# University of Chicago
# Invitational Programming Contest

# Judging Committee

- Tom Conerly
- Mark Gordon
- Darko Aleksic
- Johnny Ho
- Danny Sleator
- vanb

# CosmoCraft

- No need to stockpile money
  - Better off buying workers
- Build as many workers as possible, as early as possible
  - Build Production with remaining $$
- Wait as long as possible to build armies to counter attack
- Build armies the turn of attack. If not enough, go back 1 turn
  - No need to go back any further than 1 turn. Because of the rapid growth of workers & production, you'll actually build fewer armies if you start too soon.

# Covered Walkway

- There is a simple DP solution that is, unfortunately, $O(n^2)$, and that's too slow
- Let's look at it anyway:

For each A[i] compute

best[i] = $MIN_{j<i}\{best[j] + C + (A[i]-A[j+1])^2\}$

Interpretation: For each i, look for j, j<i, where breaking between j and j+1, and making [j+1,i] the last cover, is the best you can do.

- Then, best[n-1] is your answer

# Covered Walkway

- We need a more efficient way to compute best [i].
- Suppose j<k<i. Compare breaking at j with breaking at k.

$$best[j] + C + (A[i]-A[j+1])^2 \;??\; best[k] + C + (A[i]-A[k+1])^2$$

$$best[j] + C + A[i]^2 - 2A[i]A[j+1] + A[j+1]^2 \;??\; best[k] + C + A[i]^2 - 2A[i]A[k+1] + A[k+1]^2$$

Since C and A[i] do not contain j or k, they cannot affect the comparison. So,

$$best[j] - 2A[i]A[j+1] + A[j+1]^2 \;??\; best[k] - 2A[i]A[k+1] + A[k+1]^2$$

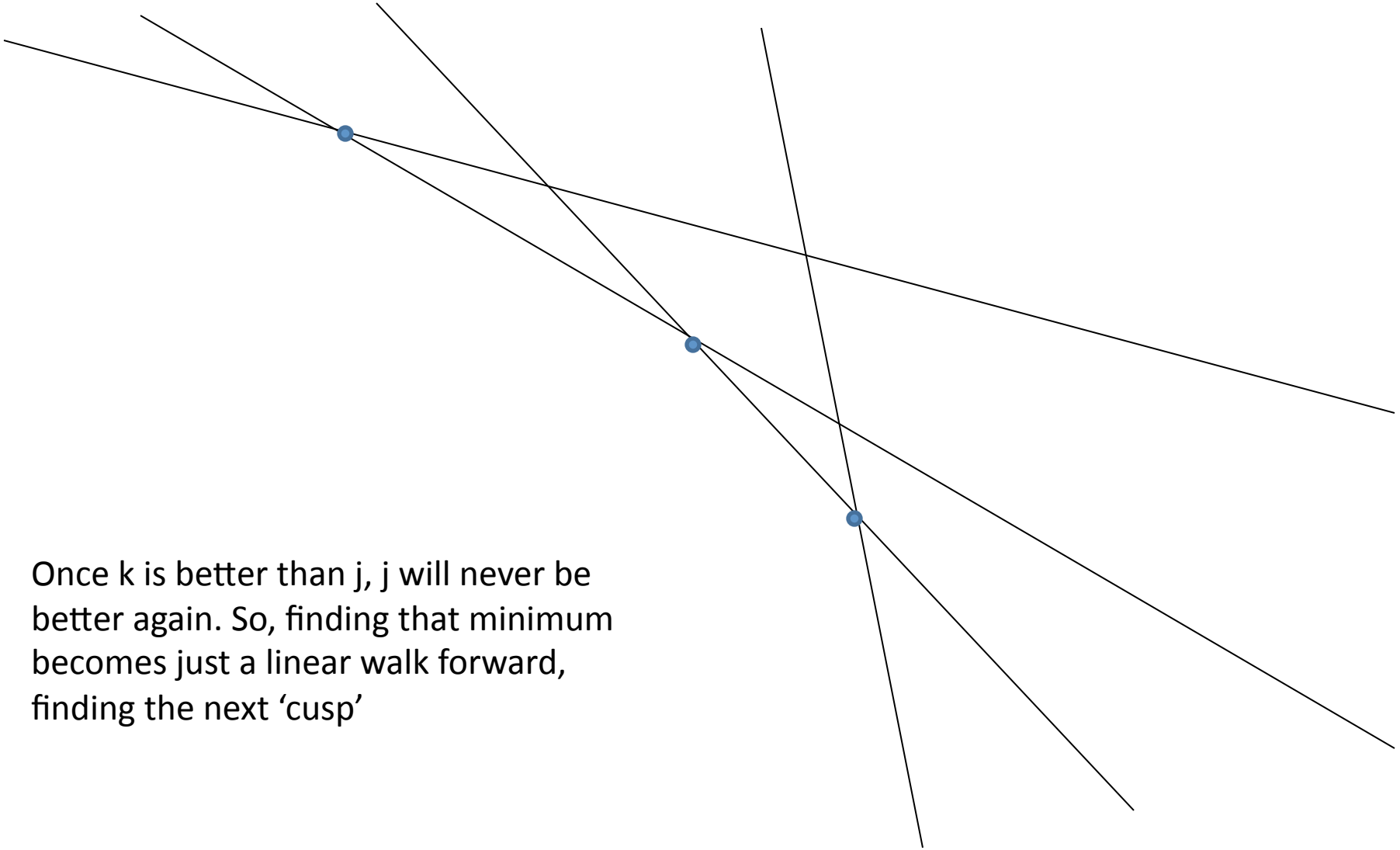
# Covered Walkway

- Express as a function of A[i]:

  $-2A[j+1]*a[i] + (A[j+1]^2+best[j])$ ?? $-2A[k+1]*a[i] + (A[k+1]^2+best[k])$

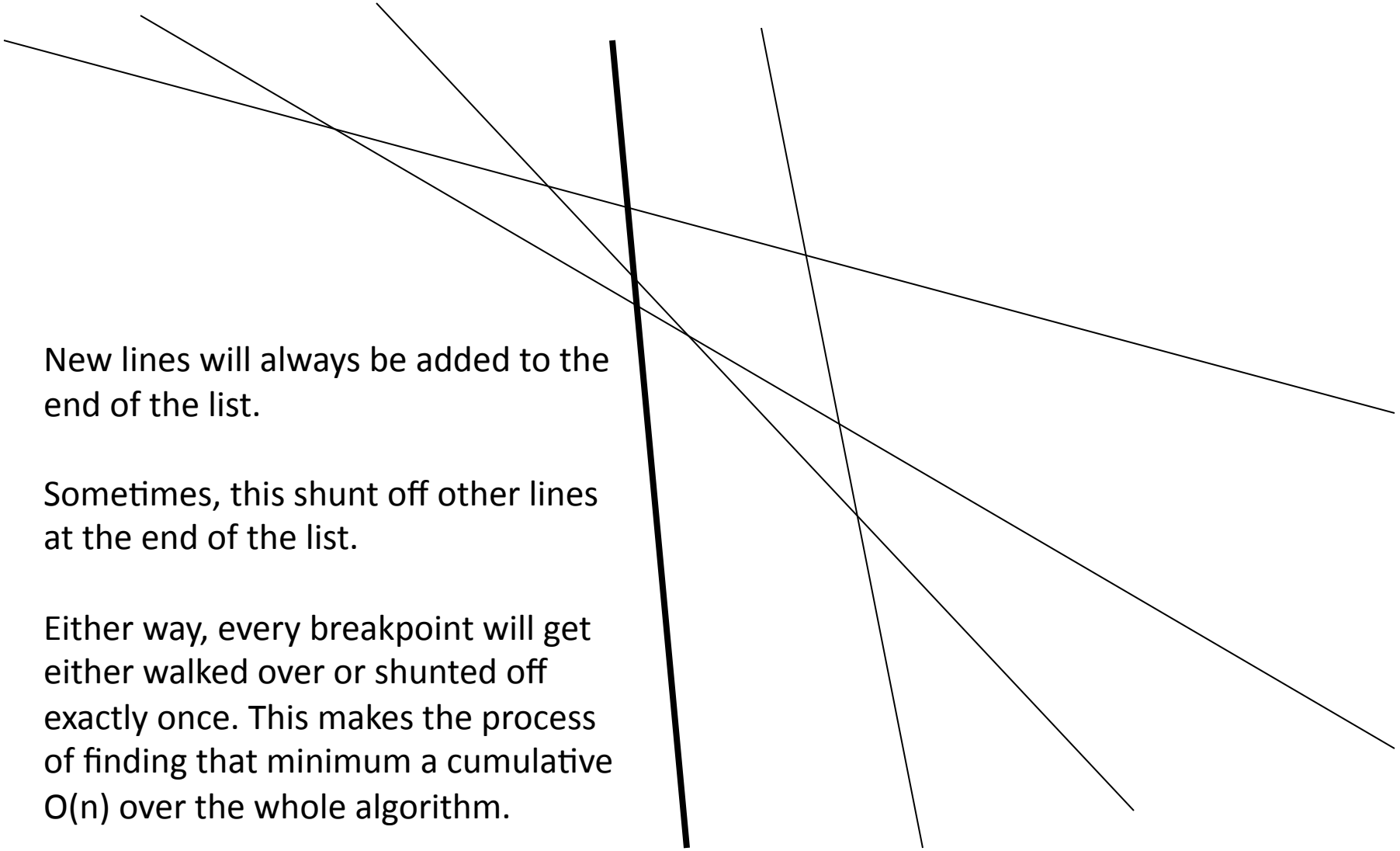
- So, for any i>j,k we can tell whether j or k is a better breakpoint by simply comparing that linear function.
- Because it's linear, there are some geometric interpretations that can help us.
- In particular, look at the "slope". The larger j/k are, the more intensely negative it is. As j/k increase, A[j/k] increases, and that slope gets more and more negative.

# Covered Walkway



Once k is better than j, j will never be better again. So, finding that minimum becomes just a linear walk forward, finding the next ‘cusp’

# Covered Walkway



New lines will always be added to the end of the list.

Sometimes, this shunt off other lines at the end of the list.

Either way, every breakpoint will get either walked over or shunted off exactly once. This makes the process of finding that minimum a cumulative O(n) over the whole algorithm.

# Double Dealing

- Too big to simulate
- Form a graph.
  - Each node represents a card
  - Each node has a single link, to the place where that card goes after one deal/pickup
- Find all the cycles in that graph.
  - A cycle of length n means that after n deal/pickups, all the cards in the cycle will be back where they started
- The answer is the LCM of all of the cycle lengths

# The End of the World

- Key Fact: To solve a ToH with n disks takes $2^n$-1 moves
- To move a pyramid of size n from *source* to *dest*:
  - First move the pyramid of size n-1 on top out of the way, by moving it from *source* to *other*
  - The, move disk n from *source* to *dest*
  - Then, move the n-1 pyramid back on top, from *other* to *dest*.

```
public long remaining( int n, char source, char dest, char other )
{
    long moves = 0;
    if( n>0 )
    {
        --n;
        if( disks[n]==source )
            moves = (1l<<n)+remaining( n, source, other, dest );
        else
            moves = remaining( n, other, dest, source );
    return moves;
}
```

# Estimation

- Two parts:
  - Compute cost[i][j], which is the minimum error if A[i]..A[j] are grouped together in a partition
  - DP to figure out the optimum partitioning

- Fun Fact: For a group of integers A[i]..A[j], the number C which minimizes $\sum_{i \le k \le j}\{|a[k]-C|\}$ is the median of A[i]..A[j].

# Estimation

- Use a binary tree/sorted doubly linked list combo to track medians and costs
  - Reset for each i, go through the remaining js
  - Inserting x as left child of BT node y? Then x immediately precedes y in the sorted list.
    - Likewise, Inserting x as right child of BT node y, x immediately succeeds y in the sorted list.
  - Too many numbers smaller than the current median? Then just step back one in the list.
    - If too many are bigger, step forward one.
- cost[i][j] can be computed easily from cost[i][j-1]
- Use AVL balancing to keep it efficient, even in pathological cases

# Estimation

- Then, once cost[][] is computed, the DP is simple.
- best[i][k] is the best estimation you can come up with for A[0]..A[i], using k partitions.
- best[n-1][p] is your answer

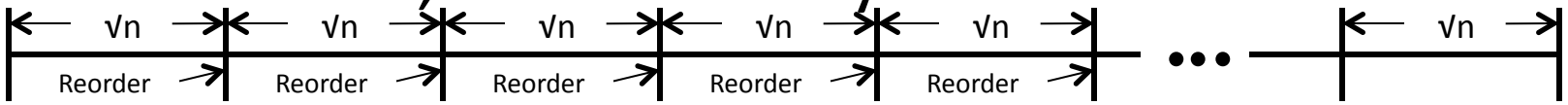
```
for( int i=0; i<n; i++ )
{
    best[i][0] = LARGE;
    best[i][1] = costs[0][i];
    for( int k=2; k<=p; k++ )
    {
        best[i][k] = LARGE;
        for( int j=0; j<i; j++ )
        {
            int cost = best[j][k-1] + costs[j+1][i];
            if( cost<best[i][k] ) best[i][k] = cost;
        }
    }
}
```

# Juggler

- Simple (but too slow) solution:
  - The number of moves to drop the first ball is simple: It's just array index arithmetic for both clockwise and counterclockwise.
  - Dropping the second ball is the same, BUT you've got to account for where the first ball WAS, and decrement that count.
  - Dropping the third? You've got to account for the first AND second.
  - This is all going to add up to an $O(n^2)$ solution.
- What if we reordered the list, moving all of the undropped balls to the beginning?
  - Then, every drop is an O(1) operation, BUT
  - Reordering the list is an O(n) operation. Do that n times, and we're back to $O(n^2)$.

# Juggler

- What if we only reorder the list *sometimes*? For instance, reorder every √n.

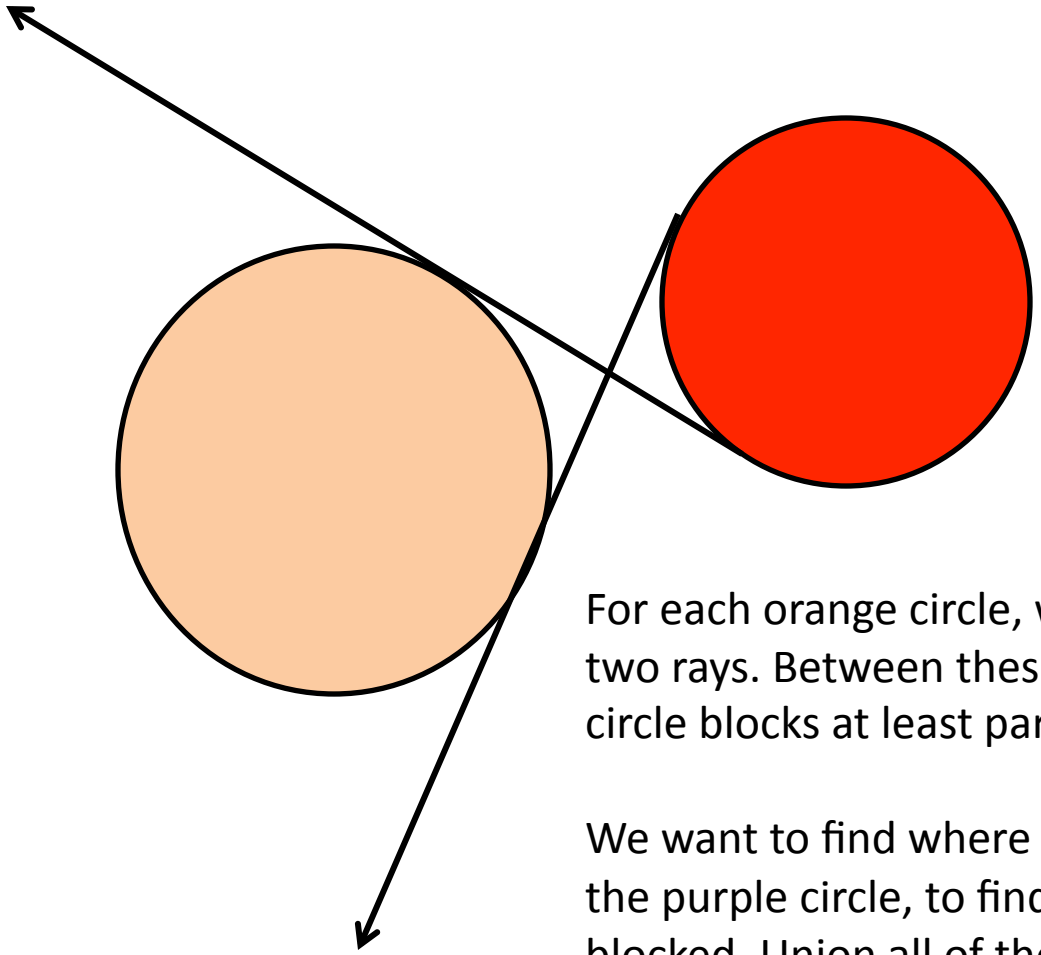



- For each block, figuring out the number of moves is $O(n^2)$. Since there are √n things in each sublist, that's O(n). We'll do that √n times, so that comes to O(n√n).
- What about reordering? That's O(n), and we do it √n times, that comes to O(n√n)
- This brings in the whole algorithm at O(n√n)

# Red/Blue Spanning Tree

- Just use the standard Kruskal's algorithm for Minimum Spanning Tree, twice
  - First pass, figure out which blue edges are *necessary*.
    - Start with only red edges, and go as far as you can. This will identify the red components which must be connected with blue components.
    - Then fill in with blue, and remember the *necessary* blue edges.
  - Second pass, clear everything out & start again.
    - Start with the *necessary* blues
    - Then use the other blues, until you have exactly ***k***.
    - Finish off with reds
- At the end of the second pass, if you've used exactly ***n***-1 edges, and exactly ***k*** of them are blue, then you've succeeded.

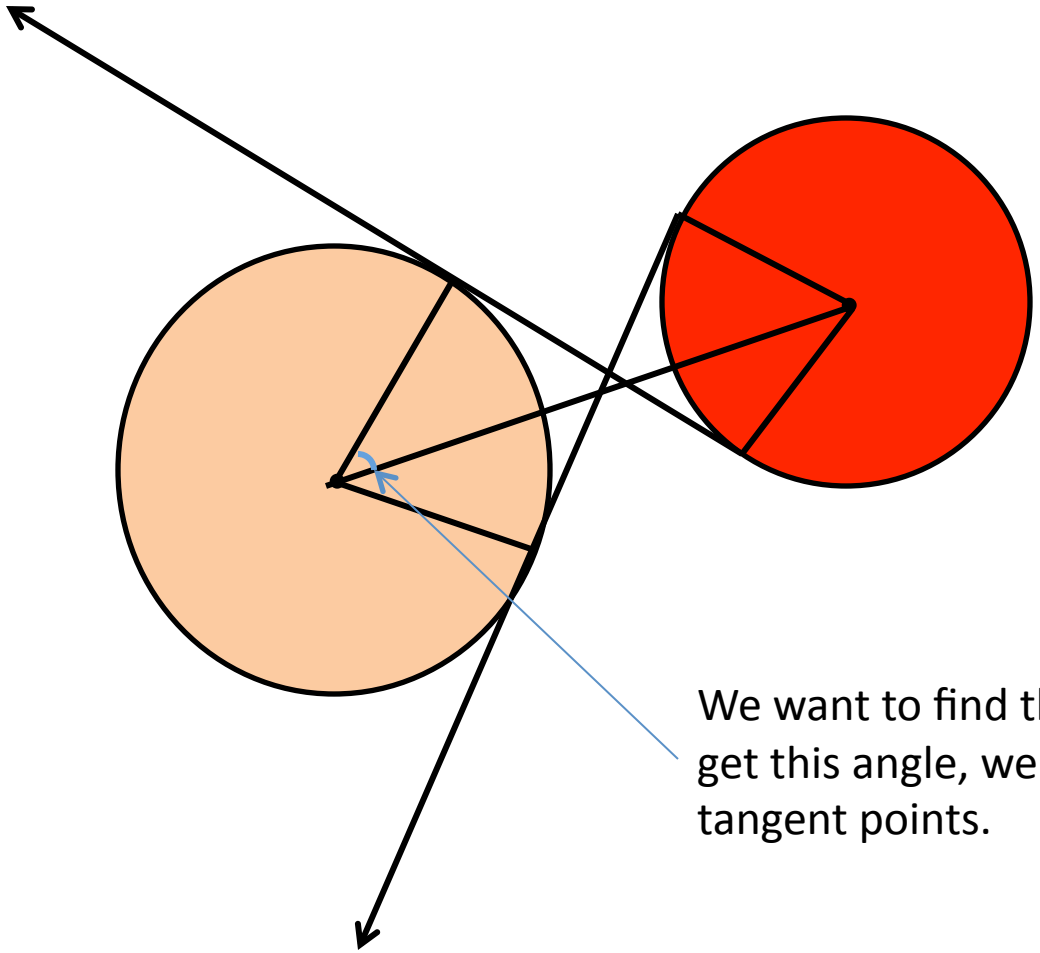
# The Red Gem



For each orange circle, we want to find these two rays. Between these rays, the orange circle blocks at least part of the red circle.

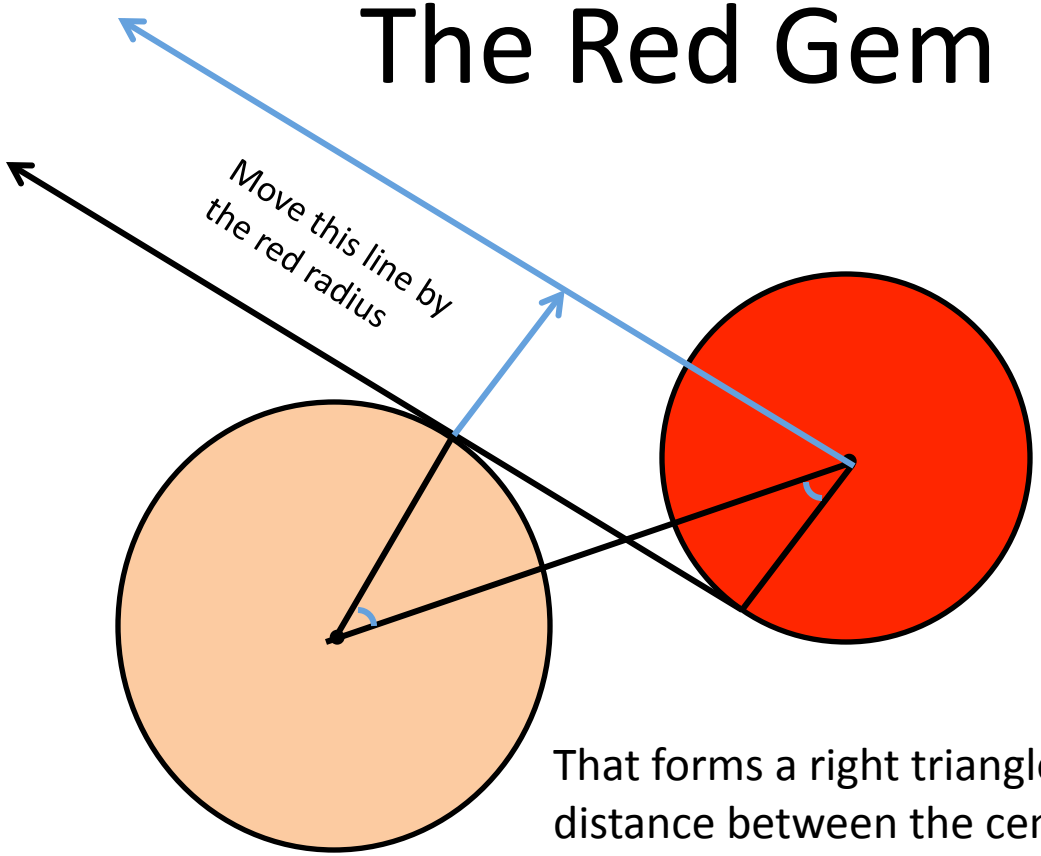
We want to find where these rays intersect the purple circle, to find segments that are blocked. Union all of those segments to get to proportion of the purple perimeter blocked, which is 1-(answer)

# The Red Gem



We want to find this angle. Once we get this angle, we can find all of the tangent points.

# The Red Gem




That forms a right triangle. Hypotenuse is the distance between the centers, adjacent side is the sum of the radii. So, the angle we seek is:

arcsine( (or+rr) / dor )

Once we know that angle, we can use the circle centers and radii (inputs) and the angle between the centers (atan2!) to get the tangent points.

# The Red Gem

- Once we have these points for both rays, we can project them to the surface of the purple circle.
- Express the rays parametrically:

  $x = px + qx*t$

  $y = py + qy*t$

  $t>0$
- Plug these into the equation for the circle ($x^2+y^2=r^2$)
- That will give us a quadratic in t, which we can solve with the quadratic formula.
- Plug that t solution back into the parametric equations, and we'll get the (x,y) where the ray intersects the purple circle
- Then atan2(y,x) will give us the angle we seek!

# Science!

- Ford-Fulkerson!!
- Build a FF matching graph:
  - 1 source, n people, n buttons, 1 sink
  - Source linked to all people, weight TBD
  - All buttons linked to sink, weight TBD
  - Person linked to button if the YN matrix sez so, weight 1
- Use FF repeatedly to find the max number of workable permutations
  - Set the source→people weights and and button→sink weights to k, see if the max flow is n*k
  - Use binary search for efficiency
- Once you know k, eliminate unused links
  - They can get you stuck in a corner
- Finally, use FF repeatedly to generate the permutations
  - Set source→people and button->sink weights to 1
  - Do FF, see which edges were used between people & buttons
  - Eliminate those edge, repeat k times

# The Worm in the Apple

- First, build a 3D convex hull. Must do this efficiently ($O(n^2)$ or less)
  - There are known algorithms for this, but even if you don't have one in your hack pack, you can still reason it out. The next slide will outline the algorithm used in one of the judges' solutions
- Then, find an efficient way to perform the queries
  - For each face of the hull, determine the distance from the query point to the plane

# The Worm in the Apple

- First, find the point with the smallest z value. Translate all points so that this point is at the origin.
- Now, find the point which has the smallest angle with the XY plane.
  - If you roll the figure like a soccer ball, this will be the first point to touch the ground.
  - Minimize arcsine( $z / \sqrt{(x^2+y^2+z^2)}$ )
- We've now got an edge. If we can find a third point, we'll have a face.
  - Rotate in the XY plane (around the z axis) so that this second point is above the X axis (y=0).
  - Then, rotate in the XZ plane (around the Y axis) until this point is on the positive X axis
  - Find the point which makes the smallest angle in the YZ plane. That's the third point of this face.
- Now, we've got 3 vertices, 3 edges, 1 face. How do we find the others?

# The Worm in the Apple

- We'll proceed like a Breadth-First Search
- Keep a queue of edges (start with the first 3).
- Each edge lies between two triangular faces. We'll already know one. If we know the other, we can move on to the next edge on the queue. If not, we'll need to find the other.
- Do the translation/rotation trick to put the known face in the XY plane
  - Point 1 at the origin
  - Point 2 on the positive X axis
  - Point 3 with in the XY plane (z=0), with y<0.
- Then, the point with the smallest angle in the YZ plane forms the other triangle with the edge that's on the X axis.
- That will give us 2 more edges to put on the queue.

# The Worm in the Apple

- Now, the queries.
- For each face, remember the translation/rotation parameters that put that face on the XY plane (z=0).
- For each query point, go through all the faces and perform the translation/rotation. Then, |z| is the distance we seek.
- We don't need to test if the point is actually directly above the face. The smallest distance is guaranteed to be to a face that the point is directly above.