Second Round Theoretical

Solutions

Swiss Olympiad in Informatics

March 4–7, 2022

Consensus

Task Idea	Joël Huber
Task Preparation	Johannes Kapfhammer
Description English	Joël Huber
Description German	Jan Schär
Description French	Florian Gatignon
Solution	Johannes Kapfhammer

Subtask 1: Solve the example (10 Points)

a) With $k = 3$ the largest possible consensus is 6 and can be reached as follows.

- $[\underline{1}, \underline{6}, 1, \underline{8}, 0, 3, 3]$. Invite $0, 1, 3$ and agree to 6.
- $[\underline{6}, \underline{6}, \underline{1}, 6, 0, 3, 3]$. Invite $0, 1, 2$ and agree to 6.
- $[\underline{6}, \underline{6}, 6, 6, \underline{0}, 3, 3]$. Invite $0, 1, 4$ and agree to 6.
- $[\underline{6}, \underline{6}, 6, 6, 6, \underline{3}, 3]$. Invite $0, 1, 5$ and agree to 6.
- $[\underline{6}, \underline{6}, 6, 6, 6, 6, \underline{3}]$. Invite $0, 1, 6$ and agree to 6.
- $[6, 6, 6, 6, 6, 6, 6]$. Everyone is of the opinion that the camp should have length 6.

b) With $k = 2$ the largest possible consensus is 8 and can be reached as follows.

- $[\underline{1}, 6, 1, \underline{8}, 0, 3, 3]$. Invite $0, 3$ and agree to 8.
- $[\underline{8}, \underline{6}, 1, 8, 0, 3, 3]$. Invite $0, 1$ and agree to 8.
- $[\underline{8}, 8, \underline{1}, 8, 0, 3, 3]$. Invite $0, 2$ and agree to 8.
- $[\underline{8}, 8, 8, 8, \underline{0}, 3, 3]$. Invite $0, 4$ and agree to 8.
- $[\underline{8}, 8, 8, 8, 8, \underline{3}, 3]$. Invite $0, 5$ and agree to 8.
- $[\underline{8}, 8, 8, 8, 8, 8, \underline{3}]$. Invite $0, 6$ and agree to 8.
- $[8, 8, 8, 8, 8, 8, 8]$. Everyone is of the opinion that the camp should have length 8.

(That organizer who thinks the camp should have length 0 though. . .)

Subtask 2: Design an Algorithm (90 Points)

There are two steps to the full solution. First, we realize that the answer is always the $\lfloor (k-1)/2 \rfloor$-th largest number. Then, we develop an effient algorithm to compute that.

The answer is always the $\lfloor(\mathbf{k}-1)/2\rfloor$-th largest number

To clarify the notation, $\lfloor\cdot\rfloor$ means rounding down to the next integer. In this case $\lfloor (k-1)/2 \rfloor$ stands for the largest integer that is smaller than or equal to $(k-1)/2$.

Let X be the $\lfloor (k-1)/2 \rfloor$-th highest number.

As with most proofs there are two parts:

(a) X is a lower bound (the answer is $\geq X$) and

(b) X is an upper bound (the answer is $\leq X$).

Lower Bound We can prove the this by construction.

- Take the k highest numbers and make an agreement towards X. This is possible because $\lfloor(k-1)/2\rfloor + 1$ numbers will be $\geq X$, and $\lfloor(k-1)/2\rfloor + 1 = \lfloor(k+1)/2\rfloor \geq k/2$; as well as $k - \lfloor(k-1)/2\rfloor - 1 = (k-1) - \lfloor(k-1)/2\rfloor = \lceil(k-1)/2\rceil \geq k/2$.

  Another, possibly clearer, way is to make a case distinction between even k and odd k; that way one can avoid dealing with the rounded expressions.

- As long as there is not a consensus, take $k-1$ numbers which are X and one that is not X and set them all to X (for this to work we need $k \geq 2$).

After this process ends all the numbers have value X.

Upper Bound We still need to argue that there is no better solution. Let's do a proof by contradiction. Suppose, for example, that the best answer is Y, where $Y > X$. Let's label all values $\geq Y$ as red and all values $< Y$ as green. By definition of Y there are initially $< \lfloor(k-1)/2\rfloor$ red numbers. Since Y is a consensus during some mediation the number of red values must go from $< \lfloor(k-1)/2\rfloor$ to $\geq \lfloor(k-1)/2\rfloor$. Let's look at the first mediation where this happens. The agreement was made on a red number Z where $Z \geq Y$. However as there were only $\lfloor(k-1)/2\rfloor$ red numbers before, The majority of values is $< Y$ therefore such proposal would be rejected – contradiction. Therefore no answer can be $> X$, or in other words, all possible answers must be $\leq X$.

Both statements together imply that the answer must be exactly X.

Computing the i-th largest number

Now that we know what we want, we can look at different ways to compute it efficiently. Let $i = \lfloor(k-1)/2\rfloor = O(k)$ for the following analysis.

The following solutions solve the subtasks in $O(n)$ time and $O(1)$ space:

a) For $k = 3$, this means finding the second largest number. This can be done in a single iteration through the array maintaining the largest and second largest value.

b) For two different options, counter the number of 8's and the number of 9's. If the number of 9's is larger or equal to $i+1$, then the answer is 9, otherwise it's 8.

The following solutions work for the general task:

$O(n \log n)$ time/$O(n)$ space *Sorting the input values and looking at the i-th largest number.* Sorting the array can be done in $O(n \log n)$ time. Since the input is only given in read-only memory and sorting needs to write to the array, this uses $O(n)$ memory.

$O(n \log k)$ time/$O(k)$ space *Only keeping the i largest numbers.* We don't need to remember all values, we only need to store the i largest ones. We could do this by keeping a `priority_queue` of size k. When iterating through the array we push the new value and pop the smallest one. The size of the priority queue never exceeds k, therefore push and pop run in $O(\log k)$ and the memory is in $O(k)$.

$O(n)$ time/$O(n)$ space *Using the median of medians algorithm.* The function `nth_element` of the C++ standard library can compute the k-th element of a vector in $O(n)$ time. The Median of medians algorithm would be a possible implementation. As the list needs to be rearranged to compute it, this requires $O(n)$ memory.

$O(n)$ time/$O(k)$ space *Using Median of medians to keep the i largest numbers.* We can reduce the memory by combining the median of medians idea with the priority queue idea: Take the first i values. Then repeatedly read i more values, compute the median and only keep the i largest values. That way we only need to store $2i = \mathcal{O}(k)$ values, and compute the median for $\lceil n/i \rceil$ chunks each taking $\mathcal{O}(2i)$ time – which means the total running time is $\mathcal{O}(n)$.

Scientific Conference

Task Idea	Johannes Kapfhammer, Ema Skottova
Task Preparation	Petr Mitrichev
Description English	Petr Mitrichev
Description German	Jan Schär
Description French	Alberts Reisons
Solution	Petr Mitrichev

Subtask 1: Solve an example (10 points)

In this example, there is exactly one set of scientists that Binna could invite: A, D and E. They have not published any papers together, so the popularity of each scientist at this conference would be 0, and $1 + 0 < \frac{3}{2}$.

Subtask 2: Very few papers (20 points)

We will describe the solution for this subtask as part of the solution for the next subtask.

Subtask 3: Few papers (30 points)

Idea

There are multiple approaches that can be used for these subtasks. Let us describe the most direct one.

First, let us try to invite all mouse scientists to the conference. If the conditions from the problem statement are satisfied, then we have found the solution. If they are not, then there exists a mouse scientist with popularity of at least $\frac{n}{2} - 1$.

Let us remove this mouse scientist and all papers they have published from consideration. We now have a set of $n' = n - 1$ mouse scientists, and the number of papers they have published have been reduced by at least $\frac{n}{2} - 1$: $m' \leq m - (\frac{n}{2} - 1)$.

If the remaining set of mouse scientists satisfies the conditions from the problem statement, then we have found the solution. If not, then there exists a mouse scientist with popularity of at least $\frac{n'}{2} - 1 = \frac{n-1}{2} - 1$.

Now we will remove this mouse scientist and their papers from consideration as well. We now have a set of $n'' = n - 2$ mouse scientists, and the number of papers they have published is $m'' \leq m - (\frac{n}{2} - 1) - (\frac{n-1}{2} - 1)$.

We can continue this process until we either get a good set of mouse scientists to invite, or remove all scientists from consideration. However, if we have removed all scientists from consideration, then the number of remaining papers, which is 0, satisfies the inequality $0 \leq m - \sum_{i=1}^{n}(\frac{i}{2} - 1) = m + n + \frac{1}{2}\sum_{i=1}^{n} i = m + n - \frac{n(n-1)}{4}$, so $m \geq \frac{n(n-1)}{4} - n = \frac{n(n-5)}{4}$.

However, in these two subtasks we know that $m < \frac{(n-5)^2}{4} \leq \frac{n(n-5)}{4}$, which contradicts the inequality above. This means that in these two subtasks the approach with removing the mouse scientists from consideration one by one will always find a valid set of mouse scientists for the conference before removing all mouse scientists.

Implementation

In order to implement this algorithm, we need to repeatedly do the following: find the vertex with the highest degree in an undirected graph, and then either stop or remove this vertex with all adjacent edges from the graph.

This can be done by putting all (degree, vertex) pairs into a balanced binary search tree (such as `std::set`), which allows to find the biggest element, and whenever we remove a vertex we update the degrees of the adjacent vertices and remove and re-add the corresponding pairs to the tree. Since the tree does each operation in $O(\log n)$, and we do at most $m + n$ operations, the total running time is $O((m + n) \log n)$, and we use $O(m + n)$ memory (to keep the graph from the input).

We can get rid of the $\log n$ factor by choosing a more appropriate data structure that would allow to find the vertex with the highest degree repeatedly. For example, we can put all vertices into a vector of vectors, where the outer vector is indexed by the degree of the vertex. Every time we decrease the degree of a vertex because of removing some edge, we will push it to the vector corresponding to its new degree. Ideally we would also remove it from the vector corresponding to its old degree, but removing elements from vectors is slow, so we will keep it there, and instead check if the degree matches when traversing that vector.

This will cause some vertices to appear in more than one vector by degree, but the total number of pushes will still be $O(m + n)$ (because we push n vertices initially and then at most m times when an edge is removed), so the running time and memory usage will be $O(m + n)$.

Pseudocode

```
n, g = read_graph_as_adjacency_lists()
deg = [g[i].size() for i in 0 .. n-1]
by_deg = [[], [], ..., []]
for i in 0 .. n-1:
  by_deg[deg[i]].push(i)
removed = [0, 0, ..., 0]
k = n
for i in n-1 .. 0:
  for a in by_deg[i]:
    if removed[a] == 1 or deg[a] != i:
      continue
    if 2 * (i + 1) < k:
      return [i for i in 0.. n-1 if removed[i] == 0]
    removed[a] = 1
    k -= 1
    for b in g[a]:
      if removed[b] == 0:
        deg[b] -= 1
        by_deg[deg[b]].push(b)
```

Subtask 4: Many papers (40 points)

Idea

The approach described above does not work when we have more edges, so we need a different approach for this subtask. The example from the first subtask hints at the following

Lemma 1: If there is a valid set of mouse scientists to be invited to the conference, there always exists a valid set of exactly 3 mouse scientists to be invited to the conference.

Proof: Consider a valid set S of mouse scientists to be invited to the conference, $|S| = k$. Take any scientist $a \in S$. Since the popularity of a is less than $\frac{k}{2} - 1$, the set $T \subset S$ of those scientists in S who have not published papers together with a has more than $\frac{k}{2}$ elements. Now take any scientist $b \in T$, and by the same argument the set $U \subset T$ of those scientists in T who have not published papers together with b has more than 0 elements. Finally, take any scientist $c \in U$. We can see that the scientists a, b and c have published no papers together, and therefore can form a valid conference.

Therefore we just need to find an *anti-triangle* in the given graph: three vertices with no edges between them.

Implementation

We can just check all possible options for an anti-triangle with three nested loops. In this subtask $m = \Omega(n^2)$, therefore we can represent the graph as the adjacency matrix without increasing the asymptotical time or memory usage, and then we can check the existence of every edge in $O(1)$, making the entire solution run in $O(n^3)$, using $O(n^2)$ memory.

There are algorithms to find an anti-triangle faster, for example `https://en.wikipedia.org/wiki/Triangle-free_graph` explains how to find a triangle in $O(n^{2.373})$ using fast matrix multiplication, and we can simply apply it to the complement of the graph to find an anti-triangle, but this was not required to get full score in this subtask.

Pseudocode

```
n, a = read_graph_as_adjacency_matrix()
for i in 0 .. n-1:
  for j in i+1 .. n-1:
    for k in j+1 .. n-1:
      if a[i][j] == 0 and a[i][k] == 0 and a[j][k] == 0:
        return {i, j, k}
return "No solution"
```

Previous subtasks revisited

The same approach of just looking for anti-triangles can also be used in subtasks 2 and 3. However, our $O(n^3)$ implementation is much slower than $O(m + n)$ that we achieved with a dedicated solution for those subtasks.

However, it is also possible to find an anti-triangle in $O(m + n)$ when $m < \frac{(n-5)^2}{4}$. One approach inspired by the proof of `https://en.wikipedia.org/wiki/Tur%C3%A1n%27s_theorem` is: take any two vertices which are not connected. Either there is an anti-triangle formed by these two vertices and some other vertex, or the number of edges adjacent to them is at least $n - 2$. Remove those two vertices together with all adjacent edges from the graph and repeat.

Similar to the other solution for subtasks 2 and 3 described above, either this algorithm will find an anti-triangle at some point, or the number m of edges in our graph has to be at least $(n-2)+(n-4)+\cdots \geq \frac{1}{2}(((n-2)+(n-4)+\ldots)+((n-3)+(n-5)+\ldots)) = \frac{1}{2}\sum_{i=1}^{n-2} i = \frac{(n-1)(n-2)}{4}$.

The last condition contradicts $m < \frac{(n-5)^2}{4}$, therefore this algorithm will always find an anti-triangle in these subtasks. We can implement it to run in $O(m + n)$ if we represent the graph using adjacency lists and have a separate boolean array that marks the vertices that were removed. Since we process each vertex only once, we can simply iterate over its adjacency list to find the number of edges that are leading from it to non-removed vertices.

There are also other approaches to find an anti-triangle that work for graphs with few edges (they are called *sparse* graphs). Some of them work only for subtask 2, but not for subtask 3, and this was the reason for the existence of subtask 2.

Oneway Portals

Task Idea	Jan Schär, Joël Huber
Task Preparation	Joël Huber
Description English	Joël Huber
Description German	Joël Huber
Description French	Mathieu Zufferey
Solution	Joël Huber, Daniel Rutschmann

This was without a doubt the hardest problem of the problemset. Also note that this solution writeup is very technical and formal, you don't need to be that technical in writing your solution for 2T, as time is limited there.

Let's start with some general observations first. Let $f(v)$ again be the distance the length of a shortest path from v to the root (as in the problem statement), and let $d(u, v)$ be the length of a shortest path from u to v (or ∞ if no such path exists). Also, let's call one way of adding portals a "configuration".

Note that minimizing $\frac{\sum_{v \in S} f(v)}{|S|}$ is the same as minimizing $\sum_{v \in S} f(v)$, thus instead of the average, we can focus on the sum.

Observation 1: In an optimal configuration, each portal used by a shortest path will connect a vertex with the root.

Proof: Suppose in an optimal configuration, there is a portal that doesn't go to the root. Let this portal go from u to v. Let's replace this portal with a portal connecting u and the root and thus get a new configuration. Now look at a vertex w where in the old configuration, there was a shortest path using the portal from u to v. Then in the new configuration, there is a new shortest path whose length is $f(v)$ shorter than the path in the original graph. Thus for any vertex w with a shortest path passing through this portal in the old configuration, $f(w)$ has been reduced. Now, we only need to show that there is no vertex w, such that $f(w)$ has increased. Indeed, this is rather easy to proof: Note that the only thing that we changed was the portal between u and v. As said, vertices w with a path from u to v have their $f(w)$ reduced, vertices w where no shortest path passed through this portal don't have their f increased as these shortest paths haven't changed.

These two observations give us a clear rules on what we can do. All the portals we are going to use connect vertices to the root. If there are more portals than marked vertices, we can just ignore the additional ones, as then it would be optimal to just connect each marked vertex to the root.

Subtask 1: Solve the example (10 points)

In this example, we need to add a portal from 6 to 0. For the second portal, we can either connect 3 to 0 or 5 to 0. Either way will lead to a sum of 6, meaning that the answer for this case is $\frac{6}{4} = \frac{3}{2} = \mathbf{1.5}$.

Subtask 2: Expensive Portals, Poor Government (20 points)

In this case, we can only add one portal. In order to do this, let's first calculate the value of $\sum_{v \in S} f(v)$ without a portal. Then, let's calculate for each vertex v how much we would save if we added a portal from v to the root. Let c_v be the number of inhabited planets in the subtree of v. Then adding a portal at position v would save us $c_v \cdot (d_v - 1)$, where d_v is the depth of vertex v. This is beacuse each vertex in the subtree of v can now take the portals and thus save $d_v - 1$ edges (taking the portal takes 1 edge). Thus we could calculate this value for each vertex v and take the vertex maximizing it (minimize the sum $\leftrightarrow$ maximize what we save by adding a single portal) We can calculate c_v and d_v for each vertex with a simple DFS. See the pseudocode below:

```
initial_sum = 0
max_save = 0

def dfs(curr, depth):
    # Saves the number of inhabited planets in the subtree of curr
    subtree_count = 0

    if inhabited[curr]:
        # Add the distance from curr to the root
        # to the initial sum
        initial_sum += depth

        # Add the current vertex to the current subtree count
        subtree_count += 1

    # Recurse
    for child in children[curr]:
        subtree_count += dfs(child, depth + 1)

    # Calculate how much can be saved by adding a portal here
    max_save = max(max_save, (depth - 1) * subtree_count)

    return subtree_count

def solve():
    inhabited_count = dfs(0, 0)
    return (initial_sum - max_save) / inhabited_count
```

DFS in a tree takes $O(n)$ time and $O(n)$ space.

Subtask 3: The Alliance (70 Points)

If we continue experimenting, it gets very clear that the solution is going to be some sort of dp. Thus, in classical dp fashion, let's first try to come up with any dp that works and then optimize it later.

In order for the dp recurrence to become more manageable, let's suppose that any edge incident to the root takes no time to cross. Note that this will also mean that newly added portals are free to cross. Note that if we know calculate $\sum_{v \in S} f(v)$, we will end up with the solution of the original problem minus $(|S| - 1)$. This will simplify the dp formula a lot.

Consider the following dp: $\text{dp}[v][i][j]$ is the minimal cost of the vertices in the subtree of v if we add i edges there and the next ancestor of v with an added edge is j edges above v. This dp has $O(n^2 k)$ states. Now let's look at the transitions:

$$\text{dp}[v][i][d] = \min \left\{ \text{dp}[v][i-1][0], \texttt{inhabited}[v] \cdot d + \left(\min_{i_0 + \cdots + i_{c_v - 1}} \sum_{l=0}^{c_v - 1} \text{dp}[u_{v,l}][i_l][d+1] \right) \right\}$$

where c_v is the number of children of vertex v, and $u_{v,0}, \ldots, u_{v,c_i - 1}$ are the children of v. Let's disect this transition. There are two possibilities for $\text{dp}[v][i][d]$: Either we add an edge to v, or we don't. If we do, we have one edge less available in the subtree, but there's an edge connected to v. Thus we need to take the value of $\mathrm{dp}[v][i-1][0]$. If we don't add an edge there, the next added edge above v will be d edges above v. Thus if v is inhabited, we need to add d to account for vertex v. Now, we need to combine the values of the children. For each child, the next added edge above it has distance $d+1$ to them (as the distance from v is d). Finally, we need to distribute the i additional edges to the subtrees of the children of i and take the best assignment of these edges. This is where the $\min_{i_0+\cdots+i_{c_v-1}}$ comes from.

Doing everything except for $\min\limits_{i_0+\cdots+i_{c_v-1}} \left\{ \sum\limits_{l=0}^{c_v-1} \mathrm{dp}[u_{v,l}][i_l][d+1]) \right\}$ is simple. Let's focus on calculating this for the moment. Let's look at $\mathrm{dp}[v][i]$ as a dp where the values are vectors, where we can easily merge two of these vectors in $O(n)$. We can calculate $\mathrm{dp}[v][i]$ more efficiently by calculating it for all i at the same time: Let

$$\mathrm{dp}_j[v][i] = \min_{i_0+\cdots+i_j} ((\mathrm{dp}[u_{v,0}][i_0] + \cdots + \mathrm{dp}[u_{v,j}][i_j])$$

Then the recurrence is simpler:

$$\mathrm{dp}_j[v][i] = \min_{l\in\{0,\ldots,i\}} \left\{ \mathrm{merge}(\mathrm{dp}_{j-1}[v][l], \mathrm{dp}[u_{v,j-1}][i-l]) \right\}$$

Doing this and calculating the bounds naively gives a solution that runs in $O(n^2k^2)$ time and uses $O(n^2k)$ memory. However, we can optimize this even further.

Let $f_{v,i} = \mathrm{dp}[v][i][x]$. Then, there are two things we can optimize: The number of merges we need to do, and the time needed to merge these functions. In our previous solution, the number of merges would be estimated as $O(nk^2)$ and the time for a merge as $O(n)$.

Recalculating the number of merges

Let's recalculate the number of merges. We will see that this procedure will actually do a lot less merges than we calculated. For simplicity, suppose that the tree is a binary tree (the bound we are calculating for the binary tree will still work for a normal tree, as we can transform a normal tree with n vertices to a binary tree with less than $2n$ vertices). Let s_v be the size of the subtree of vertex v. Then the at vertex v, the second dimension of the dp has size $\min(k, s_v)$, or in other words, we have $\min(k, s_v)$ of these $f_{v,i}$. To combine the two subtrees of the vertices a and b, we will now need $\min(k, s_a) \cdot \min(k, s_b)$ time.

Calculating a stronger bound

Suppose we need to store s_v elements instead of $\min(k, s_v)$, and thus combining costs $s_a \cdot s_b$. This will obviously be slower than what we're doing.

Claim: This strategy will merge some amount in $O(n^2)$ times.

Proof: When we combine the two subtrees and pay $s_a \cdot s_b$ in time, look at it as $(1+\cdots+1)\cdot(1+\cdots+1)$. As there are s_a ones in the first subtree, assign to each of these ones a node. Do the same thing with the second subtree. If we expand, each combination of a one on the left side and a one on the right side will contribute one to the final sum of $s_a \cdot s_b$. Thus the number of operations we do in one combination is precisely the number of pairings of a vertex from subtree a and a vertex of subtree b. Now consider any pair of two vertices in the tree, u and v. We will add the pairing of these two verties exactly once to the final cost, namely at the lca of u and v. Thus the sum of the costs is the number of pairings of vertices in the tree, which is in $O(n^2)$.

Remark: You can read about this trick in more detail here: `https://usaco.guide/adv/comb-sub`.

Calculating an even stronger bound

When we calculated the sum before, we ignored the min part, so clearly, the real bound must be a bit better.

Claim: This strategy will merge some amount in $O(nk)$ times.

Proof: Now, when pairing, we are limited to at most k vertices. Consider the preorder of the vertices. Now when merging, if $s_a > k$, only consider the last k vertices from the first subtree with respect to the preorder, and if $s_b > k$, only consider the first k vertices from the preorder. Now a pair of vertices is only a valid pairing if their distance in the preorder is at most k. So for each vertex, there are only $2k$ possible other vertices for a pairing, thus we will have $O(nk)$ pairings, and thus we will have that many merges.

Optimizing the time needed for a merge

In this subsection, let $\text{conf}(v, i, x)$ be the optimal configuration of the added edges in the subtree at v if we have i edges to add and the next edge above v is at distance x (like the definition of dp).

Lemma 1: Let z_v be the number of inhabited planets in the subtree of v. Then $f_{v,i}(1) - f_{v,i}(0) \le z_v$.

Proof: Consider $\text{conf}(v, i, 0)$ for the case of $x = 1$. Now, the cost of each inhabited vertex will be at most one bigger than its cost in $\text{conf}(v, i, 0)$, as if their path doesn't go to v, its cost stays the same, and otherwise, it increases at most by 1. Thus $f_{v,i}(1) \le f_{v,i}(0) + z_v$.

Lemma 2: $f_{v,i}(x + 1) - f_{v,i}(x) \ge 0$.

Proof: Consider $\text{conf}(v, i, x + 1)$ for the case $f_{v,i}(x)$. Clearly, the cost of no vertex increases, as for the paths that don't go through v nothing changes. The paths through v only get shorter. Thus $f_{v,i}(x + 1) \ge f_{v,i}(x)$.

Lemma 3: $f_{v,i}$ is concave.

Proof: Note that while calculating the formula above, there are only two things we do when merging $f_{v,i}$s. Let's prove by induction on $h_v + i$, where h_v is the distance from v to the furtest leaf in the subtree of v. When we do calculate $f_{v,i}$, there are only two things we're doing:

1. $f_{v,i}(x) \leftarrow \min(f_{v,i}(x), g(x))$, where $g(x)$ is also concave (when we either merge it with other $f_{u,j}$, which is concave by the induction hypothesis, or when we do $\text{dp}[v][i][x] \leftarrow \min(\text{dp}[v][i-1][0], \text{dp}[v][i-1][x])$, a constant is also concave).
2. $f_{v,i}(x) += x$ (when we do account for v begin inhabited)

In the first case, we need to prove that the minimum of two concave functions is still concave. Let $a(x) = \min(b(x), c(x))$, where $b(x)$ and $c(x)$ are concave. Consider the smallest x such that $a(x) - a(x-1) < a(x+1) - a(x)$. Clearly, as both functions are concave, not all three of $a(x + 1), a(x)$ and $a(x - 1)$ can come from the same function. Without loss of generality, $b(x) = a(x)$. Suppose $a(x - 1) = c(x - 1)$. Then clearly $b(x) - b(x - 1) \le a(x) - a(x - 1)$, and thus $a(x+1) - a(x) \le b(x+1) - b(x) \le b(x) - b(x-1) \le a(x) - a(x-1)$, a contradiction. Thus suppose $a(x - 1) = b(x - 1)$. Then $a(x + 1) - a(x) \le b(x + 1) - b(x) \le b(x) - b(x - 1) \le a(x) - a(x - 1)$. Another contradiction. Thus a is concave. So the first case holds.

For the second case, Note that adding x to $f(x)$ will just add 1 to every $f(x + 1) - f(x)$. Thus $f(x) - f(x-1) \ge f(x+1) - f(x) \Leftrightarrow f(x) - f(x-1) + 1 \ge f(x+1) - f(x)$.

Lemma 4: $f_{v,i}$ is a piecewise linear function, consisting of at most z_v linear segments.

Proof: By lemma 1, the first slope of the function is at most z_v. As the function is concave, we get $z_v \geq \cdots \geq f(x+1) - f(x) \geq \cdots \geq 0$. Thus first there is some segment with slope at most z_v, then some segment with a smaller slope, then again, until we arrive at slope 0.

Thus instead of the whole function, we could only save the points where the slope changes.

Lemma 5: Let r be the root. Then $f_{r,k-1}(0) - f_{r,k}(0) \leq c \cdot \frac{n^2}{k^2}$, where c is a constant. (Note that $f_{r,k}(0)$ is the total cost of an optimal configuration with k portals).

Proof: Consider conf$(r, k, 0)$ and look at the vertices with an additional edge. For each of these vertices, mark it if there are at most $\frac{2n}{k}$ vertices with a shortest path going through that edge. Note that if there are less than $\frac{k}{2}$ marked vertices, then the total number of vertices whose shortest path takes a portal, is bigger than $\frac{2n}{k}(k - \frac{k}{2}) = n$, which is a contradiction. Thus the number of marked vertices is at least $\frac{k}{2}$. For each marked vertex, consider it and its $\frac{2n}{k}$ ancestors. If for any vertex the root is among is $\frac{2n}{k}$ ancestors, then we can remove the additional edge from this vertex while increasing the cost by at most $\frac{4n^2}{k^2}$. Otherwise, we are considering $(\frac{2n}{k}+1)\frac{k}{2} > n$ vertices. Thus by the pigenhole principle, one vertex v must be considered at least twice by two marked vertices u and w. Removing the portals at u and w will increase the cost by at most $\frac{8n^2}{k^2}$. Either way, we get a configuration for $k-1$ with which costs at most $c\frac{n^2}{k^2}$ more, thus $f_{r,k-1}(0) - f_{r,k}(0) \leq c\frac{n^2}{k^2}$.

Lemma 6: Let take$_{v,i}(x)$ be true if there is an optimal configuration where we add i edges in the subtree of v and the next added edge above v has distance x to v. Then there is a constant c such that $f_{v,i}(x) - f_{v,i}(0) \geq c\frac{n^2}{k^2}$ implies that take$_{v,i}(x)$ is false.

Proof: Let c be the constant from Lemma 5. Then consider an optimal configuration where take$_{v,i}(x)$ is true and $f_{v,i}(x) - f_{v,i}(0) \geq c\frac{n^2}{k^2}$. Then $f_{r,k}(0)$ be the cost of this configuration. Now, place an additional portal at v. We get a new configuration with $k+1$ portals, which costs more than $c\frac{n^2}{k^2}$ less. Thus, $f_{r,k}(0) - f_{r,k+1}(0) > c\frac{n^2}{k^2} > c\frac{n^2}{(k+1)^2}$, which is a direct contradiction to Lemma 5.

Lemma 7: We only need to store an amount in $O(\frac{n}{k})$ of linear segments for $f_{n,i}(x)$.

Proof: The amount needed is at most $\sqrt{2c}\frac{n}{k} + 1 = m$, where c is the constant from Lemma 6. Consider the first m linear segments. Let x be the start of the next segment. Then $f_{v,i}(x) = f_{v,i}(0) + l_0 a_0 + \cdots + l_{m-1} a_{m-1}$, where the ls are the lengths of the segments and the as are the slopes of the segments. As $l_0, \ldots, l_{m-1} > 0$ and $a_{m-1} > a_{m-1} > \cdots > a_0 \geq 0$, we get that $f_{v,i}(x) \geq f_{v,i}(0) + 0 + 1 + \cdots + (m-1) = f_{v,i}(0) + \frac{m(m-1)}{2} = f_{v,i}(0) + \frac{(\sqrt{2c}\frac{n}{k}+1)\sqrt{2c}\frac{n}{k}}{2} > c\frac{n^2}{k^2}$. Thus $f_{v,i}(x) - f_{v,i}(0) \geq c\frac{n^2}{k^2}$. By Lemma 6, these values are irrelevant, as take$_{v,i}(x')$ for $x' \geq x$ will never be true.

Putting it together

We have n vertices, for each of the vertices, we need to store k piecewise linear functions, with at most $c \cdot \frac{n}{k}$ different segments for each of these functions. Thus this needs $c \cdot n \cdot k \cdot \frac{n}{k} \in O(n^2)$ memory.

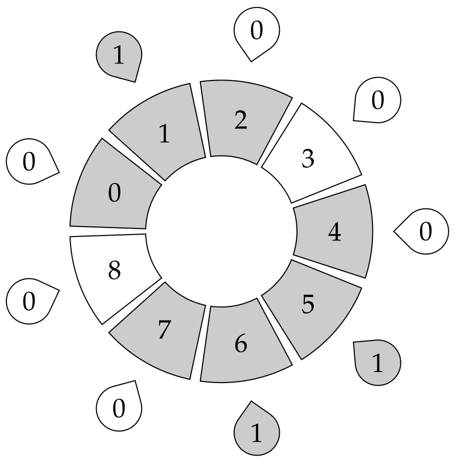
Additionally, we do $O(nk)$ merges. We can perform each merge in time linear in the number of segments in the functions we are merging. Thus one merge takes $O(\frac{n}{k})$, so the total runtime is again in $O(n^2)$.

Particle detector

Task Idea	Bibin Muttappillil
Task Preparation	Jan Schär
Description English	Jan Schär
Description German	Christopher Burckhardt
Description French	Florian Gatignon
Solution	Jan Schär

Subtask 1: A toy detector (15 points)

There can be at most 7 broken sensors. Only one such configuration exists; it is shown here, with broken sensors shaded:



Subtask 2: All sensors report 1 (10 points)

The answer is N – all sensors can be broken.

This is easy to see, since in this case, every sensor is broken and both its neighbors are broken too, which means that every sensor reports 1.

Subtask 3: All sensors report 0 (15 points)

If N is divisible by 3, the answer is $\frac{2N}{3}$, otherwise 0.

Each sensor reports 0, which means that either the sensor is good and both neighbors are good, or the sensor is good and both neighbors are bad, or the sensor is bad and exactly one of the neighbors is bad.

Let's do a case distinction:

N is divisible by 3 It is possible that $\frac{2N}{3}$ sensors are broken. The broken sensors are 0, 1, 3, 4, 6, 7, ..., $N-3$, $N-2$. This works, because every broken sensor has one broken neighbor, and every good sensor has two broken neighbors, which means that all sensors report 0.

If more than $\frac{2N}{3}$ sensors are broken, then there must be at least one sensor which is broken and has two broken neighbors, and then that sensor would report 1.

Thus, $\frac{2N}{3}$ is the maximum number of broken sensors.

N is not divisible by 3 There can't be any broken sensors. Let's assume towards a contradiction that there exists at least one broken sensor. Then one of the neighbors of this sensor must be broken too, otherwise it would report 1. Let's rotate the detector such that these two adjacent sensors are numbered 0 and 1. As sensor 1 is broken, it must have exactly one broken neighbor, hence sensor 2 must be good. Because sensor 2 is good and its first neighbor is broken, its other neighbor 3 must be broken too. We can continue this reasoning, and find that all sensors i with $i \bmod 3 = 0$ and $i \bmod 3 = 1$ must be broken, while all sensors with $i \bmod 3 = 2$ must be good.

We know that N is not divisible by 3, i.e. $N \bmod 3 \neq 0$, and hence $(N-1) \bmod 3 \neq 2$. This means that sensor $N-1$ is broken.

So sensor 0 is broken and both its neighbors, 1 and $N-1$ are broken, which means that sensor 0 reports 1 – we have reached a contradiction.

Subtask 4: General case (60 points)

The main insight is that once the brokenness of two neighboring sensors is known, the brokenness of all other sensors can be inferred. If we pick two neighboring sensors and try out each of the four possible combinations of their brokenness, and then infer the rest, we have covered all possible configurations.

The algorithm

Try out each configuration for sensors 0 and 1: good/good, good/broken, broken/good, and broken/broken. For each of these, iteratively infer the brokenness of sensors $2, 3, \ldots, N-1$. If sensor $i-1$ reports 1 and none or both of sensors $i-2$ and $i-1$ are broken, then sensor i is broken. If sensor $i-1$ reports 0 and exactly one of sensors $i-2$ and $i-1$ is broken, then sensor i is broken too. In all other cases, sensor i is good. At the end, we check if the reports from sensors $N-1$ and 0 are as expected, otherwise this configuration for sensors 0 and 1 is not possible.

For each of the possible configurations, we count the number of broken sensors, and then output the maximum of those. If no possible configuration was found, output -1.

Here is the solution in pseudo-code:

Algorithm: Solution for the general case

```
def Report($a, b, c$):
    ▹ From the brokenness of three adjacent sensors, compute the report of the middle one
    if ($b = 0$ and $a \neq c$) or ($b = 1$ and $a = c$) then
        return 1
    else
        return 0
answer ← $-1$
foreach $(b_0, b_1) \leftarrow \{(0,0),(0,1),(1,0),(1,1)\}$ do
    count ← $b_0 + b_1$
    $t_0, t_1 \leftarrow b_0, b_1$
    for $i \leftarrow 2$ to $N - 1$ do
        ▹ Infer the brokenness of sensor $i$
        if Report($t_0, t_1, 0$) $= r_{i-1}$ then
            $t_2 \leftarrow 0$
        else
            $t_2 \leftarrow 1$
        count ← count + $t_2$
        $t_0, t_1 \leftarrow t_1, t_2$
    if Report($t_0, t_1, b_0$) $= r_{N-1}$ and Report($t_1, b_0, b_1$) $= r_0$ then
        answer ← max(answer, count)
return answer
```

Correctness

Let's take any possible configuration of broken sensors. Look at the brokenness of sensors 0 and 1 in this configuration. One of the four iterations of the solution has the same configuration for these two sensors. All the other sensors will be inferred with the same brokenness as well, since there is only one way to infer their brokenness.

Thus, the solution exhaustively checks all possible configurations. This means that in particular, it checks the configuration with the maximum number of broken sensors, as long as a valid configuration exists.

Since reports of sensors $1, 2, \ldots, N-2$ are used for the inferrence, they must necessarily be consistent, and we check if the reports from sensors 0 and $N-1$ are consistent. This means that the solution skips all invalid configurations.

Thus, the solution is correct.

Time and memory

The algorithm iterates over the input 4 times, so its running time is $O(n)$.

It is not necessary to store all the inferred brokenness states in an array, it suffices to keep the last two values and a count of broken sensors, so the algorithm can be implemented with $O(1)$ memory usage.