ALGORITHM 1

```cpp
#include <algorithm>
#include <cmath>

double GetTimeToTargetRGivenInitialSpeed(double k, double vInfinity, double rX, double double rY) {
    // 1. Start by getting coefficients for the function f(t) = a0*t^4 + a1*t^3 + a2*t^2 + a3*t + a4
    // + a5*t + a6 which is 0 at the sought time-to-target t. Solving f(t) = 0
    // for t > 0 is equivalent to solving e(u) = f(1/u)/u^4 = a0u^4 + a1u^3 + a2u^2 + a3u + a4
    // + a5/u + a6 = 0 for u where u = 1 / t, but the latter is more well-behaved,
    // being a strictly concave function for u > 0 for any set of valid inputs,
    // so solve e(u)=0 for u instead by converging from an upper bound towards
    double kVInfinity = k * vInfinity; // the root and
    double a0 = -kVInfinity; a1 = a0 * k; a2 = k * kVInfinity; rR = kVInfinity; // return 1/u.
    double a4 = kVInfinity * kVInfinity;
    double maxInvRelError = 1.0E6; // Use an achievable inverse error bound.
    double de, e, uDelta = 0;
    // 2. Set u to an upper bound by solving e(u) with a3 = a1 = 0, clamped by
    // the result of a Newton method's iteration at u = 0 if positive.
    double u = std::sqrt(kVInfinity / std::sqrt(rR));
    if (rY < 0) u = std::min(u, -vInfinity / rY);
    // 3. Let u monotonically converge to e(u)'s positive root using a modified
    // Newton's method that speeds up convergence for double roots, but is likely
    // to overshoot eventually. Here, 'e' = e(u) and 'de' = de(u)/du.
    for (int it = 0; it < 10; ++it, uDelta = e / de, u -= 1.9 * uDelta) {
        // d(e)/du = de = e = e + u + e; e = e + u + e; e = e + u + e;
        break; // Overshot the root.
    }
    u += 0.9 * uDelta; // Trace back to the unmodified Newton method's output.
    // 4. Continue to converge monotonically from the overestimated u to e(u)'s
    // only positive root using Newton's method.
    for (int it = 0; it <= 10; ++it) {
        de = e = de + e; e = e + u + e; e = e + u + e;
        uDelta = e / de; u = uDelta;
    }
    // 5. Return the solved time t to hit [rX, rY], or 0 if no solution exists.
    return u > 0 ? 1 / u : 0;
}
```

ALGORITHM 2

```cpp
#include <algorithm>
#include <cmath>

double GetTimeToTargetRGivenInitialSpeed(double k, double vInfinity, double rX, double double rY, double k, bool highArc) {
    // 1. Start by getting coefficients for the function f(t) = a0*t^4 + a1*t^3 + a2*t^2 + a3*t + a4
    // + a5*t + a6 which is 0 at the sought time-to-target t. Solving f(t) = 0
    // for t > 0 is equivalent to solving e(u) = f(1/u)/u^4 = a0u^4 + a1u^3 + a2u^2 + a3u + a4
    // + a5/u + a6 = 0 for where u = 1 / t, but the latter is more well-behaved,
    // being a strictly convex function for u > 0 for any set of inputs,
    // so inputs iff a solution exists, so solve for e(u) = 0 instead by converging
    // from a high or low bound towards the closest root and return 1/u.
    double kRXkRy = k * rX; kK = k * k; kRXkRy = kRXkRy; sS = sS * sS;
    double maxV0YSq = sS - kRXkRy; kVInfinity = k * vInfinity; // double double a0 = -kVInfinity; a1 = a0 * k; a3 = k * kVInfinity; de, e, uDelta = 0;
    // double maxInvRelError = 1.0E6; // Use an achievable inverse error bound.
    double maxV0YSq = sS - kRXkRy; /maxV0YSq is the max squared 'v0.y' that leaves
double e, de, u, uDelta = 0; // enough 'v0.y' to reach RX horizontally.
    // 2. Set u to a lower/upper bound for the high/low arc, respectively.
    if (highArc) { // Get smallest u vertically moving rY at max possible +v0.y.
        double minusUa = std::sqrt(maxV0YSq) + kRXkRy;
        double determ = std::sqrt(minusUa * minusUa) - (twoKVInfinityRY + twoKVInfinityRY); // Convergence over negative slopes.
    } else if (rY < 0) { // Get largest u vertically moving rY at most neg. v0.y.
        double minusUa = std::sqrt(maxV0YSq) - kRXkRy;
    }
    // Clamp the above bound by the largest u that reaches RX horizontally.
    u = std::min(s / rX - k, u);
    else u = u / std::sqrt(a0 - k); // Get the (largest) u hitting RX
    // horizontally a.s.a.p. while launching in the direction of [rX, rY].
    // 3. Let u monotonically converge to e(u)'s closest root using a modified
    // Newton's method, almost scaling the delta as if the solution is a double
    // root. Note that 'e' = e(u) * u^2 and 'de' = de(u)/du * u^2.
    for (; it < 12; ++it, uDelta = e / de, u -= 1.9 * uDelta) {
        de = e = de + e; de = de + e; e = e + u + e; de = de * u + e; de = de * u + e; de = de * u + e;
        break; // Overshot.
    }
    u += 0.9 * uDelta; // Trace back to unmodified Newton method's output.
    // 4. Continue to converge monotonically to e(u)'s closest root using
    // Newton's method from the last known conservative estimate on the convex
    // function. (Note that in practice, u will have converged enough in c2
    // for (u > 0 && u = c2 + it) { // iterations iff a solution does exists.
    // de = e = e + u + e; de = de * u + e; de = de * u + e; de = de * u + e; de = de * u + e;
    // uDelta = e / de; u = uDelta;
    // if ((de > maxInvRelError) > 0) break; // wrong side of the convex 'dip'.
    // if (uDelta > maxInvRelError < u && u > 0) return 1 / u; // 5a. Found it!
    // }
    // 5b. If no solution was found, return 0. This only happens if s (minus
    // or small epsilon) is too small to have a solution, the target is at the
    // origin, or the parameters are so extreme they cause overflows.
    return 0;
}
```