edge b) {
    if (a.val != b.val) return a.val < b.val;
    if (a.from != b.from) return a.from < b.from;
    return a.to < b.to;
}

int ufs[maxn], n, m, ans = 0, cnt = 1;
int find(int x) { return ufs[x] == x ? x : ufs[x] = find(ufs[x]); }

void Kruskal() {
    for (int i = 1; i <= n; ++i) ufs[i] = i;
    sort(edges.begin(), edges.end(), cmp);
}

```



```

    for (int i = 0; i < m; ++i) {
        Edge& e = edges[i];
        int x = find(e.from), y = find(e.to);
        if (x != y) {
            ufs[x] = y;
            ans += e.val;
            cnt++; // 运行后若 cnt!=n, 则图不连通
        }
    }
}

```

### 6.3 二分图匹配

不是重点，但是学过

### 6.4 Tarjan

没学过，目前不会，溜了溜了

### 6.5 树相关

#### 6.5.1 树的 DFS 序

在进行深度优先遍历时，依次经过的节点序列，长度为  $2n$ ，可以把树转化成区间问题

```

void dfs(int x) {
    a[++cnt] = x;
    vis[x] = true;
    for (int i = 0; i < son[x].size(); ++i) {
        int y = son[x][i];
        if (vis[y]) continue;
        dfs(y);
    }
    a[++cnt] = x;
}

```

### 6.5.2 树的深度

```
void dfs(int x) {
    vis[x] = true;
    for (int i = 0; i < son[x].size(); ++i) {
        int y = son[x][i];
        if (vis[y]) continue;
        dep[y] = dep[x] + 1;
        dfs(y);
    }
}
```

### 6.5.3 树的重心

```
void dfs(int x) {
    vis[x] = true; size[x] = 1;
    int max_part = 0;
    for (int i = 0; i < son[x].size(); ++i) {
        int y = son[x][i];
        if (vis[y]) continue;
        dfs(y);
        size[x] += size[y];
        max_part = max(max_part, size[y]);
    }
    max_part = max(max_part, n - size[x]);
    if (max_part < ans) {
        ans = max_part;
        pos = x;
    }
}
```

### 6.5.4 树的直径

```
int dfs(int x) {
```

```

    int res1 = 0, res2 = 0;
    for (int i = 0; i < G[x].size(); ++i) {
        Edge& e = edges[G[x][i]];
        res2 = max(res2, dfs(e.to) + e.val);
        if (res2 > res1) swap(res1, res2);
    }
    ans = max(ans, res1 + res2);
    return res1;
}

```

结果：ans

#### 6.5.5 最近公共祖先 (LCA)

模板：Luogu P3379 最近公共祖先

```

int n, m, s;
int dep[maxn], f[maxn][20], lg[maxn];

void dfs(int u, int fa) {
    dep[u] = dep[fa] + 1;
    f[u][0] = fa;
    for (int i = 1; (1 << i) <= dep[u]; ++i) {
        f[u][i] = f[f[u][i - 1]][i - 1];
    }
    for (int i = 0; i < G[u].size(); ++i) {
        Edge& e = edges[G[u][i]];
        if (e.to != fa) {
            dfs(e.to, u);
        }
    }
}

int LCA(int x, int y) {
    if (dep[x] < dep[y]) swap(x, y);
}

```

```

while (dep[x] > dep[y]) {
    x = f[x][lg[dep[x]] - dep[y]] - 1;
}
if (x == y) return x;
for (int k = lg[dep[x]] - 1; k >= 0; k--) {
    if (f[x][k] != f[y][k]) {
        x = f[x][k];
        y = f[y][k];
    }
}
return f[x][0];
}

```

lg 数组的预处理：

```

for (int i = 1; i <= n; ++i) {
    lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
}

```

## 7 字符串算法

### 7.1 KMP 字符串匹配

模板：Luogu P3375KMP 字符串匹配

```

char s1[maxn], s2[maxn]; int l1, l2, nxt[maxn];
void pre() {
    nxt[1] = 0;
    int j = 0;
    for (int i = 1; i < l2; ++i) {
        while (j > 0 && s2[j + 1] != s2[i + 1]) j = nxt[j];
        if (s2[j + 1] == s2[i + 1]) j++;
        nxt[i + 1] = j;
    }
}

```

```

void kmp() {
    int j = 0;
    for (int i = 0; i < l1; ++i) {
        while (j > 0 && s2[j + 1] != s1[i + 1]) j = nxt[j];
        if (s2[j + 1] == s1[i + 1]) j++;
        if (j == l2) {
            printf("%d\n", i - l2 + 2);
            j = nxt[j];
        }
    }
}

```

## 7.2 字符串哈希

模板: *Luogu P3370* 字符串哈希

各种哈希: [https://tony031218.github.io/2019/01/10/Cpp - - /](https://tony031218.github.io/2019/01/10/Cpp-%20-%20/)

```

typedef unsigned long long ULL;
ULL base = 131, a[10010], mod = 19260817;
char s[10010];

ULL hash(char* s) {
    int len = strlen(s);
    ULL Ans = 0;
    for (int i = 0; i < len; i++)
        Ans = (Ans * base + (ULL)s[i]) % mod;
    return Ans;
}

```

## 7.3 Trie 字典树

上面说过了

## 8 实用技巧

### 8.1 别忘了文件输入

```
freopen("题目名.in", "r", stdin);  
freopen("题目名.out", "w", stdout);
```

文件名不要打错。。。

### 8.2 头文件

在考场上，尽量不要使用 `bits/stdc++.h` 万能头，可能评测机没有这个库，但是我们可以去电脑里找这个头文件的位置，把其中要用的东西复制过来

```
#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <cstring>  
#include <ctime>  
#include <algorithm>  
#include <bitset>  
#include <deque>  
#include <iostream>  
#include <list>  
#include <map>  
#include <queue>  
#include <set>  
#include <stack>  
#include <string>  
#include <vector>
```

大概这些就是常用的头文件

### 8.3 读入优化

没什么好说的，大数据输入一定要用

```

inline int read() {
    int x = 0; int f = 1; char ch = getchar();
    while (!isdigit(ch)) {if (ch == '-') f = -1; ch = getchar();}
    while (isdigit(ch)) {x = x * 10 + ch - 48; ch = getchar();}
    return x * f;
}

inline LL read() {
    int x = 0; int f = 1; char ch = getchar();
    while (!isdigit(ch)) {if (ch == '-') f = -1; ch = getchar();}
    while (isdigit(ch)) {x = x * 10 + ch - 48; ch = getchar();}
    return x * f;
}

```

建议在经常调用的函数前面加上 `inline`

## 8.4 随机数

做不上的题目或者子任务，不要空下，按照输出格式输出随机数  
记得在 `main()` 函数开头加上 `srand((int)time(0));`，再使用 `rand()`

## 8.5 时间检测

因为考试环境是 Linux，我们可以使用本机命令 `time` 来测试自己的程序运行最大数据（题目提供）所需时间

命令: `time ./problem < problem.in > a.out`

## 8.6 根据数据大小判断算法时间复杂度

$n \leq 100$  -  $O(n^3)$  或更小

$n \leq 10000$  -  $O(n^2)$  或更小 ( $O(n^2)$  不一定)

$n \leq 100000$  -  $O(n \cdot \sqrt{n})$  或更小

$n \leq 1000000$  -  $O(n \log n)$  或更小

$n \geq 1000000$  -  $O(n)$  或  $O(1)$

大概就这些了，不够以后再补

*NOIp* 2019 `rp++`