CTU Open 2015

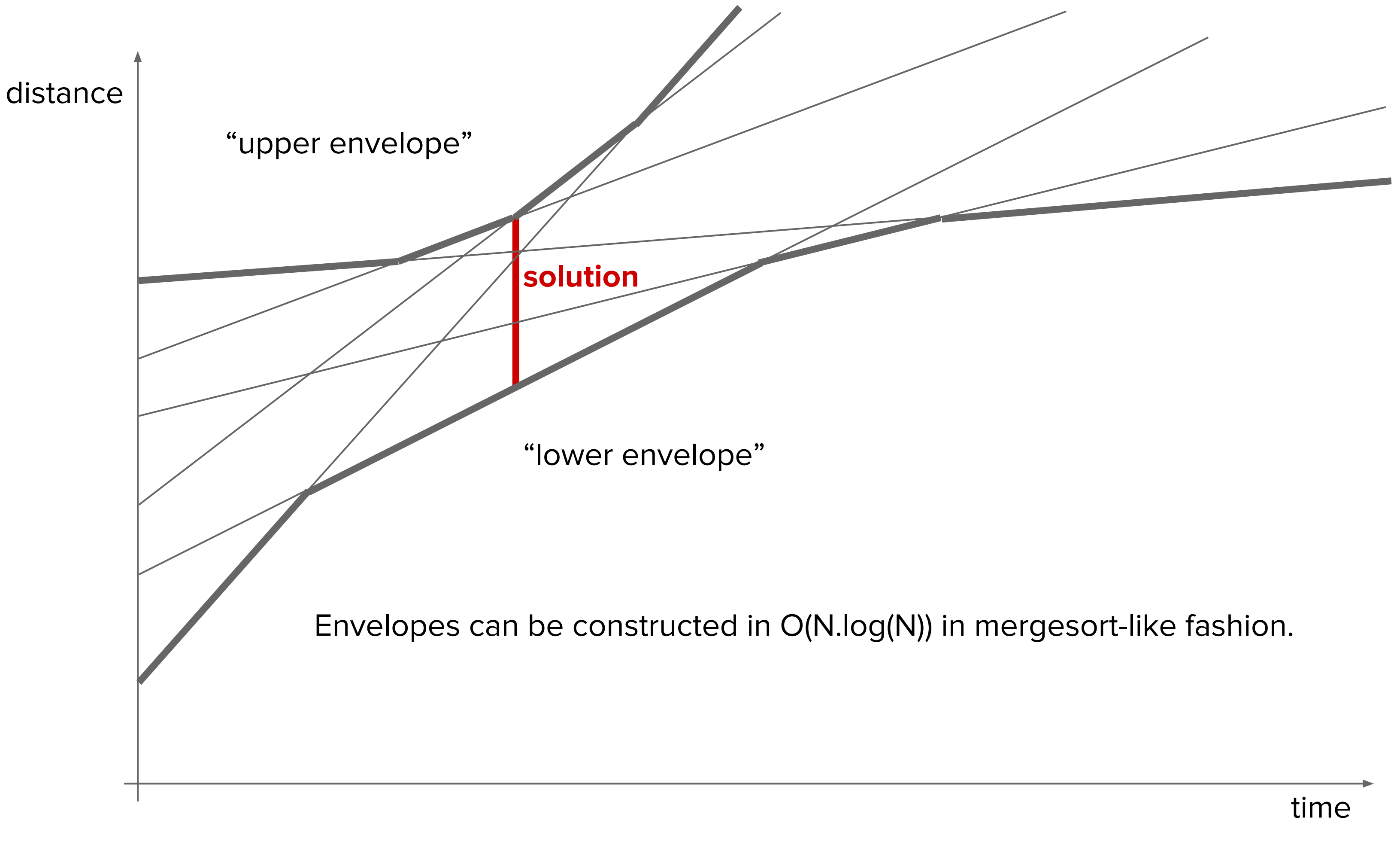
Solutions Discussion

Chasing the Cheetahs

Chasing the Cheetahs

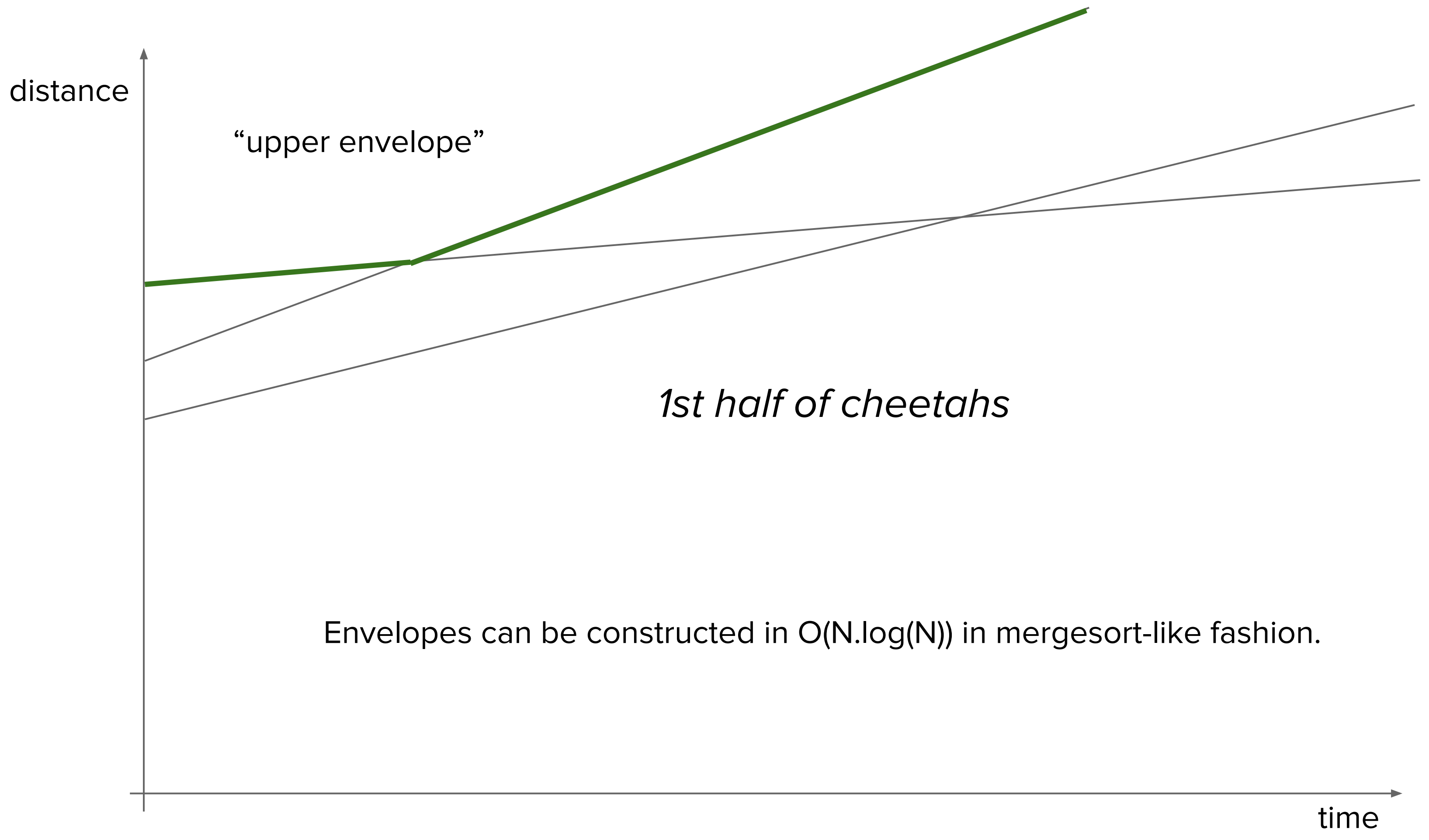
- **Goal:** Determine the shortest distance between the first and the last cheetah during the race.
- **Idea:** Build a *list of cheetahs running first* (at any time) during the race (“*upper envelope*”) and keep the times in which they outrun each other, build a similar *list for the last cheetahs (“lower envelope”).* Then, **find the shortest distance between the envelopes** - the times at which the distance must be considered is 0 and any time a cheetah is outrun by another. This allows for O(N.log(N)) solution, see the example on the next slides.

Chasing the Cheetahs [cont'd]



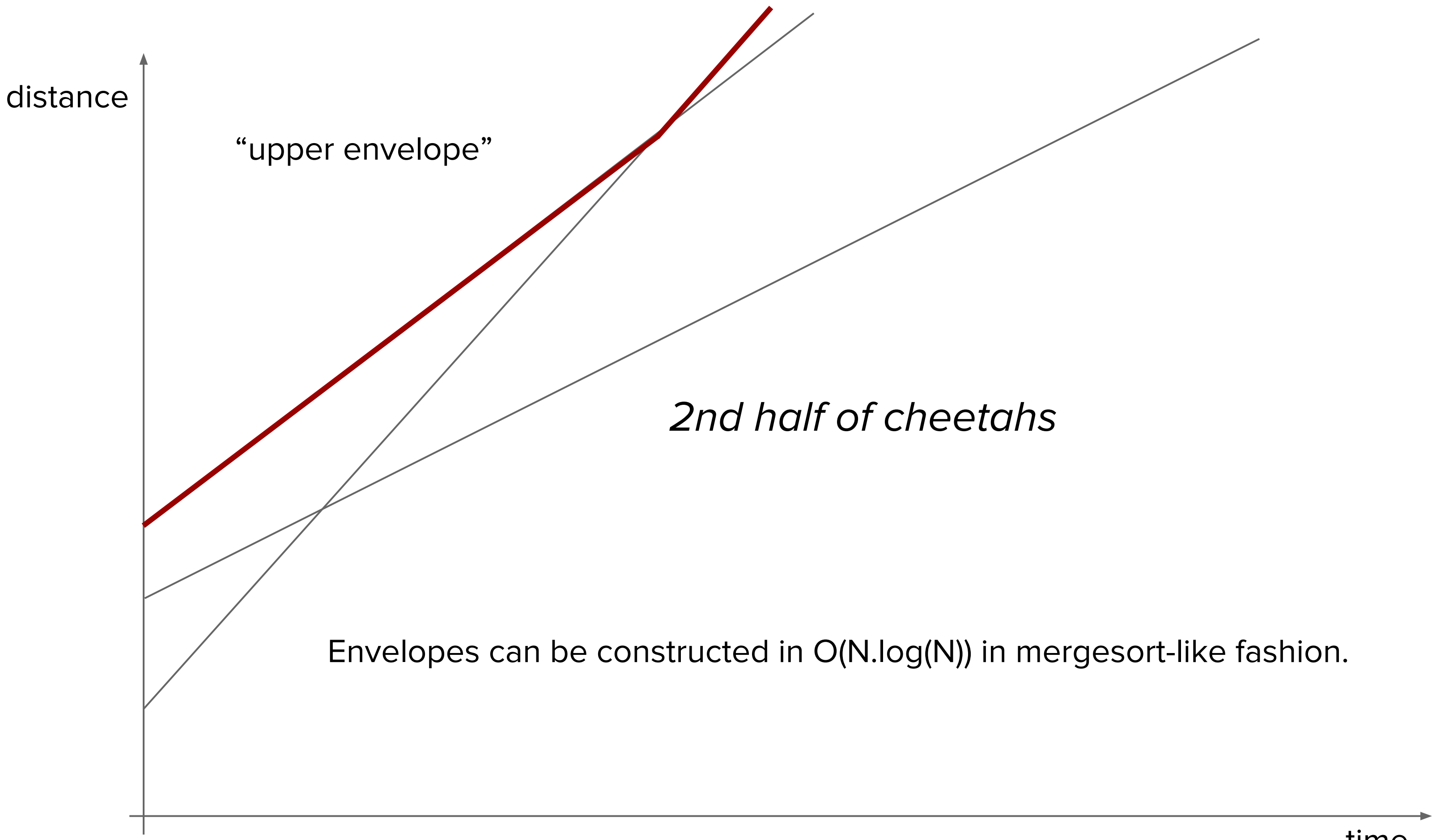
Envelopes can be constructed in O(N.log(N)) in mergesort-like fashion.

Chasing the Cheetahs [cont'd]



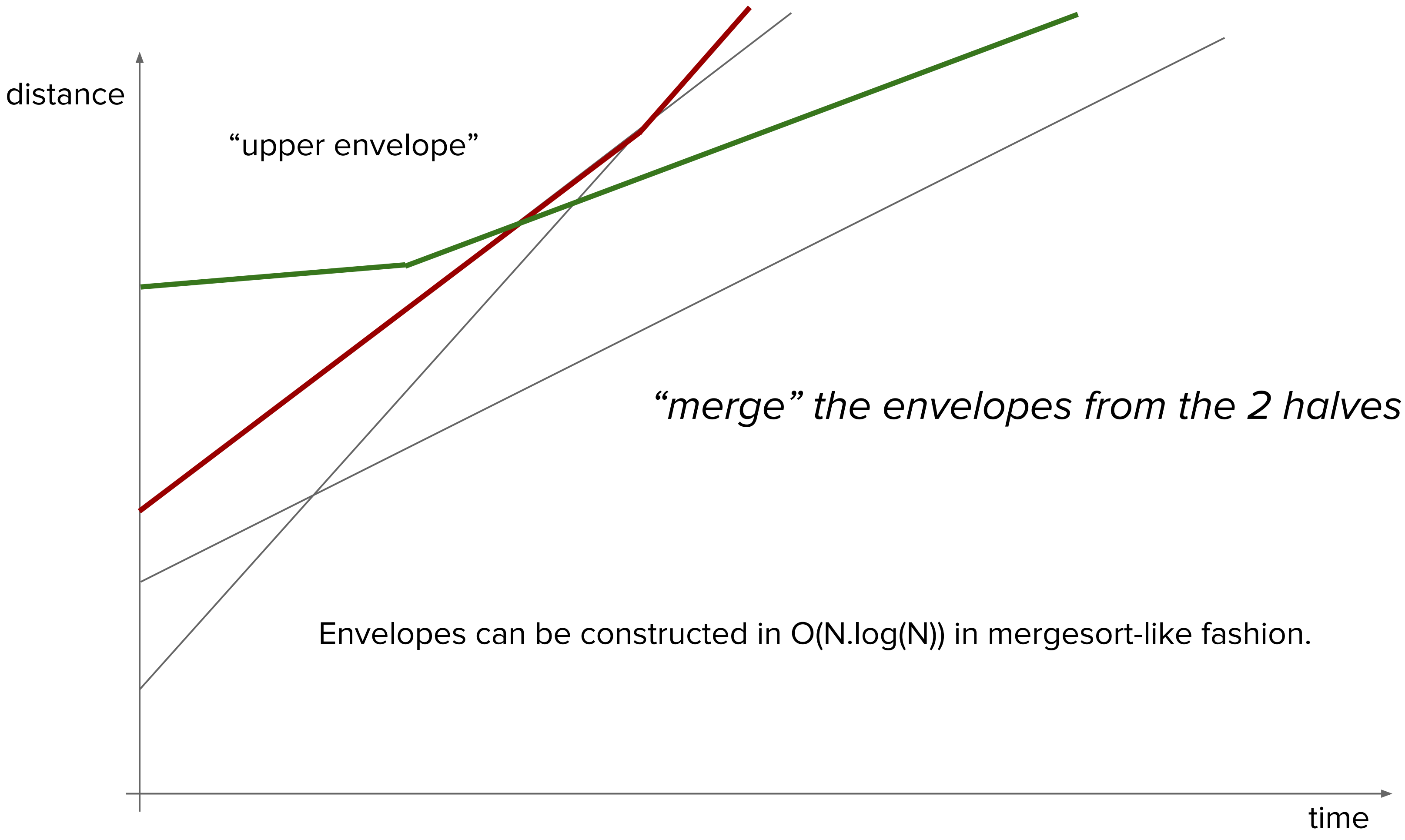
Envelopes can be constructed in O(N.log(N)) in mergesort-like fashion.

Chasing the Cheetahs [cont'd]



Envelopes can be constructed in O(N.log(N)) in mergesort-like fashion.

Chasing the Cheetahs [cont'd]

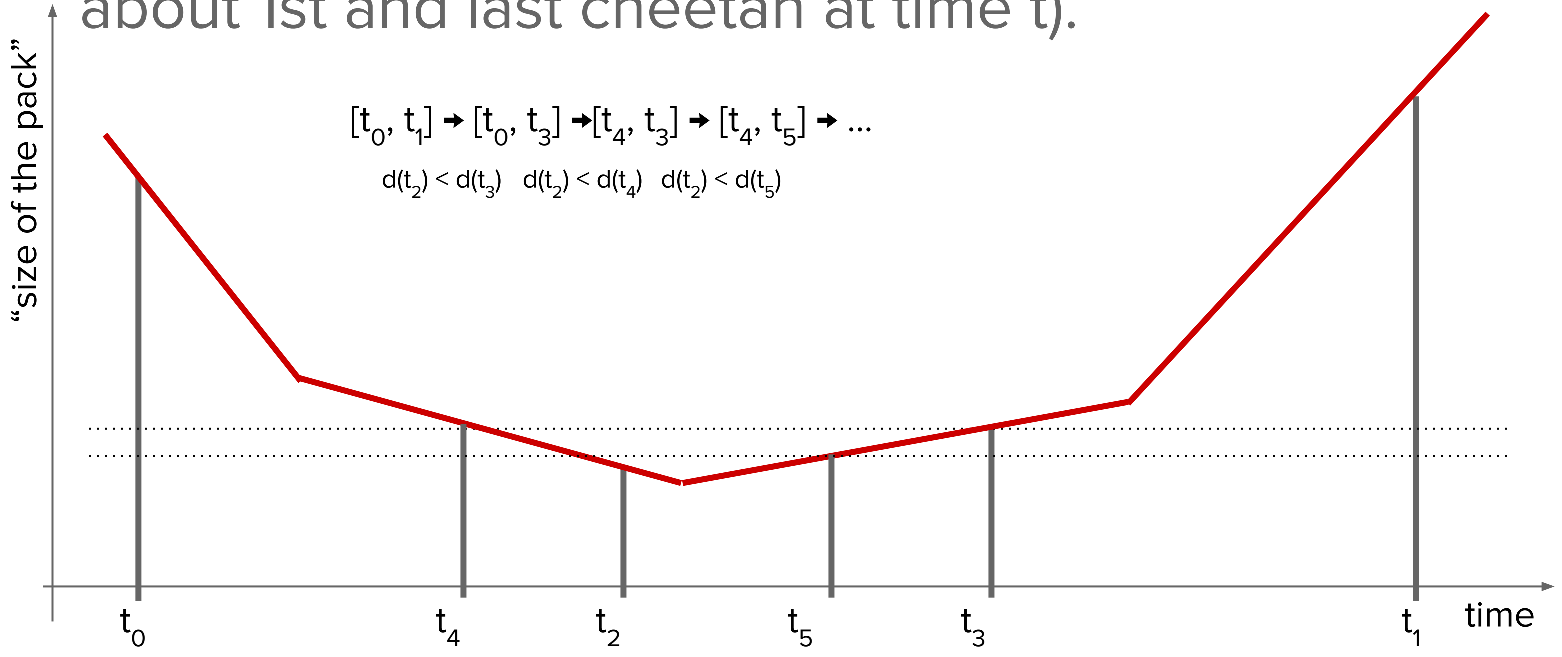


"merge" the envelopes from the 2 halves

Envelopes can be constructed in O(N.log(N)) in mergesort-like fashion.

Chasing the Cheetahs [alternative solution]

- **Idea:** The distances between the envelopes as a function of time, **d(t)**, is convex linear-fractional function. This function is unimodal and we can use **ternary search method** (binary search works with additional information about 1st and last cheetah at time t).



Chasing the Cheetahs [alternative solution]

```
be = “time the last cheetah runs out”
en = “upper bound on the duration of the race”

for (step = 0; step < ENOUGH; ++step):
    t1 = (2 * be + en) / 3
    t2 = (be + 2*en) / 3
    v1 = d(t1)
    v2 = d(t2)
    if v1 <= v2:
        en = t2
    else
        be = t1
```

Falcon Dive

Falcon Dive

- **Goal:** Extrapolate the image of the diving falcon from two pictures taken in two subsequent moments in time.
- **Ideal:** Locate the top-left pixel of the falcon's silhouette in both images and compute the translation vector from it. The background can be reconstructed easily (silhouettes do not overlap) by taking pixels from any image if it's not part of the falcon or taking it from the other image if it is. Then, given the position of pixels of the falcon in the 1st image it is just a matter of adding twice the translation vector to it and putting the pixel on it in the output image.

The Fox and the Owl

The Fox and the Crow

- **Goal:** For given integer N (huge!) find the largest integer smaller than N with sum of its digits larger by 1 than the sum of digits of N.
- **Idea:** Implementational problem. Needed careful case analysis and implementation.
  - -99999 → -199999
  - -12**3**999 → -12**4**999
  - **104899** → 99950: Sum of the 4 **green** digits + 2 (LHS) is below 4 * 9 (do not forget to skip **0**s), decrease **red** digit (discard it if it becomes 0) and fill the rest of the digits to the right of it by { min(LHS, 9); LHS -= 9; } while LHS > 0;

The Fox and the Crow

- 4899 ➜ -4999
- 9999 -> -19999: using our rule we hit the left end, in that case we need to *distribute* the sum of digits + 1 (modulo 9) to the left of the negative sign, and possibly, fill the rest of the number by 9s.
- 9998 ➜ -9999

Feeding the Herrings

Feeding the Herrings

- **Goal:** Determine how many combinations (a, b, c) exist, such that a, b, c are at least L and a + b + c equals N. Moreover, there *must not* be 3 in the digits of a, b, c.
- **Idea:** The size of numbers (obviously) suggest an approach based on representation of the number a, b, c as arrays of digits + **dynamic programming**:
    - Let’s consider the case when L = 0:
        - **comb[i][c]** - “# of combinations *correct* (a, b, c) which add up to N[i..len(N)] with carry c.
        - Fill in the table **comb[i][c]** (next slide).
        - Print the answer **comb[0][0]**.

Feeding the Seals [cont'd]

- Code snippet for filling the table **comb[i][c]**:

```
int solve(i, c):
    if (comb[i][c] != -1) return comb[i][c]
    if (i == len(N)):
        ans = (c == 0) ? 1 : 0
    else:
        ans = 0
        for a in [0, 1, 2, 4, ..., 9]:  // Try all allowed digits for (i + 1)-th place of a.
            for b in [0, 1, 2, 4, ..., 9]:
                for c in [0, 1, 2, 4, ..., 9]:
                    c2 = 10 * c + N[i] - a - b - c
                    if (0 <= c2 <= 2):
                        ans = (ans + solve(i + 1, c2)) % MOD
    comb[i][c] = ans
    return comb[i][c]
```

NOTE: The for-loops in the solution when L > 0 must use the lower limit induced by **L** and the table must reflect that!

NOTE: Check for yourself the allowed values for the carry **c**.

Jumping Yoshi

Jumping Yoshi

- **Goal:** Find out the furthest pebble Yoshi can jump to while satisfying the given *rule*.
- **Idea:** Beat the naive $O(N^2)$ solution by the following observation:

    rule: pebble[i] + pebble[j] == distance[j] - distance[i]

  for every ***oriented* edge (i, j)**, which implies:

    distance[i] + pebble[i] == distance[j] - pebble[j]

  so by building 2 lists of pebbles for each **n in [0... N-1]**:

    out[n] = [i | distance[i] + pebble[i] == n]

    in[n] = [j | distance[j] - pebble[j] == n]

Jumping Yoshi [cont'd]

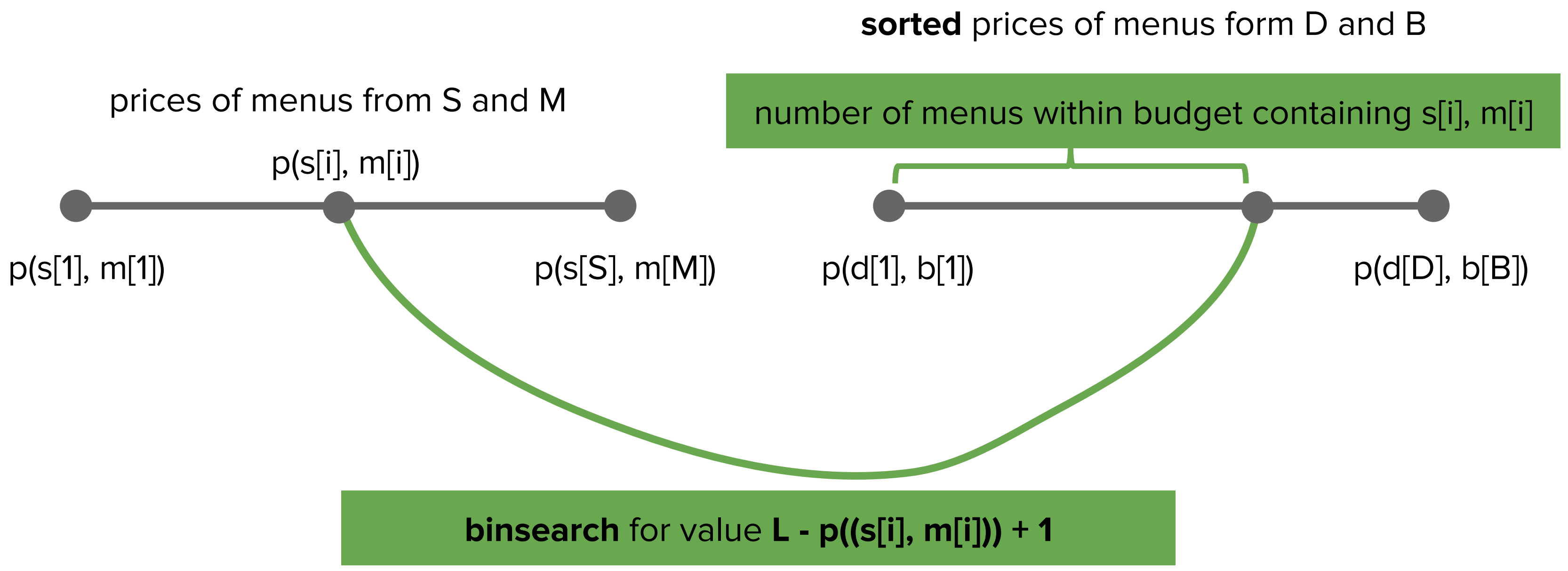
- Now, start **BFS** from **0** (1st pebble) on implicitly defined edges by the two lists **in** and **out** (ignore orientation!):

```
Q = {0}
while not Q.empty():
    i = Q.pop()
    if distance[i] + pebble[i] < N:
        for j in in[distance[i] + pebble[i]]: // check the edges (i, j)
            if not visited[j]: Q.push(j), visited[j] = True

    if distance[i] - pebble[i] >= 0:
        for j in out[distance[i] - pebble[i]]: // check the edges (j, i)
            if not visited[j]: Q.push(j), visited[j] = True

answer = max{i | visited[i] == True}
```

Lunch Menu

Lunch Menu

- **Goal:** Count how many quadruples (s, m, d, b) - “menus” - can be made with a price under the given budget.
- **Idea:** Beat the naive $O(N^4)$ solution by sorting and using binary search:


sorted prices of menus form D and B
prices of menus from S and M
number of menus within budget containing s[i], m[i]
p(s[i], m[i])
p(s[1], m[1])
p(s[S], m[M])
p(d[1], b[1])
p(d[D], b[B])
binsearch for value **L - p((s[i], m[i])) + 1**

The Owl and the Fox

The Crow and the Fox

- **Goal:** For given integer N (small!) find the largest integer smaller than N with the sum of its digits ***smaller*** by 1 than the sum of the digits of N.
- **Idea:** The size of N permitted the use of **brute-force**:

```
while(--N > 0 and digitsum(N) + 1 != dsOfN):
    continue

int digitsum(N):
    sum = 0
    while (N > 0): sum += N % 10, N /= 10
    return sum
```

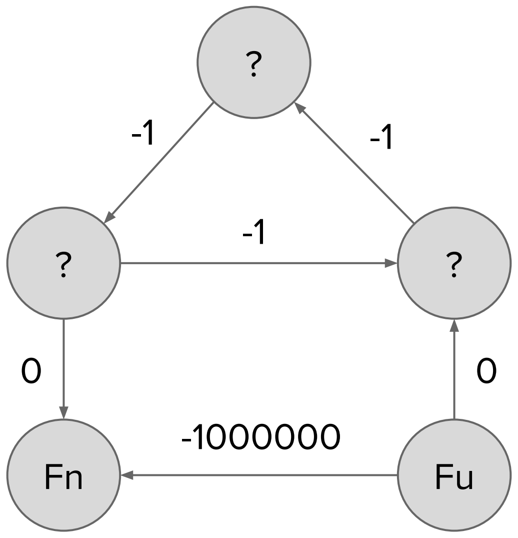
Plankton Food

Plankton Food

- **Goal:** Determine whether there is a negative cycle reachable from the start vertex (unnecessary food) to the destination vertex (necessary food).
- **Idea:** Run **Bellman-Ford** algorithm for T iterations and mark the vertices whose *distance* from the start *changed in the last iteration* - each of these lies on a negative cycle. Then, (with edges reversed) run **BFS/DFS** from the target and see whether you can get to a marked vertex -- the answer is TRUE, or not -- FALSE.

Plankton Food [cont'd]

- **BEWARE:** You might tried running the Bellman-Ford algorithm for (T - 1) iterations, keeping the distances, running it one more time for another T iterations and finally checking whether the distance in the target has changed, but this does not work:



Hacking the Screen

Hacking the Screen

- **Goal:** Evaluate the given *simple* arithmetic expression.
- **Idea:** Pure implementational problem:
  - Implement subroutine evaluating SIMPLE expressions

    eval(char *exp, int s, int e)

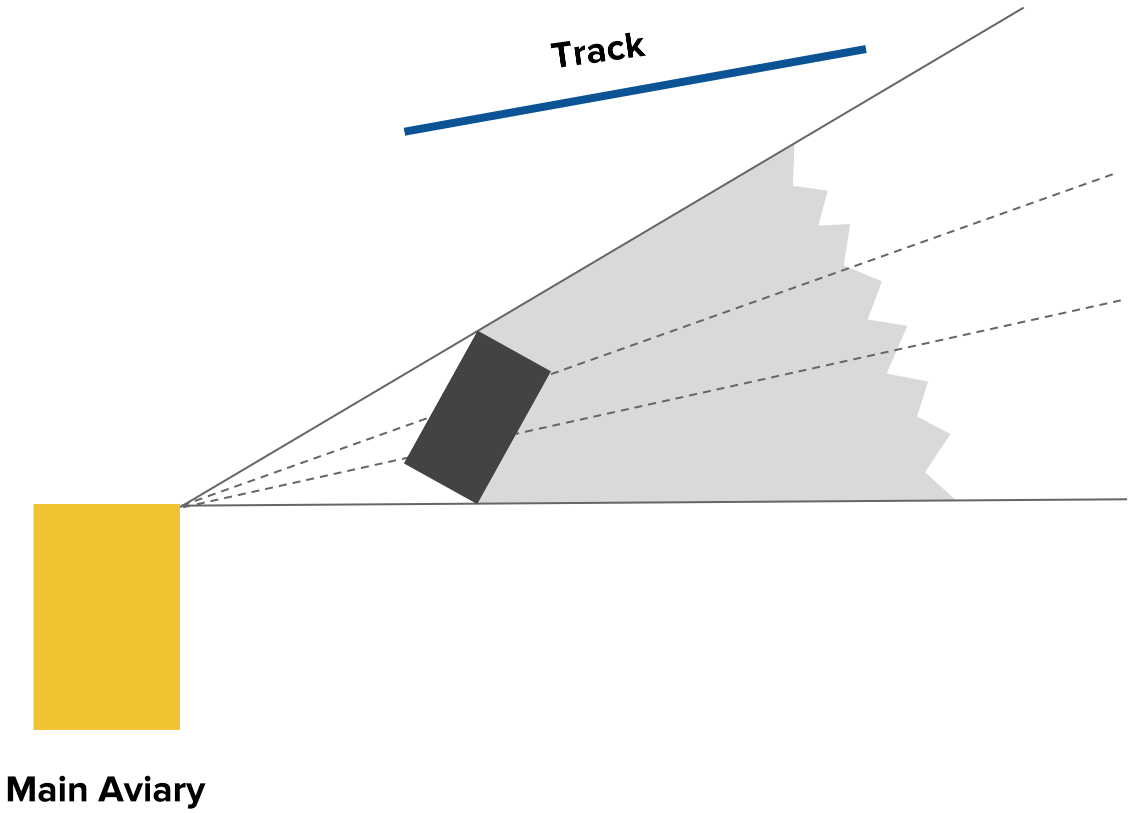
    which evaluates an expression starting/ending in the string **exp** at **s**/**e**.
  - Use the number of ‘**-**’ (‘=’) when you hit ‘\’ (first ‘=’) to find **s**/**e** for a square root (fraction) sub-expression.
  - Build a *simple expression* by substituting results from sub-expressions and call **eval** on the whole thing.

Visitors' Train

Visitors' Train

- **Goal:** Find the total length of the segments on the track from which the main aviary is visible and is not obscured, not even partially, by any other aviary.
- **Idea:** Utilize algebraic primitives, such as *cross product* (to determine angles) and *Cramer's rule* (determine points of intersection), to compute the "shade" casted on the track. Think hard over the possible track/aviaries layouts and make sure your solution works for a more "troublesome" compositions.

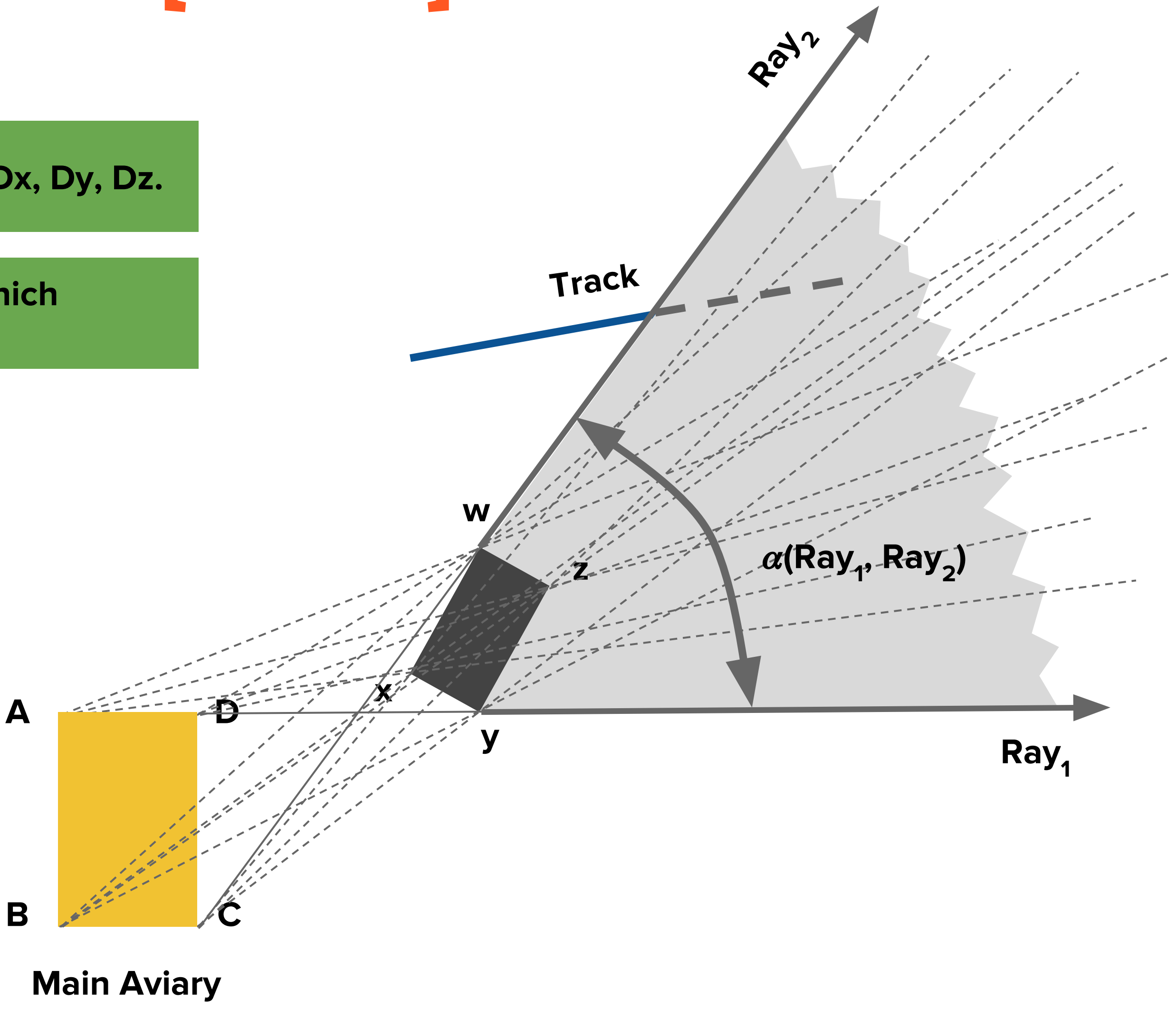
Visitors' Train [cont'd]



Visitors' Train [cont'd]

16 lines Aw, Ax, Ay, …, Dx, Dy, Dz.

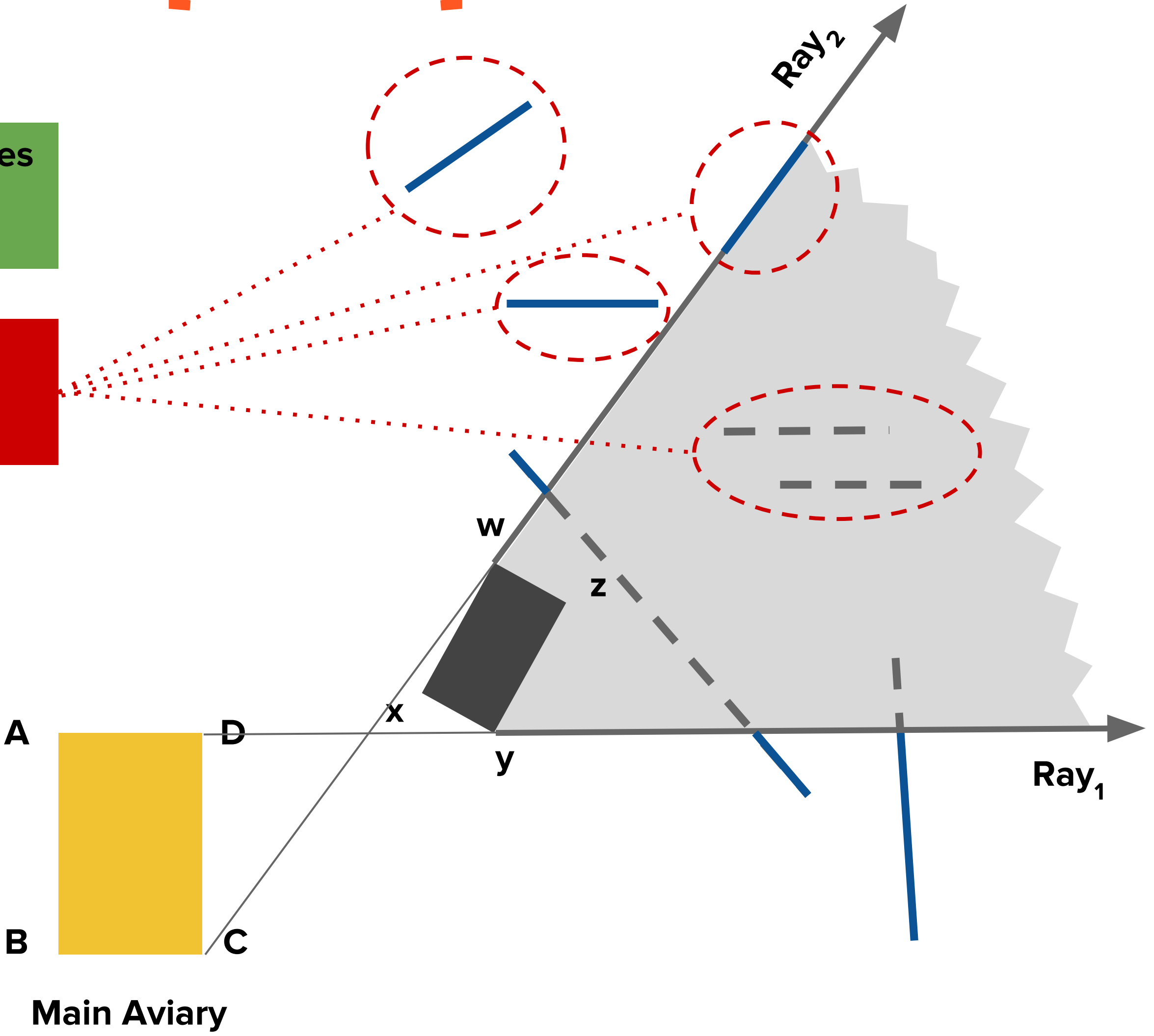
Select Ray$_1$ and Ray$_2$ which maximize α(Ray$_1$, Ray$_2$)



Visitors' Train [cont'd]

Detect and process cases using precise integer arithmetics

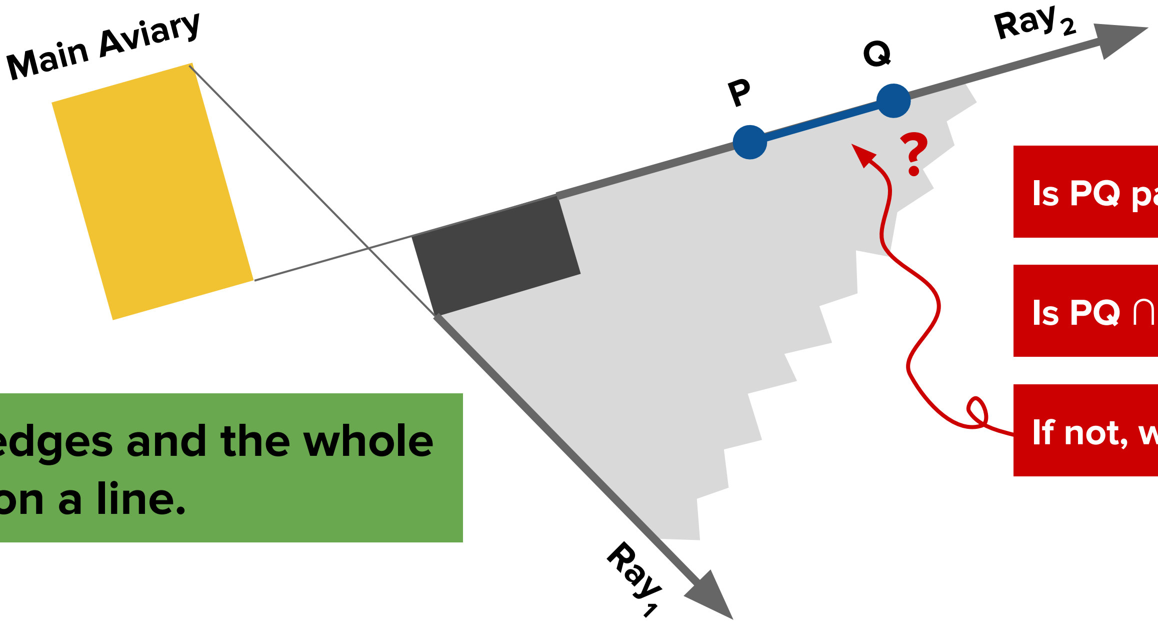
Trivial Cases



Visitors' Train [cont'd]

Canonical Troublesome Case

Floating-point arithmetic is not guaranteed to yield the correct values in this case. Careful use of ε helps!



Is PQ parallel with Ray_2?

Is PQ ∩ Ray_2 empty?

If not, where is it?

Aviaries' edges and the whole track lies on a line.

Questions?

Great Thanks Goes to Problem Setters

Marko Genyk-Berezovskyj

Josef Cibulka

Michal “Mimino” Danilák

Tomáš Tunys

Martin Kačer

Pavel Strnad