# Link github: <https://github.com/lamtruongolpo123/bai_tap_kiemthuphanmem_baitapcanhan>

# I. VERIFICATION & VALIDATION

**What are the problems of those two systems?**  
(Dịch: Những vấn đề của hai hệ thống là gì?)

## Vấn đề của System 1

* **Công thức nghiệm bị sai**: trong code ghi x2 = -b - sqrt(DELTA/2a), nghĩa là -b - sqrt(Delta / (2\*a)), trong khi đúng phải là  
  x2 = (-b - sqrt(Delta)) / (2\*a).
* **Không có bước tính biệt thức** rõ ràng: không thấy phần tính Δ = b² − 4ac (giao diện của System 1 không hiển thị bước này).
* **Không xử lý các trường hợp ngoại lệ**: ví dụ a = 0 (không còn là phương trình bậc hai), Δ < 0 (nghiệm phức, cần báo hoặc xử lý), hoặc Δ = 0 (nghiệm kép).

## Vấn đề của System 2

* **Tính Δ đúng**, nhưng **công thức nghiệm có thể bị hiểu sai do thiếu dấu ngoặc**: dòng x2 = (-b - sqrt(DELTA))/2a nếu bị viết không đúng ngoặc có thể bị hiểu thành -b - (sqrt(DELTA) / (2\*a)) thay vì (-b - sqrt(DELTA)) / (2\*a).
* **Không xử lý trường hợp Δ < 0**: nếu không kiểm Δ trước khi gọi sqrt() sẽ gây lỗi hoặc giá trị không hợp lệ.
* **Không xử lý a = 0**: cần kiểm a == 0 để chuyển sang phương trình bậc hai hạng thấp (hoặc báo lỗi).
* Giao diện 2 bước (tính Δ rồi giải) là ý hay, nhưng cần **kiểm tra tính hợp lệ (validity check)** trước khi thực hiện sqrt.

# II. TEST-CASES (mỗi câu: số test-case + danh sách + giải thích)

## a) Hàm f1

int f1(int x) {

if (x > 10)

return 2 \* x;

else

return -x;

}

* **Số test-case cần**: **2**.
* **Test-cases gợi ý**:
  1. x = 11 → output 22 (nhánh x > 10).
  2. x = 10 → output -10 (nhánh else).
* **Giải thích**: 2 test này đảm bảo cả hai nhánh TRUE/FALSE được kiểm tra. (Nếu muốn kiểm biên thêm x = 9).

## b) Kiểm tra hàm f1 nếu bị cài sai như sau:

int f1(int x) {

if (x > 10)

return 2 \* x;

else if (x > 0)

return -x;

else

return 2 \* x;

}

* **Số test-case cần**: **3**.
* **Test-cases gợi ý**:
  1. x = 11 → 22 (giống bản đúng).
  2. x = 5 → -5 (giống bản đúng).
  3. x = -1 → khác nhau: bản đúng trả 1 (vì else trả -x trong bản đúng), bản sai trả -2 (do else cuối 2\*x) → **phát hiện lỗi**.
* **Giải thích**: cần test giá trị âm (hoặc 0) để phát hiện logic sai ở nhánh else.

## c) Hàm f2

int f2(int x) {

if (x < 10)

return 2 \* x;

else if (x < 2)

return -x;

else

return 2 \* x;

}

* **Số test-case cần**: **ít nhất 2**, nhưng để phát hiện lỗi logic cần chọn các giá trị biên.
* **Test-cases gợi ý**:
  1. x = 1 → 2 (nhánh if x < 10).
  2. x = 10 → 20 (nhánh else).
* **Giải thích**: else if (x < 2) **không bao giờ được xét** bởi vì mọi x < 2 đều thỏa x < 10 và đã bị chặn ở if đầu — nghĩa là điều kiện thứ hai vô dụng. Cần sửa thứ tự điều kiện hoặc logic.

## d) Hàm f3

int f3(int x) {

if (log(x \* x \* cos(x)) < 3 \* x)

return 2 \* x;

else

return 2 \* x;

}

* **Số test-case cần**: **1** (vì cả hai nhánh trả cùng giá trị).
* **Test-case gợi ý**: x = 1 → 2.
* **Giải thích**: cả hai nhánh đều return 2\*x → tức là biểu thức điều kiện không ảnh hưởng kết quả: đây là **lỗi logic / code thừa**, cần sửa yêu cầu hoặc giá trị trả ở một nhánh.

## e) Hàm findMax

int findMax(int num1, int num2, int num3) {

int max = 0;

if ((num1 > num2) && (num1 > num3))

max = num1;

if ((num2 > num1) && (num2 > num3))

max = num2;

if ((num3 > num1) && (num3 > num2))

max = num3;

return max;

}

* **Vấn đề / lỗi**:
  + max = 0 khởi tạo có thể sai nếu tất cả ba số âm (kết quả trả về sai).
  + Dùng > thay vì >= khiến trường hợp có giá trị lớn nhất trùng nhau (equal) không được xử lý → max có thể không cập nhật.
  + Các if độc lập không sai về cấu trúc, nhưng logic so sánh hiện tại không bao hàm trường hợp bằng nhau.
* **Số test-case tối thiểu để phát hiện lỗi**: **≥ 5** (gợi ý):
  + (3, 2, 1) → mong muốn 3.
  + (1, 4, 2) → mong muốn 4.
  + (1, 2, 5) → mong muốn 5.
  + (5, 5, 3) → mong muốn 5 (kiểm trường hợp trùng nhau).
  + (-3, -1, -2) → mong muốn -1 (kiểm số âm).
* **Sửa đề xuất**: khởi tạo max = num1; rồi if (num2 > max) max = num2; if (num3 > max) max = num3; hoặc sử dụng >= tùy yêu cầu.

# III. PRACTICE 1 — solveQuartic (mô tả, test-cases, mã kiểm tra)

## 1) Mô tả bài toán (ngắn gọn)

Bài toán: Tìm nghiệm thực của phương trình dạng bậc 4 (dạng biquadratic):  
a \* x^4 + b \* x^2 + c = 0.

* **Input**: a, b, c (double).
* **Output**: số nghiệm thực n (trả về -1 nếu vô số nghiệm), và ghi các nghiệm thực vào mảng x[].
* Quy ước:
  + Trả -1 nếu a == 0 && b == 0 && c == 0 (vô số nghiệm).
  + Trả 0 nếu không có nghiệm thực.
  + Trả n (1..4) nếu có n nghiệm thực; ghi nghiệm vào x[0..n-1].
* Khi a == 0 xử lý thành b \* x^2 + c = 0 (phương trình bậc hai chuẩn cho x^2).

## 2) Test-case cần thiết (đủ bao phủ các trường hợp)

Danh sách test-case (input, expected return, expected roots):

1. a=0, b=0, c=0
   * Expected: -1 (Infinite solutions).
2. a=0, b=0, c=1
   * Expected: 0 (No solution).
3. a=0, b=1, c=-4 (b \* x^2 + c = 0 → x^2 = 4)
   * Expected: 2, roots {2, -2}.
4. a=1, b=0, c=1 (y^2 + 1 = 0 → Δ < 0)
   * Expected: 0.
5. a=1, b=0, c=-1 (y1 = 1 → x = ±1)
   * Expected: 2, roots {1, -1}.
6. a=1, b=-5, c=4 (y^2 - 5y + 4 = 0 → y = 4,1 → x = ±2, ±1)
   * Expected: 4, roots {2, -2, 1, -1}.
7. a=1, b=-2, c=1 (y^2 - 2y +1 = 0 → y = 1 (double) → x = ±1)
   * Expected: 2, roots {1, -1}.

## 3) Mã kiểm tra tự động (C++) — Test harness

(Copy đoạn sau vào file .cpp; cần đặt định nghĩa solveQuartic trong cùng file hoặc link tương ứng.)

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

// Prototype of the tested function (assume it's implemented as in the PDF)

int solveQuartic(double a, double b, double c, double x[]);

// Helper: compare doubles with tolerance

bool approxEqual(double a, double b, double eps = 1e-6) {

return fabs(a - b) <= eps;

}

// Helper: check if expected roots are present in result (order-independent)

bool checkRoots(const vector<double>& got, const vector<double>& expected, double eps = 1e-6) {

if (got.size() != expected.size()) return false;

vector<bool> used(expected.size(), false);

for (double g : got) {

bool found = false;

for (size\_t j = 0; j < expected.size(); ++j) {

if (!used[j] && approxEqual(g, expected[j], eps)) {

used[j] = true;

found = true;

break;

}

}

if (!found) return false;

}

return true;

}

void runTest(int id, double a, double b, double c, int expReturn, const vector<double>& expRoots) {

double x[10];

int ret = solveQuartic(a, b, c, x);

vector<double> gotRoots;

if (ret > 0) {

for (int i = 0; i < ret; ++i) gotRoots.push\_back(x[i]);

}

bool okReturn = (ret == expReturn);

bool okRoots = true;

if (expReturn > 0) okRoots = checkRoots(gotRoots, expRoots);

if (expReturn <= 0) okRoots = true; // for -1 or 0 we only check return code

cout << "Test " << id << ": a="<<a<<" b="<<b<<" c="<<c<<" => ret="<<ret;

if (!okReturn) cout << " [WRONG return, expected " << expReturn << "]";

else cout << " [OK return]";

if (expReturn > 0) {

if (okRoots) cout << " roots OK";

else {

cout << " roots WRONG; got {";

for (double v : gotRoots) cout << v << " ";

cout << "} expected {";

for (double v : expRoots) cout << v << " ";

cout << "}";

}

}

cout << endl;

}

int main() {

// Place your solveQuartic implementation in the same project or link accordingly.

runTest(1, 0, 0, 0, -1, {}); // infinite

runTest(2, 0, 0, 1, 0, {}); // no solution

runTest(3, 0, 1, -4, 2, {2.0, -2.0}); // x^2 = 4 -> ±2

runTest(4, 1, 0, 1, 0, {}); // no real y

runTest(5, 1, 0, -1, 2, {1.0, -1.0}); // y1=1 -> ±1

runTest(6, 1, -5, 4, 4, {2.0, -2.0, 1.0, -1.0}); // y=4,1 -> ±2, ±1

runTest(7, 1, -2, 1, 2, {1.0, -1.0}); // double y, only two roots

return 0;

}

**Ghi chú:**

* solveQuartic phải được định nghĩa như trong đề (kiểm tra a==0, tính Δ cho y = x^2, tránh sqrt số âm, tránh lặp nghiệm).
* So sánh nghiệm không phụ thuộc thứ tự trả về (hàm checkRoots kiểm tra tập nghiệm).

## 4) Nhận xét kiểm thử & khuyến nghị (để nộp kèm)

* Luôn kiểm tra các **trường hợp biên**: a=0, b=0, c=0; Δ < 0; nghiệm trùng; nghiệm âm cho y (không cho sqrt).
* Nên **sắp xếp nghiệm** (ví dụ tăng dần) trước khi trả để thuận tiện so sánh tự động.
* Với trường hợp **vô số nghiệm**, dùng giá trị trả đặc biệt (-1) và mô tả rõ trong tài liệu.
* Viết thêm test cho nghiệm trùng để đảm bảo không đếm trùng nhiều lần.

**test\_solver.cpp**

#include <iostream>

#include <cmath>

#include <cassert>

using namespace std;

/\*

   Giải phương trình bậc 4 dạng: a\*x^4 + b\*x^2 + c = 0

   (dạng thu gọn thường gặp trong môn học kiểm thử phần mềm)

\*/

int solveQuartic(double a, double b, double c, double x[]) {

    if (a == 0 && b == 0 && c == 0) {

        return -1; // vô số nghiệm

    }

    if (a == 0 && b == 0) {

        return 0; // vô nghiệm

    }

    if (a == 0) { // b\*x^2 + c = 0

        double y = -c / b;

        if (y < 0) return 0;

        x[0] = sqrt(y);

        x[1] = -sqrt(y);

        return 2;

    }

    double delta = b\*b - 4\*a\*c;

    if (delta < 0) return 0;

    double y1 = (-b + sqrt(delta)) / (2\*a);

    double y2 = (-b - sqrt(delta)) / (2\*a);

    int count = 0;

    if (y1 >= 0) {

        x[count++] = sqrt(y1);

        x[count++] = -sqrt(y1);

    }

    if (y2 >= 0 && y2 != y1) {

        x[count++] = sqrt(y2);

        x[count++] = -sqrt(y2);

    }

    return count;

}

// ================== TEST CASES ==================

void test\_cases() {

    double x[4];

    int n;

    // Test 1: Vô số nghiệm

    n = solveQuartic(0,0,0,x);

    assert(n == -1);

    // Test 2: Vô nghiệm

    n = solveQuartic(0,0,5,x);

    assert(n == 0);

    // Test 3: bậc 2 rút gọn: x^2 = 4 => nghiệm ±2

    n = solveQuartic(0,1,-4,x);

    assert(n == 2);

    // Test 4: bậc 4 có nghiệm: x^4 - 5x^2 + 4 = 0

    n = solveQuartic(1,-5,4,x);

    assert(n == 4);

    // Test 5: vô nghiệm do delta < 0

    n = solveQuartic(1,0,1,x);

    assert(n == 0);

    cout << "✅ All test cases passed!" << endl;

}

int main() {

    test\_cases();

    return 0;

}

**run\_test.ipynb**

# Biên dịch code C++ thành chương trình thực thi

!g++ test\_solver.cpp -o solver

# Chạy chương trình kiểm thử

!./solver