基础算法

常用函数

```cpp
int mypow(int n, int k, int p = MOD) { // 复杂度是 log N
    int r = 1;
    for (; k; k >>= 1, n = n * n % p) {
        if (k & 1) r = r * n % p;
    }
    return r;
}
i64 mysqrt(i64 n) { // 针对 sqrt 无法精确计算 ll 型
    i64 ans = sqrt(n);
    while ((ans + 1) * (ans + 1) <= n) ans++;
    while (ans * ans > n) ans--;
    return ans;
}
int mylcm(int x, int y) {
    return x / gcd(x, y) * y;
}
```

```cpp
template<class T> int log2floor(T n) { // 针对 log2 无法精确计算 ll 型；向下取整
    assert(n > 0);
    for (T i = 0, chk = 1;; i++, chk *= 2) {
        if (chk <= n && n < chk * 2) {
            return i;
        }
    }
}
template<class T> int log2ceil(T n) { // 向上取整
    assert(n > 0);
    for (T i = 0, chk = 1;; i++, chk *= 2) {
        if (n <= chk) {
            return i;
        }
    }
}
int log2floor(int x) {
    return 31 - __builtin_clz(x);
}
int log2ceil(int x) { // 向上取整
    return log2floor(x) + (__builtin_popcount(x) != 1);
}
```

```cpp
template <class T> T sign(const T &a) {
    return a == 0 ? 0 : (a < 0 ? -1 : 1);
}
template <class T> T floor(const T &a, const T &b) { // 注意大数据计算时会丢失精度
    T A = abs(a), B = abs(b);
    assert(B != 0);
    return sign(a) * sign(b) > 0 ? A / B : -(A + B - 1) / B;
}
template <class T> T ceil(const T &a, const T &b) { // 注意大数据计算时会丢失精度
    T A = abs(a), B = abs(b);
    assert(b != 0);
    return sign(a) * sign(b) > 0 ? (A + B - 1) / B : -A / B;
}
```

最大公约数 `gcd`

欧几里得算法

速度不如内置函数！ 以 $\mathcal{O}(\log(a+b))$ 的复杂度求解最大公约数。与内置函数 `__gcd` 功能基本相同（支持 $a, b \leq 0$ ）。

```cpp
inline int mygcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

位运算优化

略快于内置函数，用于卡常。

```cpp
LL gcd(LL a, LL b) { // 卡常 gcd!!
    #define tz __builtin_ctzll
    if (!a || !b) return a | b;
    int t = tz(a | b);
    a >>= tz(a);
    while (b) {
        b >>= tz(b);
        if (a > b) swap(a, b);
        b -= a;
    }
    return a << t;
    #undef tz
}
```

整数域二分

自己用的

```cpp
auto l = l, r = r;
auto check = [&](auto x)->bool {

    };
while (l < r) {
```

```
    auto mid = l + r >> 1;
    if (check(mid))r = mid;
    else l = mid + 1;
}
```

旧版（无法处理负数情况）

- 在递增序列 a 中查找 $\geq x$ 数中最小的一个（即 x 或 x 的后继）

```
while (l < r) {
    int mid = (l + r) / 2;
    if (a[mid] >= x) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
return a[l];
```

- 在递增序列 a 中查找 $\leq x$ 数中最大的一个（即 x 或 x 的前驱）

```
while (l < r) {
    int mid = (l + r + 1) / 2;
    if (a[mid] <= x) {
        l = mid;
    } else {
        r = mid - 1;
    }
}
return a[l];
```

新版

- x 或 x 的后继

```
int l = 0, r = 1E8, ans = r;
while (l <= r) {
    int mid = (l + r) / 2;
    if (judge(mid)) {
        r = mid - 1;
        ans = mid;
    } else {
        l = mid + 1;
    }
}
return ans;
```

- x 或 x 的前驱

```
int l = 0, r = 1E8, ans = l;
while (l <= r) {
    int mid = (l + r) / 2;
    if (judge(mid)) {
        l = mid + 1;
        ans = mid;
    } else {
        r = mid - 1;
    }
}
return ans;
```

整体二分

```
int cal(auto x) {
    // todo
}

void solve(int ql, int qr, int l, int r) {
    if (ql > qr)return;
    if (l > r)return;
    if (l == r) {
        for (int q = ql;q <= qr;++q)ans[q] = l;
        return;
    }
    int mid = l + r + 1 >> 1;
    int cnt = cal(mid);
    solve(std::max(ql, cnt + 1), qr, l, mid - 1);
    solve(ql, std::min(qr, cnt), mid, r);
}

void solve() {
    //input
    solve(ql, qr, 0, n, zf);
    //todo
}
```

实数域二分

目前主流的写法是限制二分次数。

```
for (int t = 1; t <= 100; t++) {
    ld mid = (l + r) / 2;
    if (judge(mid)) r = mid;
    else l = mid;
}
cout << l << endl;
```

整数域三分

```
while (l < r) {
    int mid = (l + r) / 2;
    if (check(mid) <= check(mid + 1)) r = mid;
    else l = mid + 1;
}
cout << check(l) << endl;
```

实数域三分

限制次数实现。

```
ld l = -1E9, r = 1E9;
for (int t = 1; t <= 100; t++) {
    ld mid1 = (l * 2 + r) / 3;
    ld mid2 = (l + r * 2) / 3;
    if (judge(mid1) < judge(mid2)) {
        r = mid2;
    } else {
        l = mid1;
    }
}
cout << l << endl;
```

/END/