# Solutions

BAPC 2017

University of Amsterdam

October 2017

# Falling Apart (54/77)



- Sort the input in descending order ($a_1 \geq a_2 \geq \cdots \geq a_n$).
- Output $a_1 + a_3 + a_5 + \ldots$ and $a_2 + a_4 + \ldots$.

# Amsterdam Distance (51/62)



- Find the shortest path from $A$ to $B$ in Amsterdam.
- You can solve this problem with Dijkstra, but that is the stupidest way.

# Amsterdam Distance (Dijkstra)



```
V<double> dist(nv, INF); dist[s] = 0;
priority_queue<edge> pq; pq.push({s,0});
while (S(pq)) {
    edge u = pq.top(); pq.pop();
    if (dist[u.i] < u.d) continue;
    for (edge v : adj[u.i]) {
        v.d += u.d;
        if (dist[v.i] > v.d)
            dist[v.i] = v.d, pq.push(v);
    }
}
```

# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017

```
V<double> dist(nv, INF); dist[s] = 0;
priority_queue<edge> pq; pq.push({s,0});
while (S(pq)) {
    edge u = pq.top(); pq.pop();
    if (dist[u.i] < u.d) continue;
    for (edge v : adj[u.i]) {
        v.d += u.d;
        if (dist[v.i] > v.d)
            d
    }
}
```

```
def dijkstra(s, t):
  d, S, q = { s: 0 }, set(), []; heapq.heappush(q, (0,s)
  while q:
    n, u = heapq.heappop(q); x, y = u
    if u in S: continue
    S.add(u)
    if u == t: break
    for v, w in [((x+1,y),B*y),((x-1,y),B*y),((x,y-1),A)
```

# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017

```
V<double> dist(nv, INF); dist[s] = 0;
priority_queue<edge> pq; pq.push({s,0});
while (S(pq)) {
    edge u = pq.top(); pq.pop();
    if (dist[u.i] < u.d) continue;
    for (edge v : adj[u.i]) {
        v.d += u.d;
        if (dist[v.i] > v.d)
```

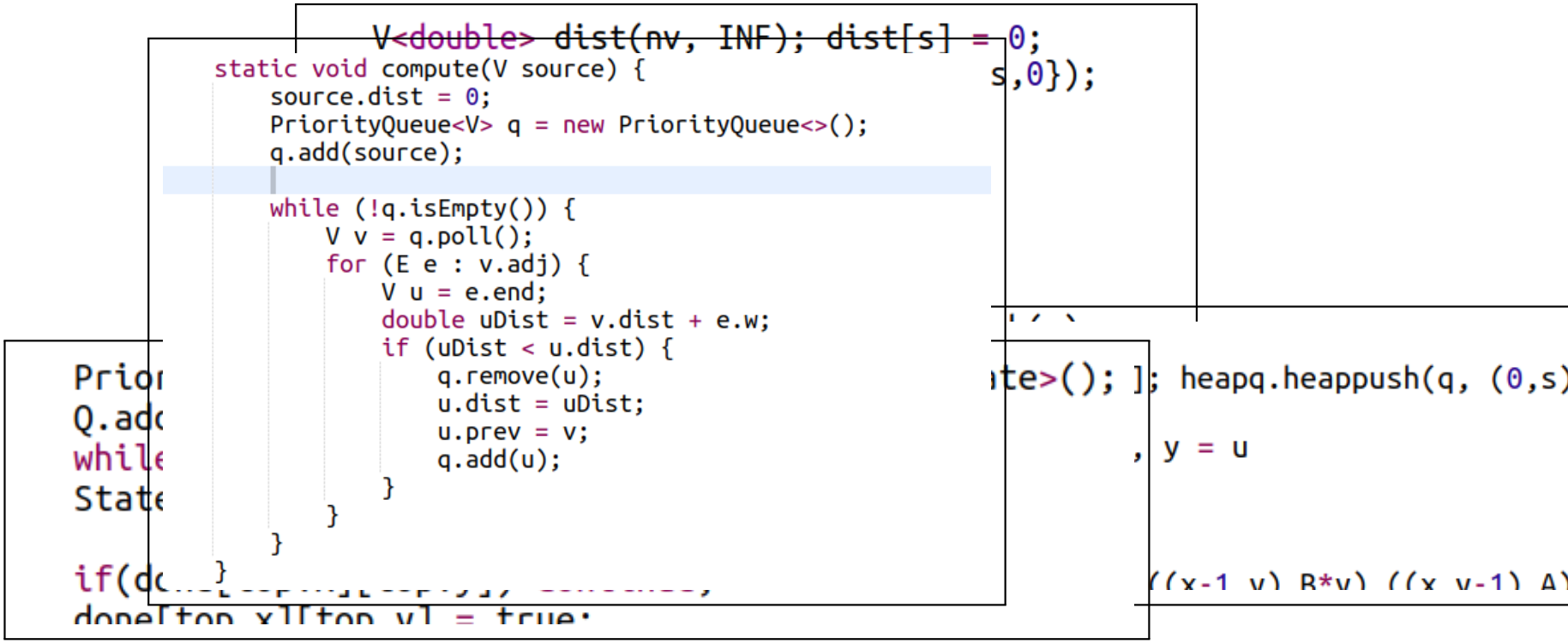
```
def dijkstra(s, t):
```

```
PriorityQueue<State> Q = new PriorityQueue<State>(); ]; heapq.heappush(q, (0,s)
Q.add(new State(ax, ay, 0));
while(!Q.isEmpty()) {                                   , y = u
State top = Q.poll();

if(done[top.x][top.y]) continue;                        ((x-1,y),B*y),((x,y-1),A)
done[top.x][top.y] = true;
```

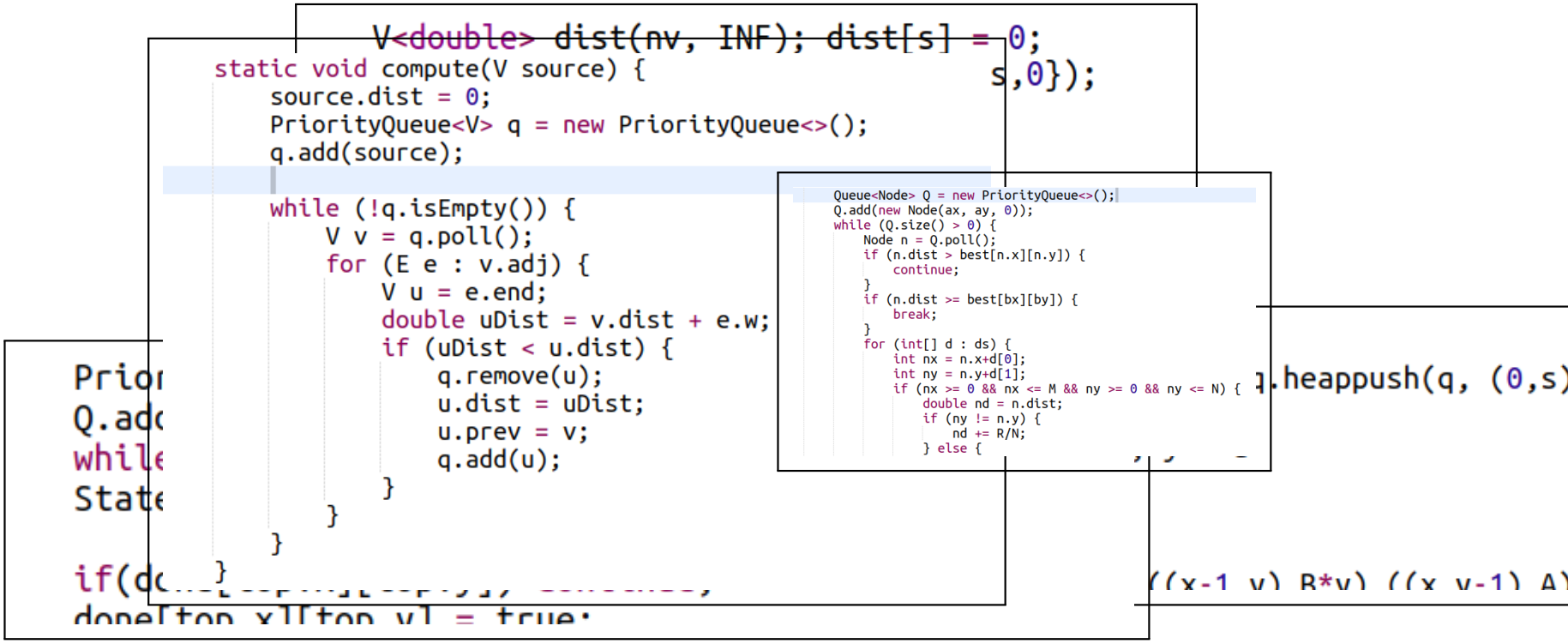
# Amsterdam Distance (Dijkstra)





# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017



# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017

```
V<double> dist(nv, INF); dist[s] = 0;
PriorityQueue<Node> pq = new PriorityQueue<Node>();
pq.add(new Node(ax, ay, 0));
DIST[ax][ay] = 0;
while(!pq.isEmpty()){
    Node s = pq.poll();
    LinkedList<Node> ns = s.getSons(M, N, R);
    for(Node nn : ns){
        if(nn.D < DIST[nn.X][nn.Y]){
            DIST[nn.X][nn.Y] = nn.D;
            pq.add(nn);
        }
    }
}
System.out.println(DIST[bx][by]);
```

# Amsterdam Distance (Dijkstra)

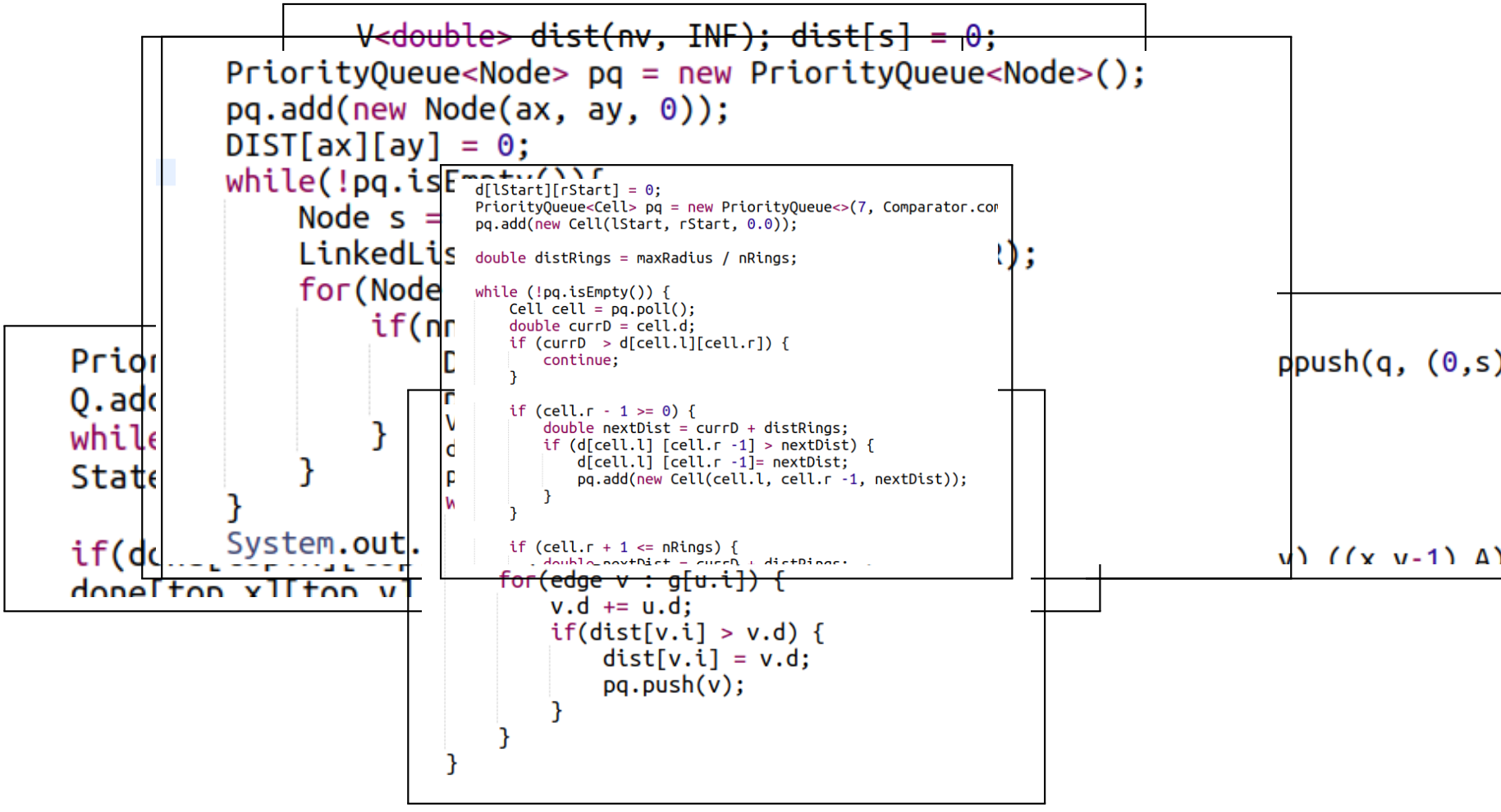
Benelux Algorithm Programming Contest 2017

```
V<double> dist(nv, INF); dist[s] = 0;
PriorityQueue<Node> pq = new PriorityQueue<Node>();
pq.add(new Node(ax, ay, 0));
DIST[ax][ay] = 0;
while(!pq.isEmpty()){
    Node s = pq.poll();
    LinkedList<Node> ns = s.getSons(M, N, R);
    for(Node nn : ns){
        if(nn.D < DIST[nn.X][nn.Y]){
            DIST[nn.X][nn.Y] = nn.D;
            pq.add(nn);
        }
    }
}
System.out.
```

```
V<double> dist(nv, INF);
dist[s] = 0;
priority_queue<edge> pq; pq.push({s,0});
while(S(pq)) {
    edge u = pq.top(); pq.pop();
    if(dist[u.i] < u.d) continue;
    for(edge v : g[u.i]) {
        v.d += u.d;
        if(dist[v.i] > v.d) {
            dist[v.i] = v.d;
            pq.push(v);
        }
    }
}
```

# Amsterdam Distance (Dijkstra)





# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017

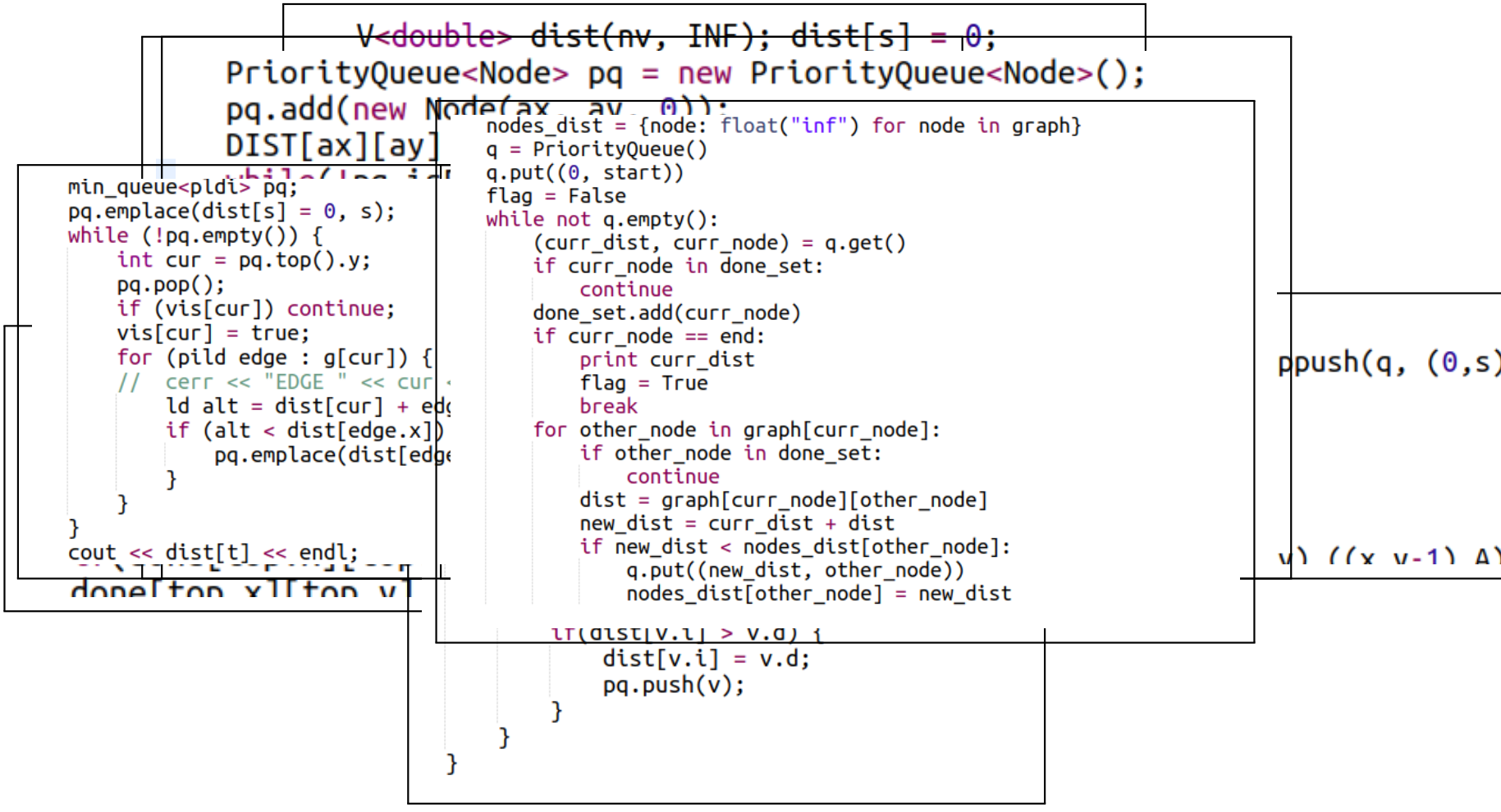
```
V<double> dist(nv, INF); dist[s] = 0;
PriorityQueue<Node> pq = new PriorityQueue<Node>();
pq.add(new Node(ax, ay, 0));
DIST[ax][ay] = 0;
```

```
min_queue<pldi> pq;
pq.emplace(dist[s] = 0, s);
while (!pq.empty()) {
    int cur = pq.top().y;
    pq.pop();
    if (vis[cur]) continue;
    vis[cur] = true;
    for (pild edge : g[cur]) {
    //  cerr << "EDGE " << cur << " -> " << edge.x << " weigh
        ld alt = dist[cur] + edge.y;
        if (alt < dist[edge.x]) {
            pq.emplace(dist[edge.x] = alt, edge.x);
        }
    }
}
cout << dist[t] << endl;
```

```
for(edge v : g[u.i]) {
    v.d += u.d;
    if(dist[v.i] > v.d) {
        dist[v.i] = v.d;
        pq.push(v);
    }
}
}
```

# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017



```
V<double> dist(nv, INF); dist[s] = 0;
PriorityQueue<Node> pq = new PriorityQueue<Node>();
pq.add(new Node(ax, ay, 0));
DIST[ax][ay]
```

```
min_queue<pldi> pq;
pq.emplace(dist[s] = 0, s);
while (!pq.empty()) {
    int cur = pq.top().y;
    pq.pop();
    if (vis[cur]) continue;
    vis[cur] = true;
    for (pild edge : g[cur]) {
    //  cerr << "EDGE " << cur
        ld alt = dist[cur] + edg
        if (alt < dist[edge.x])
            pq.emplace(dist[edge
        }
    }
}
cout << dist[t] << endl;
done[top.x][top.y]
```

```
nodes_dist = {node: float("inf") for node in graph}
q = PriorityQueue()
q.put((0, start))
flag = False
while not q.empty():
    (curr_dist, curr_node) = q.get()
    if curr_node in done_set:
        continue
    done_set.add(curr_node)
    if curr_node == end:
        print curr_dist
        flag = True
        break
    for other_node in graph[curr_node]:
        if other_node in done_set:
            continue
        dist = graph[curr_node][other_node]
        new_dist = curr_dist + dist
        if new_dist < nodes_dist[other_node]:
            q.put((new_dist, other_node))
            nodes_dist[other_node] = new_dist
```

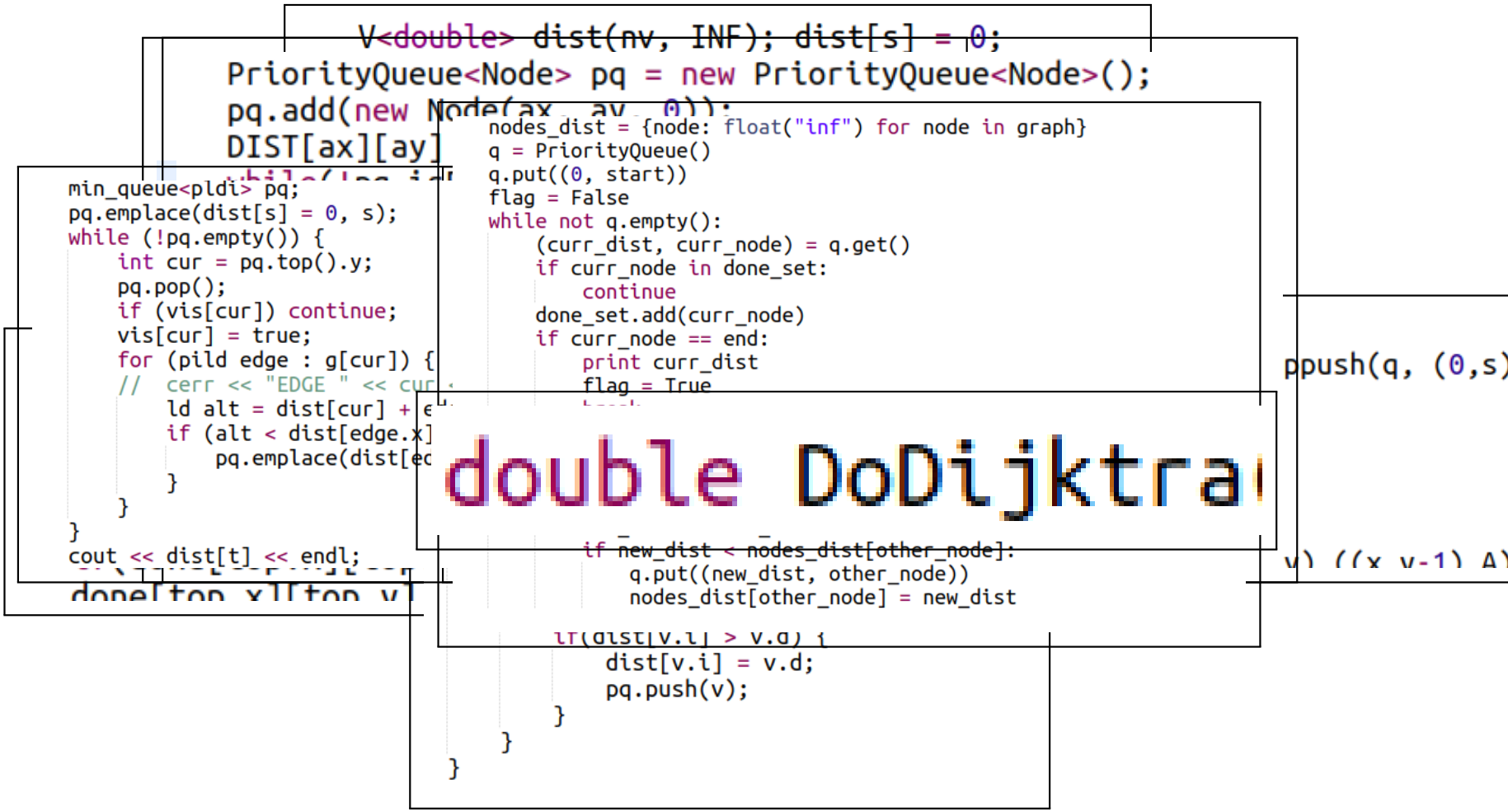
```
if(dist[v.i] > v.d) {
    dist[v.i] = v.d;
    pq.push(v);
}
}
}
```

```
ppush(q, (0,s)
```

```
v) ((x,y-1) A
```

# Amsterdam Distance (Dijkstra)

Benelux Algorithm Programming Contest 2017



# Amsterdam Distance

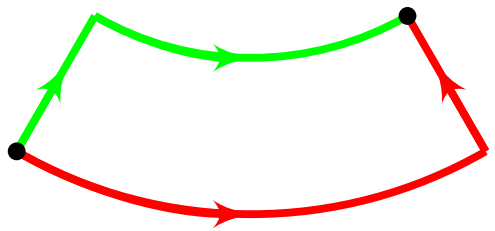


- Find the shortest path from $A$ to $B$ in Amsterdam.
- You can solve this problem with Dijkstra, but that is the stupidest way.

# Amsterdam Distance



- Find the shortest path from $A$ to $B$ in Amsterdam.
- You can solve this problem with Dijkstra, but that is the stupidest way.
- Observation: if you walk around a city block, you always want to go via the inside.

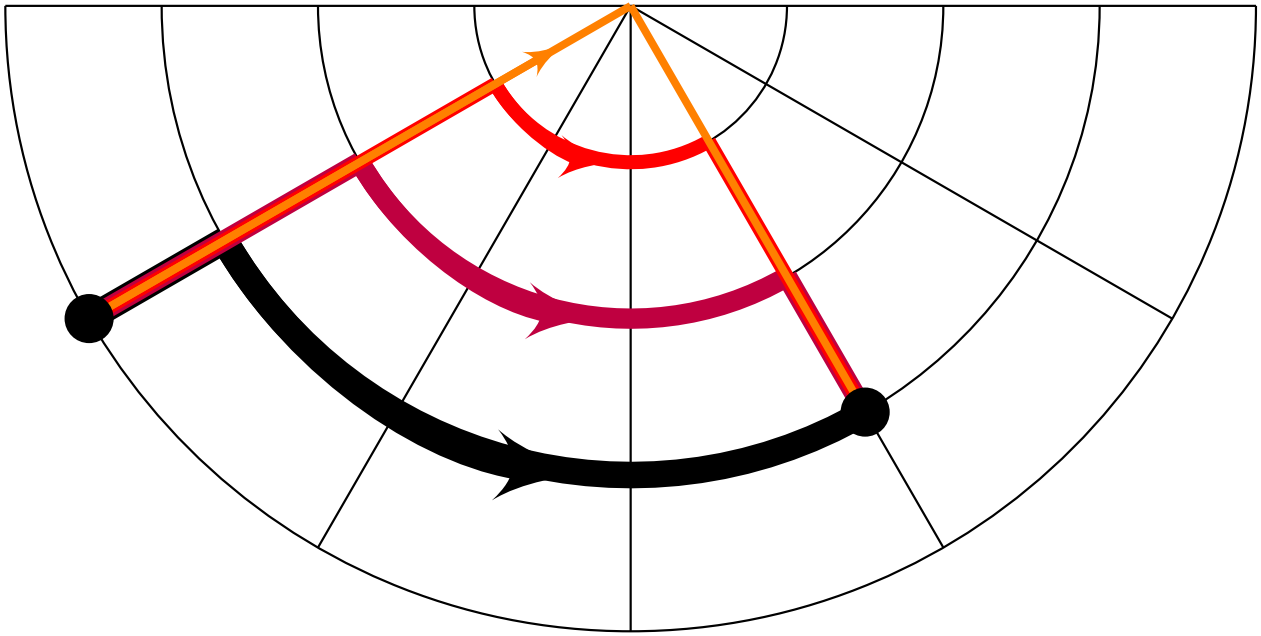


- This means every shortest route has at most three parts:
  - Walk towards the center.
  - Walk along a canal.
  - Walk away from the center.

# Amsterdam Distance



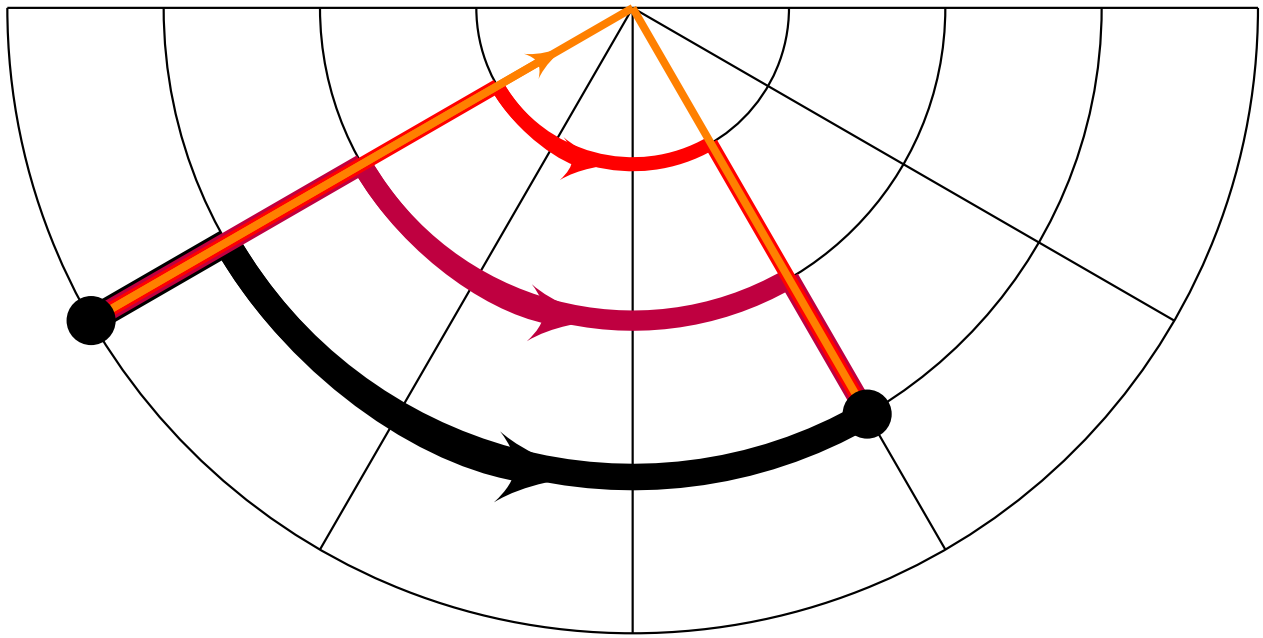
- This leaves only a few options.



# Amsterdam Distance



- This leaves only a few options.



- You can loop over the relevant circle arcs ($\leq 100$) and find the shortest path.
- In fact, because you optimize something linear, the shortest path is always the outermost (black) or innermost (orange).

# Irrational Division (32/87)



- Option 1: do a min-max DP (N.B.: the "upper left" block can be either black or white.)
- Option 2: work out small cases, find a pattern:
  - If $p$ is even, then the answer is zero.
  - If both $p$ and $q$ are odd, the answer is 1.
  - If $p$ is odd, $q$ is even and $p < q$, the answer is 2.
  - If $p$ is odd, $q$ is even and $p > q$, the answer is 0.

# Irrational Division (Code examples)



```
System.out.println((n % 2) * 2);
```

# Irrational Division (Code examples)



```
    System.out.println((n % 2) * 2);
```

```
if (p == 1 and q % 2 == 0)
    System.out.println(2);
else if (p % 2 == 1 and q % 2 == 1)
    System.out.println(1);
else
    System.out.println(0);
```

# Irrational Division (Code examples)



```
    System.out.println((n % 2) * 2);
```

```
if (p == 1 and q % 2 == 0)
    System.out.println(2);
else if (p % 2 == 1 and q % 2 == 1)
    System.out.println(1);
else
    System.out.println(0);
```

```
if (p % 2 == 1 and q % 2 == 1)
    System.out.println(1);
else if (p % 2 == 0)
    System.out.println(0);
else if (q > p)
    System.out.println(2);
else
    System.out.println(0);
```

# Irrational Division (proofs)



- By considering YOU TAKE EVERYTHING or YOU TAKE SOMETHING AND YOUR SISTER GRABS EVERYTHING, it follows that the score will be either 0, 1, or 2.
- The first two cases follow directly.
- For the last two one can the following theorem:

**Theorem**

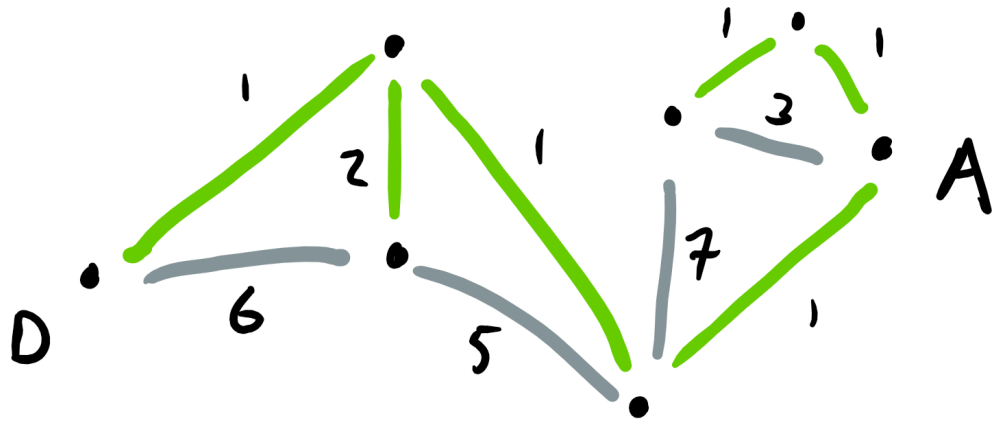
*A chocolate $n \times n$ "white" square (i.e. with a white block in the upper left corner) will net you a score of $-1$.*

This leads to a strategy where either you or your sister (depending on $p < q$ or $p > q$) will always reduce the block to a white square, and the scores follow.

# Detour (31/87)

Benelux Algorithm Programming Contest 2017

- Is there a route which *never* takes the shortest route?
- We need to know the shortest path from $x$ to Amsterdam first.
- Dijkstra from Amsterdam to determine all shortest paths to it.
- Remove those edges. Make sure not to remove in both directions.
- Run any traversal algorithm (from Delft or Amsterdam).



# Lemonade Trade (17/103)



- Find the maximum amount of blue lemonade that you can get for one litre of pink lemonade.
- For each colour $c$ keep

$$M[c] := \log(\text{maximum amount of } c \text{ lemonade you can get}).$$

- Update this for each line in the input.
- Be aware of precision and enormous overflow problems! This is the reason why you need to use logarithms for this!

# King of the Waves (16/36)

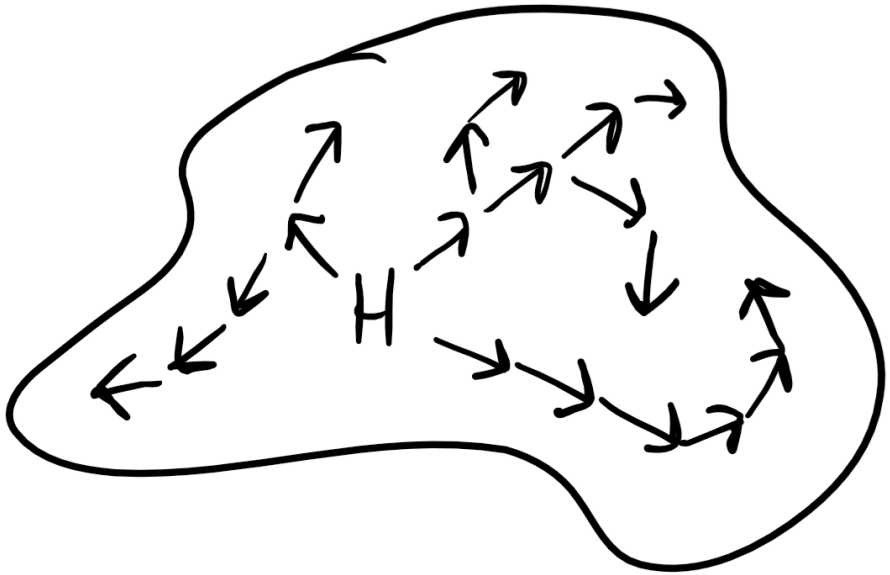


- Can we order the participants in such an order that Henk wins?
- Trying all permutations obviously too slow
- Greedy not correct: we may not know whether Henk became king or already was a king.
- Input: complete graph with edges $(u, v)$ if $u$ beats $v$.

# King of the Waves

Benelux Algorithm Programming Contest 2017

- Insight: Henk can win $\Leftrightarrow$ there is a path from Henk to every other participant.
- $\Rightarrow$: If no path to $x$, then we get stuck at person $x$ (everyone has to play!)
- $\Leftarrow$: Following one path back to Henk leaves Henk as king. What about multiple paths?




- Code: any traversal algorithm (DFS or BFS or ...?)

# Easter Eggs (6/20)



- Do a binary search on the distance $D$, and answer the decision problem: is it possible to put $N$ eggs, such that the distance between a red and blue egg is at least $D$?
- Consider the bipartite graph with the blue and red bushes as vertices. Draw an edge if the distance is at most $D$.
- We are now looking for a maximum independent subset in this graph.
- This is the complement of a minimum vertex cover.
- König's theorem states that this is equivalent to the size of a maximum matching.

# Collatz Conjecture (5/37)



- Given $a_1, a_2, \ldots, a_n$, compute the number of unique values $f(i,j) = \gcd(a_i, a_{i+1}, \ldots, a_j)$ takes on.
- Various solutions with various runtimes. The fastest runs in $O(n \log^2 A)$ time and is based on the following observations: (cont.)

# Collatz Conjecture



Observation 1: consider the following sequence of gcd's:

$$\gcd(a_1)$$
$$\gcd(a_1, a_2)$$
$$\ldots$$
$$\gcd(a_1, a_2, a_3, \ldots, a_{n-2})$$
$$\gcd(a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1})$$
$$\gcd(a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}, a_n)$$

- Since $\gcd(a, b, c) = \gcd(\gcd(a, b), c)$, each value in this sequence divides the value above it.
- Since every proper division halves the value, this sequence contains at most $\lceil \log a_1 \rceil + 1$ unique values.

# Collatz Conjecture



Observation 2:

- Let $D_i = \{ \gcd(a_j, a_{j+1}, \ldots, a_{i-1}, a_i) \mid j \leq i \}$. By observation 1 the size of $D_i$ is $\mathcal{O}(\log a_i)$
- We can easily compute $D_{i+1}$ from $D_i$, again using the identity $\gcd(a, b, c) = \gcd(\gcd(a, b), c)$, namely:

$$D_{i+1} = \{ \gcd(v, a_{i+1}) \mid v \in D_i \} \cup \{ a_{i+1} \}$$

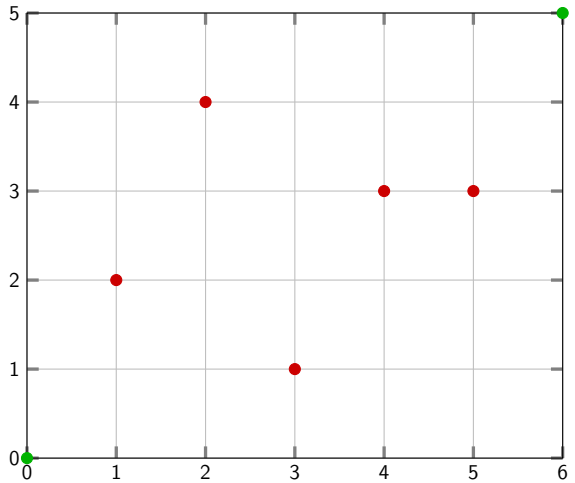
- The above computation takes $\mathcal{O}(\log^2 A)$ time.

These two observations result in a very simple $\mathcal{O}(n \log^2 A)$ algorithm: just compute and union all the $D_i$'s, and output the size of the resulting set.
(Additionally, more complicated approaches can also pass when written efficiently.)

# Manhattan mornings (4/14)

Benelux Algorithm Programming Contest 2017

- Given a rectangular grid, find the maximum errands you can run along your route without increasing the route length.
- All the errands outside the rectangle between your house $(x_h, y_h)$ and the workplace $(x_w, y_w)$ can be skipped.
- Sort all errands within those limits $(x_i, y_i)$ on $x$, resulting in a list of $y_i$ coordinates.
- If you decide to run errand $y_i$, all following errands $j$ must have $y_j > y_i$.



# Manhattan mornings



- Observation: the longest increasing subsequence of these $y$-coordinates is equal to the amount of errands you can run.
- Solve the following recurrence relation:

$$errands(n) = max_{i=0, y_i < y_n}^{n}(1 + errands(i))$$

- Naive: $O(n^2)$ - too slow
- Avoid recalculation by using a map: $O(n \log(n))$ - fast enough
- When the workplace is below the house, make sure to calculate the longest decreasing subsequence.

# Going Dutch (3/25)



- Think of transactions as a graph.
- Observe: a tree always suffices. So $K$ people require at most $K - 1$ transactions.
- If $K$ people can do it in $K - 2$ transactions, that means we can split them into two groups with balance 0.
- So if we can split the $n$ people into $i$ zero balance groups we need $N - i$ transactions.

# Going Dutch



- So we are looking for the largest number of zero-balance sets we can split $\{1, \ldots, n\}$ into.
- Equivalently: we are looking for the longest chain of sets

$$\emptyset \subsetneq S_1 \subsetneq \cdots \subsetneq S_i = \{1, \ldots, n\}$$

where each set has zero balance, so we can do it in $n - i$ transactions.

# Going Dutch



- For each set (bitmask) $S$, let $L(S)$ be the largest integer $i$ such that we have a chain

$$\emptyset \subsetneq S_1 \subsetneq \cdots \subsetneq S_i \subseteq S$$

  with each $S_i$ (but not necessarily $S$) having zero balance.
- Then for $S \neq \emptyset$,

$$L(S) = \max_{p \in S} L(S \setminus \{p\}) + \begin{cases} 1 & \text{if } S \text{ is zero-balance.} \\ 0 & \text{else.} \end{cases}$$

- Since $|S| \leq n$, this takes $n$ steps to compute from previous results, which gives a $\mathcal{O}(n2^n)$ DP algorithm.
- The answer is given by $n - L(\{1, \ldots n\})$.

# Hoarse Horses (2/15)



- Given a set of lines in the plane, count the number of non-empty regions they enclose.
- We could just find all intersections between lines and try to count all areas by starting at an intersection, and then following outgoing lines in a clockwise order.
- Error-prone and a lot of work, will probably make an error ...

# Hoarse Horses



- Combinatorics to the rescue!
- Euler: for a connected planar graph, the number of vertices $V$, edges $E$ and planar faces $F$ obeys $V - E + F = 2$ (this includes the outer face).
- There are two problems here:
  - Input graph need not be planar.
  - Input graph need not be connected.

# Hoarse Horses



We can planarize the graph by replacing two intersecting edges with four edges with a new vertex at the center.

For connectedness - Euler's formula generalizes to $V - E + F = 1 + C$ for a planar graph with $C$ components (just connect the components with $C - 1$ edges in a tree-like fashion).

# Hoarse Horses



- For the purposes of this computation, we don't actually care *where* two fences intersect, just *that* they do, so you can use integer arithmetic to avoid precision issues (in fact, we don't even care what fences intersect - we just care about the total number of intersections).
- Final answer is $\#components - n + \#intersections$.

# Jumping Choreography (0/10)

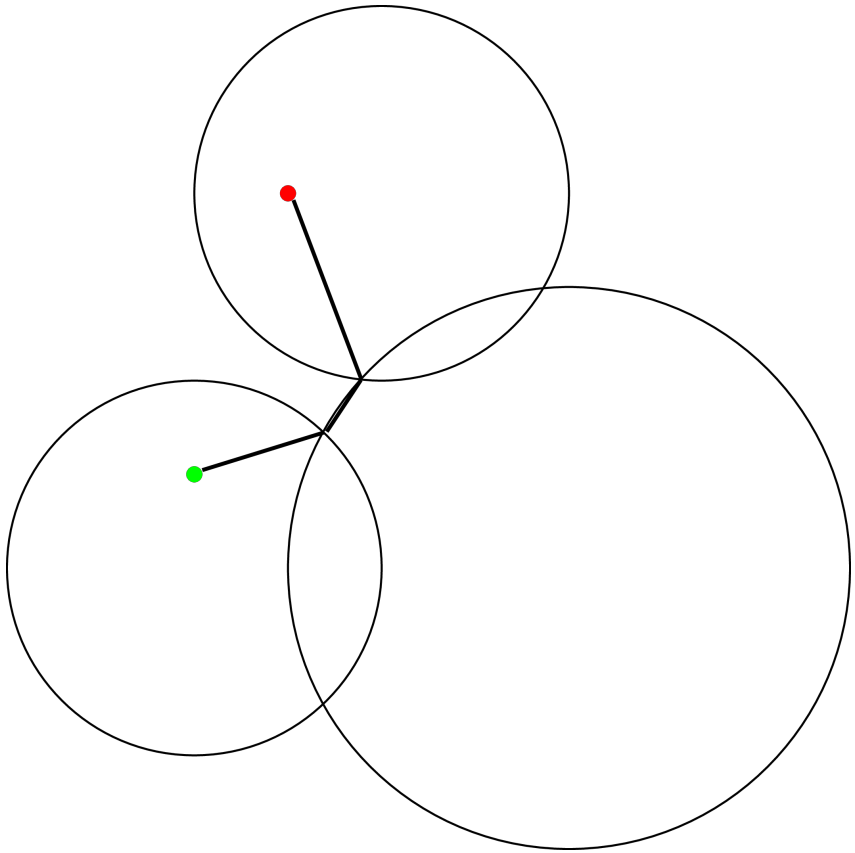


- Calculate the number of jumps needed to go $i$ to the right: Approximately $\sqrt{2i}$ steps needed, possible one extra due to parity.
- This takes at most 1500 different values. When considering even (or odd) $i$ only, the function is increasing.
- Use two fenwick trees to store the total number of steps needed to get to each position, one for odd and one for even positions.
- When adding/removing a frog, update both in $O(1500 * \ln(10^6))$.
- Querying is $O(ln(10^6))$.
- Be careful to use fast input/output!
- Brute force is easy, but too slow.

# Bearly Made It (0/0)



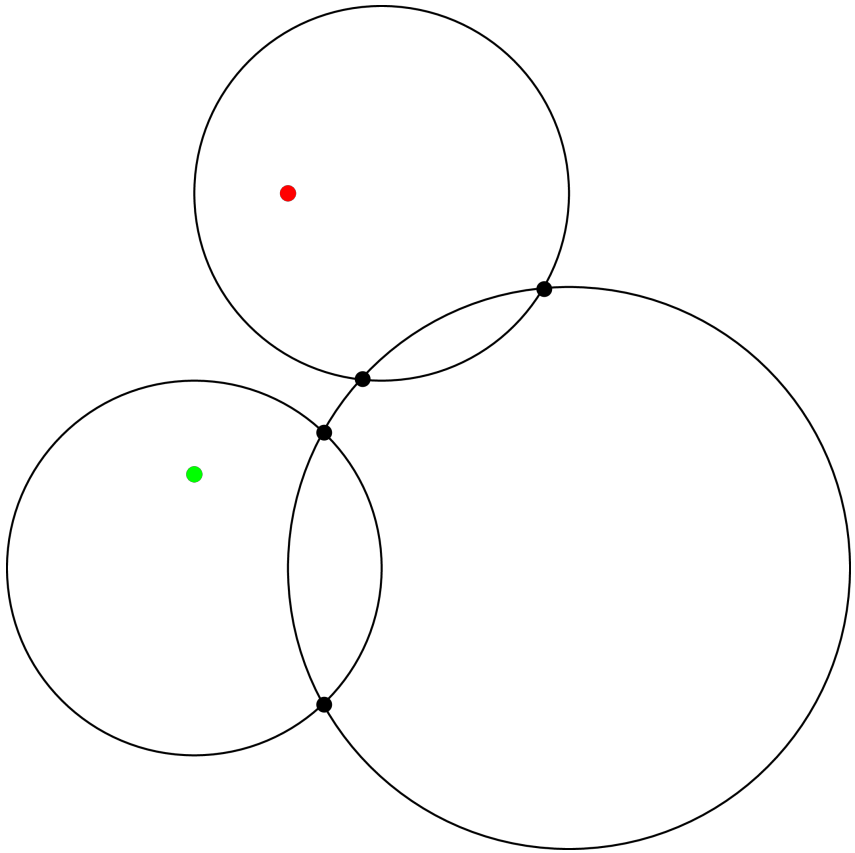
A shortest path between Barney and his mother is straight or travels through intersections of circles along straight lines in between. One can study the Euclidean Shortest Path problem to prove this fact.



# Bearly Made it



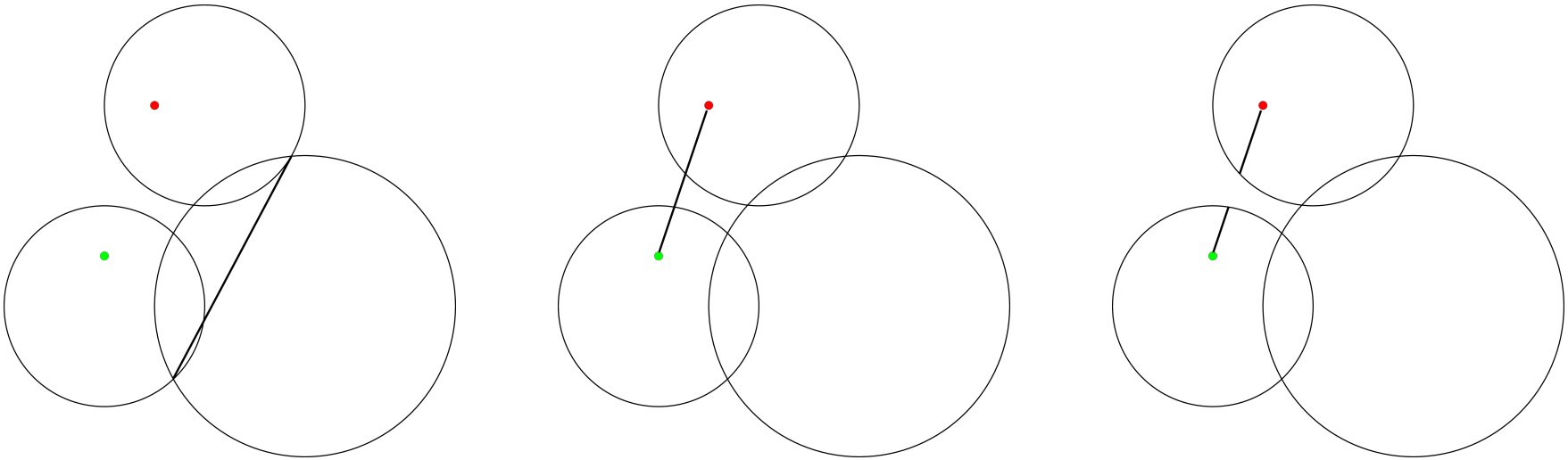
Determine all intersections between circles.



# Bearly Made it



For every pair of intersections and bear to interesection pairs, check whether the line between them is covered completely by circles. If this is the case, consider it a connection in the graph.



# Bearly Made it



Execute a shortest path algorithm on the graph induced by the completely covered lines. The path found is the shortest path from Barney to his mother. If such a path does not exist, we output impossible.