算法： 浮点数二分 讲义

1. 引言

- **问题引入**：我们都知道 √2 大约是 1.414，但这个结果是怎么算出来的？计算器按一个键就能得到答案，但它内部到底发生了什么？我们无法通过有限次的加减乘除直接得到这个无限不循环小数。这背后其实就用到了我们今天要学的思想——不断缩小范围，无限逼近。
- **价值阐述**：浮点数二分是解决**连续函数求根**、**实数域最优解**等问题的一大利器。它在科学与工程计算中扮演着至关重要的角色：
  - **方程求解**：求解 `sin(x) + x = 5` 这种无法直接解出的**超越方程**的根。
  - **物理模拟**：计算物体达到特定速度或位置所需的时间。
  - **几何计算**：在复杂图形中寻找满足特定几何属性（如面积、周长）的参数。
    它让我们能够用一种统一、简洁且高效的编程模型，来应对各种复杂的数学和工程问题。
- **学习目标**：
  1. 理解浮点数二分与整数二分的本质区别。
  2. 掌握基于**精度控制**的循环条件和边界更新方式。
  3. 学会使用“枚举+二分”的分治策略解决复杂的多根求解问题。
  4. 能够独立编写代码求解一元三次方程的根。

2. 核心思想

- **思路铺垫：手动计算 √2**
  回到 √2 的问题。我们不求精确命中，只求去逼近它。
  1. **确定范围**：我们知道 $1^2=1$ (太小了)，$2^2=4$ (太大了)。所以，√2 一定在 `[1.0, 2.0]` 这个区间里。
  2. **第一次猜测**：取区间中点 `1.5`。计算 `1.5² = 2.25`。嗯，还是太大了。说明 √2 肯定比 `1.5` 小。于是，答案范围被我们缩小到了 `[1.0, 1.5]`。
  3. **第二次猜测**：取新区间的中点 `1.25`。计算 `1.25² = 1.5625`。这次又太小了。说明 √2 比 `1.25` 大。答案范围进一步缩小到 `[1.25, 1.5]`。
  4. **继续...**：我们不断重复“**取中点 -> 判断 -> 缩小范围**”这个过程，区间的长度会以指数级速度减小，`left` 和 `right` 将会无限逼近那个真正的 √2。
- **浮点数二分与整数二分的三个核心区别**
  1. **目标不同**
     - **整数二分**：像是在一排有编号的格子里找东西，目标是**精确到**某个格子。
     - **浮点数二分**：像是在一条连续的线段上找一个点，由于浮点数无限可分，我们几乎不可能精确命中。我们的目标是**无限逼近**，直到我们的左右边界把这个点“夹”得足够紧，紧到我们无法分辨它们的区别为止。
  2. **循环条件不同：** `while (r - l > eps)` **vs.** `while (l <= r)`
     - 在整数世界，`l` 和 `r` 最终会“撞上”或“越过”对方，循环自然停止。
     - 在浮点数世界，`l` 和 `r` 可能永远不会完全相等。因此，我们引入一个极小值 `eps` (Epsilon，精度)，比如 `1e-8`。当区间的长度 `r - l` 已经比这个 `eps` 还要小的时候，我们就认为 `l` 和 `r` 已经“贴”在一起了，找到了答案。
  3. **边界更新不同：** `l = mid` **vs.** `l = mid + 1`

    - 整数二分中，如果 `mid` 不是答案，那下一个可能的答案至少是 `mid + 1`，我们可以放心地跳一格。
    - 浮点数二分中，如果 `mid` 不是答案，真正的答案可能就藏在 `mid` 和 `mid + 0.000...01` 之间。我们绝不能跳过任何一点点空间！因此，新的边界必须是 `mid` 本身，即 `l = mid` 或 `r = mid`。
- **浮点数二分模板**

```
double l, r;        // 答案所在的初始范围
const double eps = 1e-8; // 精度，题目要求k位小数，eps可设为1e-(k+2)

while (r - l > eps) { // 当区间长度还不够小时，继续二分
    double mid = l + (r - l) / 2; // 取中点
    if (check(mid)) { // check函数判断mid是否满足某种偏大的条件
        r = mid;      // 答案在左半边，或者mid就是答案
    } else {
        l = mid;      // 答案在右半边，或者mid就是答案
    }
}
// 循环结束时，l和r已经非常接近，输出任意一个均可
cout << l << endl;
```

3. 实例

- **具体案例：求解方程 $ax^3+bx^2+cx+d=0$**
  (B0193 / 书本例13-5)
  问题分析:
  这是一个一元三次方程。直接求解非常复杂。但题目给了我们几个关键的信息：
  1. **根的范围已知**：`[-100, 100]`。这直接给了我们一个初始的搜索大区间。
  2. **根的间距 ≥ 1**：它意味着，如果我们把 `[-100, 100]` 这条长长的数轴，切成 200 个长度为 1 的小区间 `[-100, -99]`, `[-99, -98]`, ..., `[99, 100]`，那么**每一个小区间里，最多只可能有一个根。**
  3. **零点存在性定理**：$f(x_1)*f(x_2) < 0$ 意味着 $f(x_1)$ 和 $f(x_2)$ 的函数值一个为正、一个为负。由于函数曲线是连续的，它必然会在这两者之间**穿过x轴**一次，也就是说，(x_1, x_2) 之间必有一个根。
- **算法策略：分治法 (枚举粗筛 + 二分精搜)**
  我们不能直接在 `[-100, 100]` 上二分，因为有三个根，天知道二分会逼近哪一个。正确的做法是：

  第一步：枚举粗筛
  我们用一个 `for` 循环，以 1 为步长，从 -100 扫描到 99。在每一步，我们检查 `[i, i+1]` 这个小区间。
    - 如果 `f(i)` 和 `f(i+1)` 的符号不同 (即 `f(i) * f(i+1) < 0`)，我们就锁定了一个藏有根的区间！

  第二步：二分精搜
  一旦我们找到了一个区间，比如 `[i, i+1]`，我们就知道这里面有且仅有一个根。现在问题变得简单了：在一个小区间内找一个唯一的根。我们立刻在这个小区间上套用浮点数二分模板，把这个根揪出来。
- **流程示例**
  1. **粗筛**：`for` 循环检查 `[-3,-2]`, `[-2,-1]`, `[-1,0]`...

2. 发现 `f(-2)` 是正的，`f(-1)` 是负的。`f(-2)*f(-1) < 0`！锁定区间 `[-2, -1]`。
3. **精搜**：在 `[-2, -1]` 内启动浮点二分，最终逼近根 `r1`。
4. `for` 循环继续，又发现 `f(0)*f(1) < 0`，锁定 `[0, 1]`，二分出 `r2`。
5. `for` 循环继续，发现 `f(2)*f(3) < 0`，锁定 `[2, 3]`，二分出 `r3`。

4. 代码实现

- **完整代码**

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

// 全局变量存储方程系数
double a, b, c, d;

// 定义方程函数 f(x)
double f(double x) {
    return a * x * x * x + b * x * x + c * x + d;
}

int main() {
    cin >> a >> b >> c >> d;

    // 设置输出格式为保留两位小数
    cout << fixed << setprecision(2);

    // 第一步：枚举粗筛
    for (int i = -100; i < 100; ++i) {
        double x1 = i;
        double x2 = i + 1;
        double y1 = f(x1);
        double y2 = f(x2);

        if (y1 == 0) {
            // 如果左端点恰好是根，直接输出
            cout << x1 << " ";
        }
        else if (y1 * y2 < 0) {
            // 如果区间内有根，启动第二步：二分精搜
            double l = x1, r = x2;
            const double eps = 1e-4; // 题目要求2位小数，1e-4的精度足够

            while (r - l > eps) {
                double mid = l + (r - l) / 2;
                // 判断根在左半边还是右半边
                // 如果 f(mid) 和 f(l) 同号，说明根在[mid, r]
                // 否则根在 [l, mid]
                if (f(mid) * f(l) > 0) {
                    l = mid;
                } else {
                    r = mid;
                }
            }
            cout << l << " ";
```

```
        }
    }
    return 0;
}
```

- **代码解读**:
  - `for (int i = -100; i < 100; ++i)`: 实现了从 -100 到 99 的步长为 1 的扫描。
  - `if (f(mid) * f(l) > 0)`: 这是判断根在哪一半区间的核心。如果 `f(mid)` 和 `f(l)` 同号（乘积为正），说明 `[l, mid]` 这个区间两端点函数值同号，根不在这里，所以根在另一半 `[mid, r]`，因此更新 `l = mid`。反之 `r = mid`。

5. 验证与深化

- **常见问题**
  1. **为什么 `eps` 要设为 `1e-(k+2)`？**
     这是一种经验法则。二分法在最后几次迭代时，精度提升会变慢。为了确保第 `k` 位小数的正确性，我们需要计算到比它更精确几位的程度，再进行四舍五入。多算两位通常是比较保险的选择。
  2. **二分判断条件 `if (f(mid) * f(l) > 0)` 可以换成 `if (f(mid) * f(r) > 0)` 吗?**
     可以，但逻辑要相应改变。如果 `f(mid)` 和 `f(r)` 同号，说明根在 `[l, mid]`，应更新 `r = mid`。只要保证逻辑自洽即可。
  3. **如果题目没有“根的间距>=1”这个条件怎么办?**
     问题会变得非常复杂。一个单位区间内可能有多个根，`f(i)*f(i+1) < 0` 这个条件就可能失效（比如有两个根，穿过去又穿回来，导致两端点同号）。这时需要借助微积分中的导数来判断函数的单调区间，在每个单调区间内再进行二分。

6. 总结

- **要点回顾**:
  1. **浮点数二分**的核心是**逼近**，而非查找。
  2. **循环条件**由精度 `eps` 控制：`while (r - l > eps)`。
  3. **边界更新**不加减1：`l = mid` 或 `r = mid`。
  4. 面对复杂问题，**分治思想**是关键。通过**枚举粗筛**将大问题分解为多个简单的、适合二分的小问题。
- **思考**:
  我们今天所有 `check` 的依据都是函数值的正负号。二分思想的应用远不止于此。任何一个参数，只要它能导致结果呈现出某种“单调性”（越大越如何，或越小越如何），我们都可以对这个参数进行二分,在整数和浮点数里是完全通用的。