Second Round Theoretical

Tasks

Swiss Olympiad in Informatics

March 3, 2018

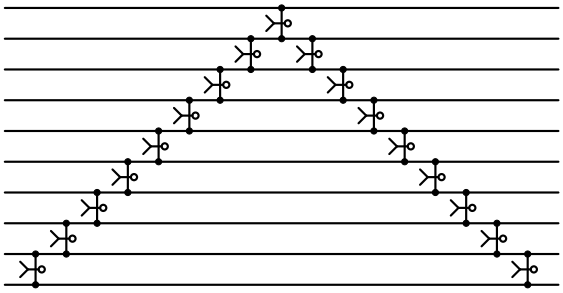
Sushi Belts

Subtask 1: Serving all meals to all customers (10 Points)

The solution depicted in the following subtask is one of the many feasible solutions of this subtask.

Subtask 2: Minimum number of waiters (30 Points)

We need at least $2b - 3$ waiters to be able to serve each meal to each seat in the restaurant. One such placement of waiters for $b = 10$ is shown below.



To see why this pattern of waiters works, note that:

- the first $b - 2$ waiters can bring a meal to belt 1 from any higher-numbered belt
- the waiter in the middle can put a meal from belt 1 to belt 0

Hence after the first $b - 1$ waiters any meal can reach any smaller-numbered belt. Furthermore, note that:

- the waiter in the middle can put a meal from belt 0 to belt 1
- the last $b - 2$ waiters can bring a meal from belt 1 to any higher-numbered belt

Hence the last $b - 1$ waiters ensure that any meal can reach any higher-numbered belt.

Let us prove that the minimum number of waiters that is needed to be able to serve each meal to each seat in the restaurant with $b > 2$ belts is exactly $2b - 3$.

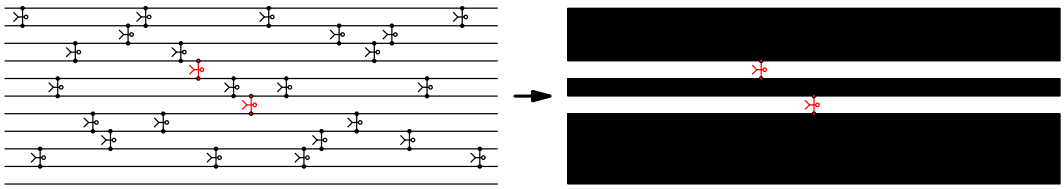
By contradiction assume that we need at most $2b - 4$ waiters. As there are b belts, there are $b - 1$ "rows" where a waiter can be located. Each "row" obviously needs to have at least one waiter. Since we have at most $2b - 4$ waiters, there are at least two "rows" with exactly one waiter in each of them. Let the first such waiter be placed between belts i and $i + 1$ at coordinate x_i and let the second such waiter be placed between belts j and $j + 1$ at coordinate x_j. Without loss of generality we may assume that $i < j$ and $x_i < x_j$.

The waiters divide the belts into three non-empty groups:

1. belts that are to the north of both waiters (numbers 0 through i)

2. belts that are between the two waiters (numbers $i + 1$ through j)
3. belts that are to the south of both waiters (numbers $j + 1$ through $b - 1$)

This is also depicted in the figure below.



To conclude the proof, it is sufficient to note that the meal from belt $b - 1$ now cannot reach seat 0: the only way for this meal to reach the second group of belts is through our second waiter, but at that point we are too far east and we missed the only waiter who could move the meal from the second group of belts into the first group.

Subtask 3: Suboptimal waiters (60 Points)

To start our solution of this subtask, note that if you pick any spot on any belt, the numbers of meals that can reach that spot will always form a contiguous range. This is easy to prove: if a meal that started on belt i can reach a particular point on belt j, any meal that started between belts i and j can get there as well by sitting on its belt for some time and then copying the trail used by the meal from belt i.

Thus, instead of computing a set of meals for each belt (i.e., up to n numbers per belt) we just need to compute two values: the smallest and the largest number of a meal that can reach the eastern end of the belt.

We will process waiters ordered by their x-coordinate (i.e., from the west to the east). Below, we assume that the waiters are already sorted in this order: $i < j$ implies $x_i < x_j$. As we process the waiters, for each belt i we maintain two numbers:

- The lowest belt number $L_{i,j}$ from which a meal can reach belt i using the first j waiters.
- The highest belt number $H_{i,j}$ from which a meal can reach belt i using the first j waiters.

Initially only the meal from belt i can reach belt i and hence $L_{i,0} := i$ and $H_{i,0} := i$.

Now assume we already processed the first j waiters (numbered 0 through $j - 1$), and we know the values $L_{i,j}$ and $H_{i,j}$ for this specific j and all i. Next, we are processing waiter number j. Almost all values $L_{*,j+1}$ and $H_{*,j+1}$ will be the same as the corresponding values $L_{*,j}$ and $H_{*,j}$, only two of them will change. Suppose the waiter moves meals between belts y and $y + 1$. Then clearly only the ranges of meals for these two belts can be affected by waiter j, all other ranges will remain untouched. Moreover, only two of the four values for these belts will change: we now may be able to bring some more

meals from the south to belt y and some more meals from the north to belt $y+1$. Thus, here is the summary how the new values of L and H are computed:

$$\begin{aligned} L_{y+1,j+1} &:= L_{y,j} \\ L_{i,j+1} &:= L_{i,j} \quad \forall i \neq y+1 \\ H_{y,j+1} &:= H_{y+1,j} \\ H_{i,j+1} &:= H_{i,j} \quad \forall i \neq y \end{aligned}$$

Once all the waiters are processed the final output values can easily be calculated as $m_i := H_{i,w} - L_{i,w} + 1$.

To conclude this solution, note that we can maintain the values L and H in place – e.g., instead of explicitly copying all values $L_{*,y}$ to get a new sequence of values $L_{*,y+1}$ we can simply have a single array L and for each waiter we just make the one update to this array in constant time. (We never need the previous values, so we can just overwrite the old value with the new one.)

One possible solution which earns full score looks as follows:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int B, W;
  cin >> B >> W;    // counts of Belts and Waiters

  // read waiters from the input
  vector<pair<int,int> > waiters(W);
  for(int i=0; i<W; ++i) {
    cin >> waiters[i].first >> waiters[i].second; // X and Y coordinates
    --waiters[i].second;  // The belts are numbered from 0.
  }
  sort(waiters.begin(), waiters.end());

  // initialization
  vector<int> highest, lowest;
  for(int i=0; i<B; ++i) {
    highest.push_back(i);   // highest[i] = max meal that can reach belt i
    lowest.push_back(i);    // lowest[i]  = min meal that can reach belt i
  }

  for(int i=0; i<W; ++i) {
    int y = waiters[i].second;  // Y coordinate
    highest[y]   = highest[y+1];
    lowest[y+1] = lowest[y];
  }

```

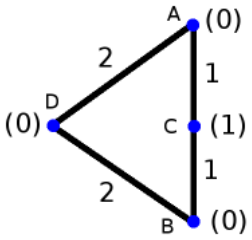
```
32   for(int i=0; i<B; ++i) {
33     cout << highest[i] - lowest[i] + 1;
34     cout << (i==B-1 ? "\n" : " ");
35   }
36 }
```

The sample solution needs to sort the waiters, the rest of the solution is obviously linear. Hence the running time is $O(w \log w + b)$. We store the waiters' locations and two vectors L and H, each of length b, hence the space usage is $O(b + w)$.

Osaka Tube

Subtask 1: Find a counterexample (10 points)

The network is depicted in the following figure. There are two zones ($z = 2$) which are put into parentheses in the figure. The travel times are given next to the edges. Both the zones cost $p_0 = p_1 = 1$ Yen. Now, the only shortest route from A to B passes through C and costs $p_0 + p_1 = 2$ Yen. The route via D is strictly longer, but cheaper, since we only have to pay for the zone 0.



Subtask 2: Circular network (30 points)

Let us assume that the network is not consistent. Then there are two stations A and B such that the (only) shortest route and the (only) cheapest route are internally vertex-disjoint. The station B can now be shifted along the shortest route in the direction away from A, as long as it stays the only shortest route. Note that the other route (whose set of zones is a subset of the original set of zones) stays the only cheapest route. Since there are two possible directions for the shortest route from A, there are at most two candidate nodes B that have to be checked with respect to a fixed node A.

Hence, for a node A and up to two candidate nodes B, we have to (efficiently) determine the length and the price of both the routes from A to B. The length of the shortest route to a node B can be computed in $O(1)$, for instance with a prefix sum array. The prices of both the routes from A to B can be computed in $O(z)$ by computing the number of stations in the respective zones in two arrays of length z and taking the sum of prices of the zones with a positive count.

Both the arrays and the corresponding prices can be easily updated in $O(1)$ after a shift of the node B to an adjacent node B'. This way, both the candidate nodes B for a fixed node A can be found and checked in $O(n + z)$.

Finally, we have to also iterate the node A. When we shift the node A to an adjacent node, we might have to shift the candidate nodes B (in $O(1)$ per shift). In total (over all considered nodes A), the two candidate nodes B revolve at most once around the circular network, i.e., there are at most $O(n)$ shifts. The runtime complexity of the whole solution is thus in $O(n + z)$ and the memory complexity cannot be more.

Grading scheme:

- Explaining how to compute the length of a route on the cycle: 5 points.

- Explaining how to compute the prices of a route on the cycle: 5 points.
- Some observation or condition when a circular network is consistent: 5 points.
- The correct runtime: 5 points. ($z \in O(n)$ can be assumed.)
- The optimal runtime: 10 points.

Subtask 3: Transport network of Osaka (60 points)

At first, we compute the lengths of the shortest routes between all pairs of stations. Using the Floyd-Warshall algorithm, this step can be implemented in $O(n^3)$.

Now, we have to determine the lengths of the shortest cheapest routes between all pairs of stations. To this end, we consider all subsets of zones and for each subset of zones, we construct the subgraph containing all the stations belonging to one of the zones in the subset. Then, we compute the lengths of the shortest routes between all pairs of stations in the constructed subgraph. Finally, for each pair of stations such that one is reachable from the other in the subgraph, we update the length of the shortest cheapest route by checking if the current subset of zones provides a cheapest route so far or if there is possibly a shorter route with the same (cheapest) price. This way, we determine the lengths of the shortest cheapest routes between all pairs of stations. Using the Floyd-Warshall algorithm, this step can be implemented in $O(n^3)$ per subset, i.e., in $O(2^z n^3)$ in total.

At the end, we have to check for each pair of stations if the length of the shortest route equals the length of the shortest cheapest route.

The overall runtime is in $O(2^z n^3)$ and the memory complexity in $O(n^2 + z)$.

Note. The overall runtime can be reduced by a factor of z by processing the subsets of zones in the order of increasing cardinality. Then for a subset Z of zones, a zone $i \in Z$ is to be found such that there are at most $\frac{n}{z}$ stations belonging to that zone (if there were no such zone, then there would be more than n stations in total — contradiction). With this, the Floyd-Warshall table for the subset $Z \setminus \{i\}$ can be extended to the Floyd-Warshall table for the subset Z by inserting the stations in the zone i (there are at most $\frac{n}{z}$ such stations and each of them can be inserted in time $O(n^2)$ – standing for the inner two loops of Floyd-Warshall algorithm). This way, the overall runtime is reduced to $O(2^z n^2 \frac{n}{z}) = O(2^z n^3/z)$. However, the memory complexity increases to $O(2^z n^2)$. The memory usage can then be reduced to $O(2^z + z n^2)$ by generating the dependency tree and traversing it in depth-first search order. (The tree has depth $\leq z$.)

Note. When Dijkstra algorithm with Fibonacci-heap is started from all the nodes to determine the shortest routes between all pairs of stations instead of Floyd-Warshall algorithm, the overall runtime can be reduced to $O(2^z(mn + n^2 \log n))$. The memory complexity remains unchanged.

Grading scheme:

- Explaining how to compute the lengths of the shortest routes between all pairs of stations: 10 points.

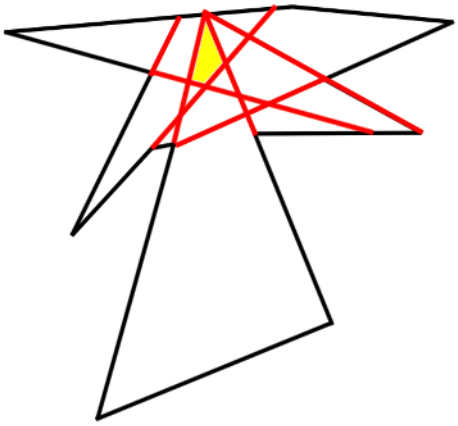
- Explaining how to compute the lengths of the shortest cheapest routes between all pairs of stations: 40 points.
- The correct runtime of a correct solution: 10 points.

Dandyshow

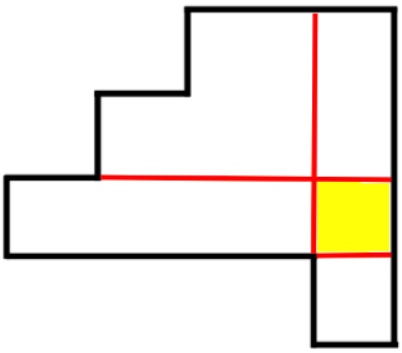
Subtask 1: (10 Points)

Makr all places, where Dandy can organise his party.

1.



2.



(how you find such a solution will be explained in the next exercise).

Subtask 2: Solve the rectilinear case (30 points)

We are given the edges of a polygon in clock-wise direction. If we go along this rectilinear polygon in clock-wise direction we can distinguish between 4 types of edges with directions towards (up, down, left right). Realise, that you can extend each of these rays to a line and that it is impossible for Dandy to do his show on one side and possible on the other.

If one then covers for each extended line the side of the polygon, one obtains exactly the area, on which Stofl can do his party (since for each edge this condition is both sufficient and necessairy).

One thereby only has to regard the right-most $\downarrow$, the left-most $\uparrow$, the down-most $\leftarrow$ and the up-most $\rightarrow$. If the right-most $\downarrow$ is to the right of the left-most $\uparrow$, then the entire area is covered and Dandy cannot do his party. Analogously, if the down-most $\leftarrow$ is

below the up-most → Dandy cannot do his party either. If this is not the case though, Dandy can do his Party anywhere in the rectangle, for example in it's center point:

$$\left(\frac{(\text{leftmost} \downarrow) + (\text{rightmost} \uparrow)}{2}, \frac{(\text{downmost} \leftarrow) + (\text{upmost} \rightarrow)}{2}\right)$$

(Depending on the definition one can distinguish between the cases where only a point or a line is left unconvered. Mentioning this only gave Brownie points.)

You were given 30 points for a correct idea in linear runtime, with available pseudocode and correct runtime analysis and were deduced points if something was missing respectively.

Slower solution

Some participants assumed, that the coordinates where whole numbers and applied a flood-fill extension for each edge, until the flood hit another edge. With the assumption, that the coordinates were whole numbers you could receive 10 points if you wrote down everything down correctly and included pseudo-code. Your solution would then run in $O(n \cdot X \cdot Y)$.

One can use the same idea and use coordinate-compression on the coordinates (sort the edge points along their x and y-axis respectively).
In the worst-case scenario all points would have different coordinates and the algorithm would take $O(n^2)$. This solution received 20 points.

Subtask 3: Solve the general case (55 points)

In general, the principle of covering still applies. Except that the remaining area is no longer a rectangle, but a convex polygon (it must be convex, since these half-surfaces (i.e. inequalities of the form $a \cdot x + b \cdot y \leq c$) is a convex set and the cutting of convex sets is still convex). But how can this intersection of these rays be computed efficiently? The trick is similar similar to the convex hull idea. First we sort the rays by their direction, i.e. by their angle to the x-axis. We can calculate this angle either by a case distinction with sin and cos or something easier with the help of the scalar product.

$$a \cdot b = |a||b||b| \cos(\measuredangle) \Rightarrow \measuredangle = \arccos\left(\frac{a \cdot b}{||a||b|}\right)$$

Since acos is only defined to $[0, \pi)$, you still have to make a small case distinction. So for any ray we get the angle as follows:

$$\measuredangle(s) = \begin{cases} \arccos\left(\frac{s \cdot t}{|s||t|}\right), \text{falls } y(s) \geq 0 \\ 2\pi - \arccos\left(\frac{s \cdot t}{|s||t|}\right), \text{falls } y(s) < 0 \end{cases}$$

where t is the vector of the x-axis, so: $t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

The exact implementation of the angle was not necessary to get the full score but there were bonus points if you got it right.

Sorting edges by their angle can be done in $O(n \log n)$ (In our case the number of vertices is = n = m = number of edges).

If there are rays with the same angle, as with subtask 2 we can only use those that cover the most. In the graphically solved tasks, the rays that are sorted out in this way are marked with an '. (This remaining beam can be found out e.g. by taking a beam which is turned 90 degrees clockwise. and the intersections of the equiangular beams. The beam that has the intersection point furthest in the direction of the rotated beam is this remaining beam).

Now we look at the two edges with the smallest angle. Realize that both are less than 180° apart, since two consecutive edges in a simple polygon have less than 180° intermediate angle. Thus, for each edge there is another one that is less than 180° apart. is. From these two rays we calculate the intersection point and set it to an empty intersection deque. We insert the two beams into an empty beam deque (first the one with the smallest angle and then the one with the second-smallest angle). A deque is simply an array that accesses $O(1)$ and amortized $O(1)$ appending in front and behind. (See also $std :: deque$).

At the end, the intersection deque will have all corners of the convex polygon, which marks the area where Dandy can do his show (the direction of the rays is stored on the ray deque).

To understand the next step, it is best to look at the already solved examples.

We iterate over the remaining rays (by angle, starting with the third smallest beam). angle). Now we have to cut each of these I ray with the so far calculated convex amount on which Dandy can do his show. The backmost ray of the ray deque is now called A for simplicity's sake. and the rearmost intersection of the intersection deque a. Now we calculate the intersection of I and A (which always exists, except in the corner case where A and I exactly point to each other, that will later be is handled). If the new intersection i comes towards the edge A **after** intersection a, we can simply add the new intersection to the intersection deque at the back and add I to the ray deque at the back. This case occurs for example at step (3) on (4) in example 1. I have drawn the values i, I, a, A there for the sake of simplicity. If the new intersection i comes towards the edge A **before** intersection a, we remove A and a from their respective deques and try again I until it works. If I with all edges **before** the respective intersection of the intersection deques, there is no solution (see example 3).

That's basically the whole idea, but cutting is something else more complicated once you look at rays that have more than 180° angle to the the first ray. These rays cut the convex set in two places (once in front and once behind). In the respective examples, the pink points are the points that intersect with the beam at the back of the ray deque and the blue points are the points that intersect with each other. with the ray from the front of the ray deque. So you just do the same thing here only once from the front and once from the back. in the back. In example 2 you can see both of these cuts very nicely from Step (6) on (7) or for example 1 from step (7) on (8).

Now you just have to look at the corner case where Dandy put on his show on a line (or a point). But this wasn't required, you so you could assume that dandy's party had to have some volume. In principle, the whole thing still works, only now it can happen that pink and blue dots can be in the same place. The convex quantities described by the inequalities, respectively by the intersections and rays of the deque still form the edges of this convex set. Example 4 should explain this better.

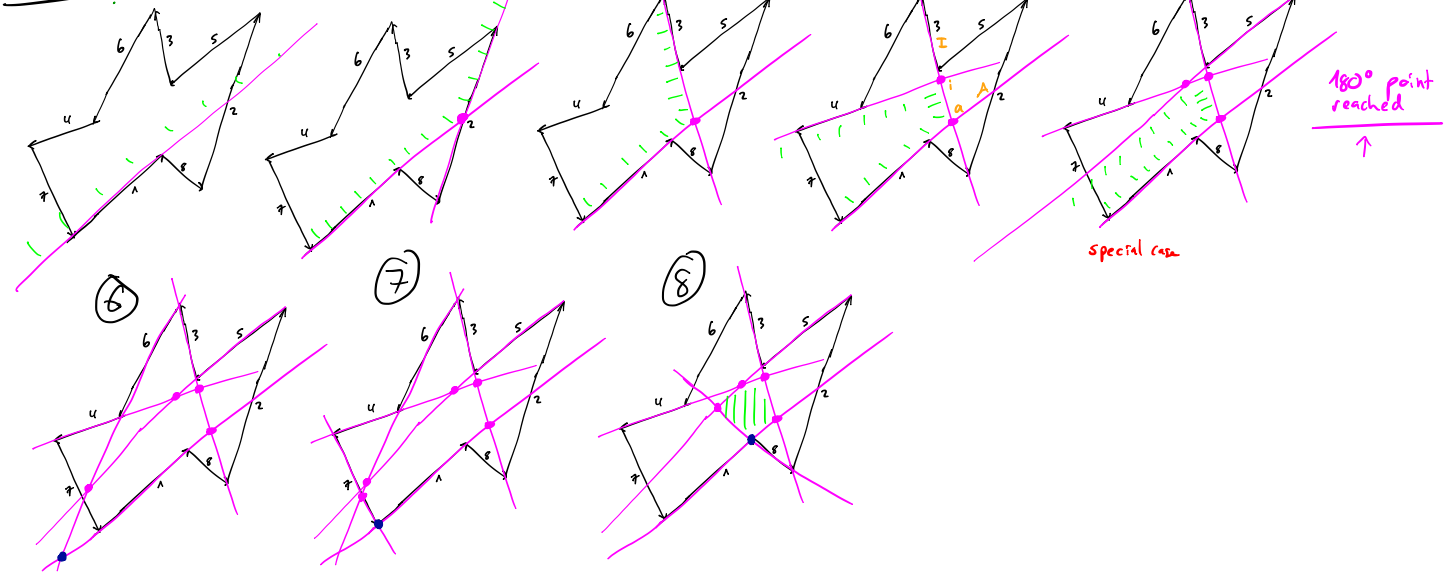
Runtime analysis: Sorting is in $O(n \cdot log(n))$. If the rays had been already sorted, the rest of the algorithm would run in $O(n)$, since each edge added to the stack at most twice, or twice from the stack (twice because each edge has both a purple intersection and a blue intersection. can). Considering this argument about the term is called amortization. The best you look at the runtime of Graham's convex hull algorithm, because it uses exactly the same payback. If you are new to amortization, consider how $std :: vector$ requires an amortized term of $O(1)$ for $push_back$.

A correct solution was awarded with 20 points. Additionally there was 10 points for a solution in $O(n^2)$ Additionally, there was 30 points for a laugh in $O(n \cdot \log n)$

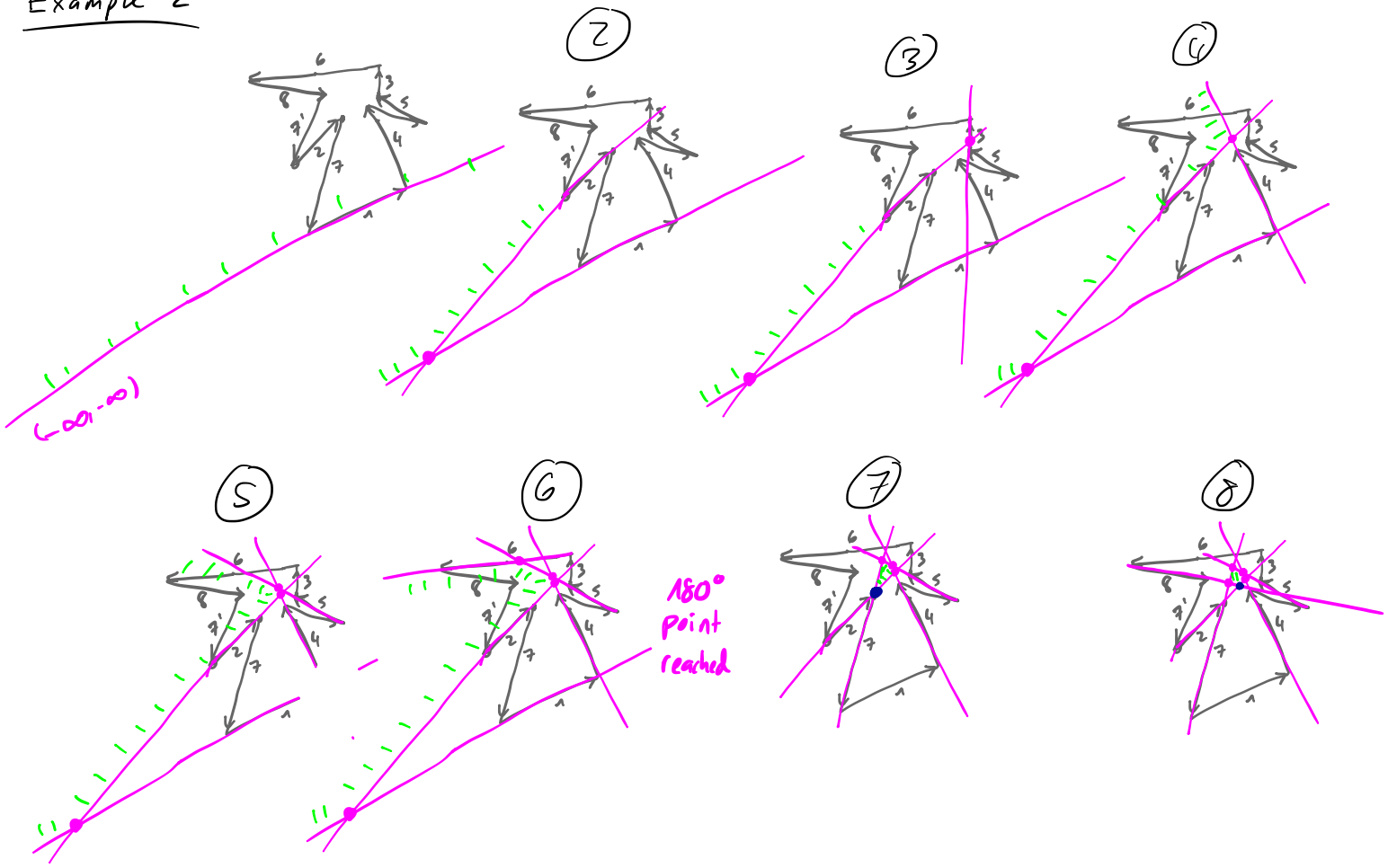
Pseudocode:

```
input: tuple<int,int,int,int>[] rays
algorithm:
sortedrays := sort_by_angle_to_x_axis (rays)
deque<tuple<int,int,int,int> > raystack
deque<pair<int,int> > intersectionstack

raystack = sortedrays[0];
firstangle = angle(sortedrays[0], x-axis)

foreach iray : sortedrays [2..N-1] {
  while(true) {
    if (raystack.size < 2) exit('too bad')
    intersectpoint= intersect (iray, raydeque.back)

    if intersectpoint comes after the intersectionstack.back along raydeque.back
      intersectionstack.push_back(intersectpoint)
      raystack.push_back(iray)
      break;
    raystack.pop_back
    intersectionstack.pop_back
  }
  if(angle (iray,x-axis) - 180 >= firstangle) {
    while(true) {
      if (raystack.size < 2) exit('too bad')
      intersectpoint= intersect (iray, raydeque.front)

      if intersectpoint comes before the intersectionstack.front along raydeque.front
        intersectionstack.push_front(intersectpoint)
        raystack.push_front(iray)
        break;
      raystack.pop_front
      intersectionstack.pop_front
    }
  }
}
```

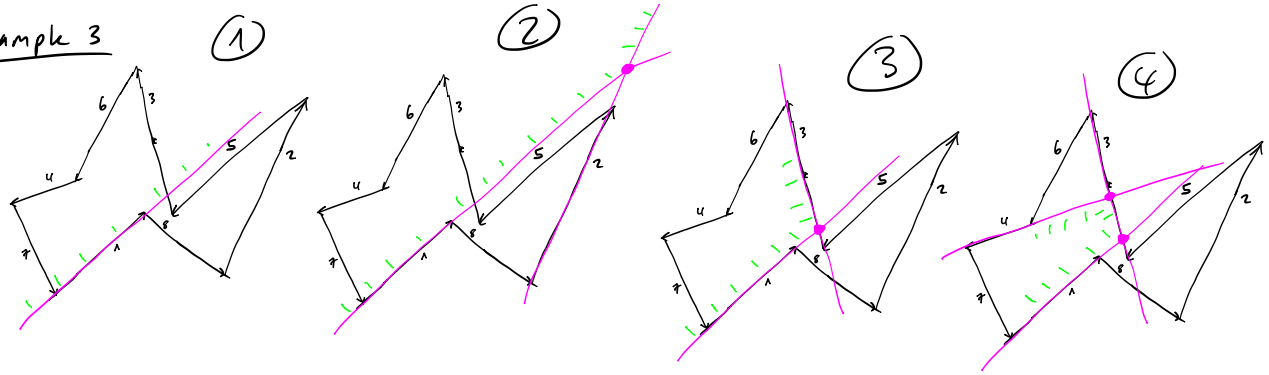
Example 1

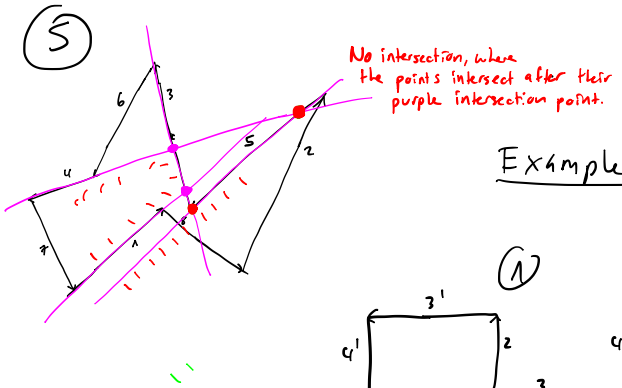


Example 2

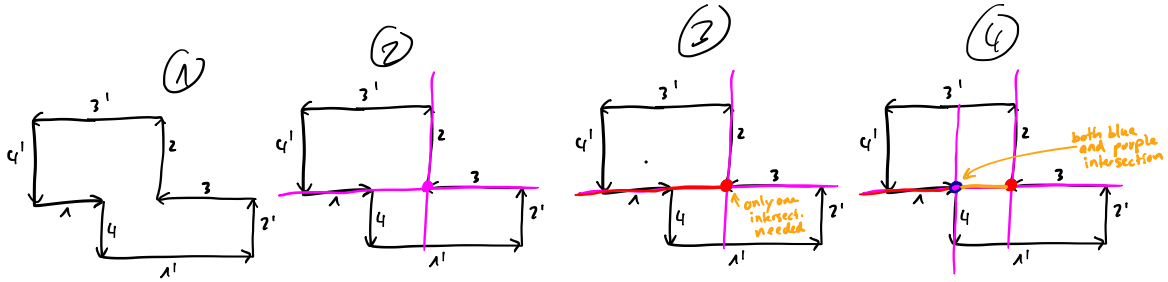


Example 3



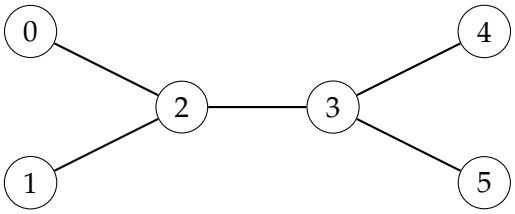


Example 4 Special case 180° →←



Erdgasversorgung

Subtask 1: (10 Points)

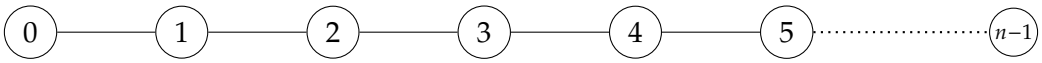


The leak can be found with 3 queries. First we look in which direction the gas flows between vertices 2 and 3. Assuming it flows to the left, i.e. the leak is either in vertex 0, 2 or 1. To determine the exact position we also check 0-2 and 1-2. Now there are 3 possibilities:

- Both flow to the right: The leak is at vertex 2.
- In line (0,2) it flows to the left: The leak is at vertex 0.
- In (1,2) it flows to the left: The leak is at vertex 1.

If the gas in (2,3) flows to the right, we proceed analogously.

Subtask 2: (30 Points)



We can find the leak with binary search in $\log n$ queries. If we make a query for the line located in the middle between vertex $m_0 = \left\lfloor \frac{n-1}{2} \right\rfloor$ and $m_0 = \left\lfloor \frac{n-1}{2} \right\rfloor + 1$ the number of vertices where the leak may be located is halved. So with each query we can halve the number of vertices that can contain the leak. After $\log n$ queries we have only one vertex left, which is the one with the leak.

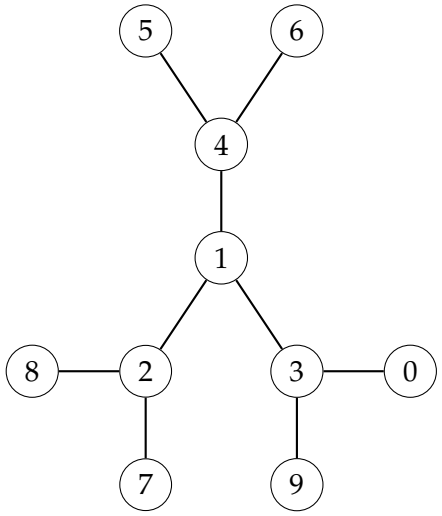
The algorithm needs to compute the two middle vertices before each request and update the interval in which the leak can be found after the request. This can be done in constant running time. Since $\log n$ requests are required, the running time of the algorithm is also $O(\log n)$. Note how we use half-open intervals in the implementation of the binary search. If, e.g., $l = 1$ and $r = 3$, then the elements in the interval are $1, 2$.

$l \leftarrow 0$
$r \leftarrow n - 1$
while $r - l > 0$ **do**
    $m_0 \leftarrow l + (r - l)/2$
    $m_1 \leftarrow l + (r - l)/2 + 1$

```
    answer ← Query(m_0, m_1)          ▷ Here we actually go check a pipe
    if answer = m_0 then
        r ← m_1
    else
        l ← m_1
    end if
end while
print "The Leak is in vertex $l"
```

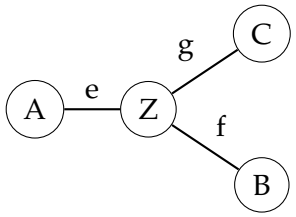
Subtask 3: (60 Points)

We would like to do something similar to the previous subtask, where we could remove half of the remaining vertices with each query. However, this approach no longer works. Let's look the following example:



No matter which line we look at, it is not possible to exclude half of the vertices. However, it is possible to exclude a third of the vertices if we consider a line connected to vertex 1. We can therefore ask ourselves whether it is always possible to exclude a third of the vertices? And the answer is indeed yes and we can prove it as follows:

Let us assume that it would not be possible to find a line that excludes at least one third of the vertices. Then e is the line where the side with fewer vertices contains as many vertices as possible. Let's visualize this:



Let A be the side with fewer vertices. We know that $|A| < \frac{1}{3} \cdot n$, since we assumed that there is no line where at least a third of the vertices could be excluded.

Note that Z is not necessarily a vertex of degree 3, $|C|$ or $|B|$ can also be zero. Now we know that $|A| + |B| + |C| + 1 = n$ (the +1 is the Z vertex) and that $|A| < \frac{1}{3} \cdot n$. So $|B| + |C| + 1 > \frac{2}{3}n$. Now at least one those two cases is true:

- $|B| > \frac{1}{3}n$: In this case we would have selected the f line instead of the e line, because it has at least one vertex more on the side with less than $\frac{1}{3}n$ vertices at f, namely Z. We chose e but so that A is maximum, that's a contradiction.
- $|C| > \frac{1}{3}n$: Same argument, but with the g line instead of f.

So in both cases it leads to a contradiction, from which we conclude that it is impossible that there is no line with which we cannot exclude at least one third of the vertices.

So we can always find a line with which we can get at least $\frac{1}{3} \cdot n$ vertices. If with every query we get rid of $\frac{1}{3} \cdot n$ vertices, then we won't ask more than $\lceil log_{3/2}(n) \rceil = \left\lceil \frac{log(n)}{log(3/2)} \right\rceil$ queries. We can also omit rounding, because it will add at most one, which is a constant term.

Now we have to design an algorithm to find such a line. To do this, we can start a depth search (DFS) that calculates how many vertices are on both sides for each line. The line where the difference between the two is minimal is the optimal one. The running time of DFS is $O(n)$. After each request we have to do a depth search on the maximum $\frac{2}{3}n$ remaining vertices again. So the total running time is $O(n + \frac{3}{2}2n + (\frac{3}{2})^2\dot{n} + \dots) = O(n)$.

There is, by the way, a better solution using a technique called "Centriod Decomposition"[1]. With this technique, the dominant factor can be reduced from $\frac{1}{log(3/2)}$ to $\frac{1}{\log \frac{1}{2}(1+\sqrt{5})}$. However, this solution was not needed to score full points and should only serve as a comment for interested parties.

[1] `https://threads-iiith.quora.com/Centroid-Decomposition-of-a-Tree`