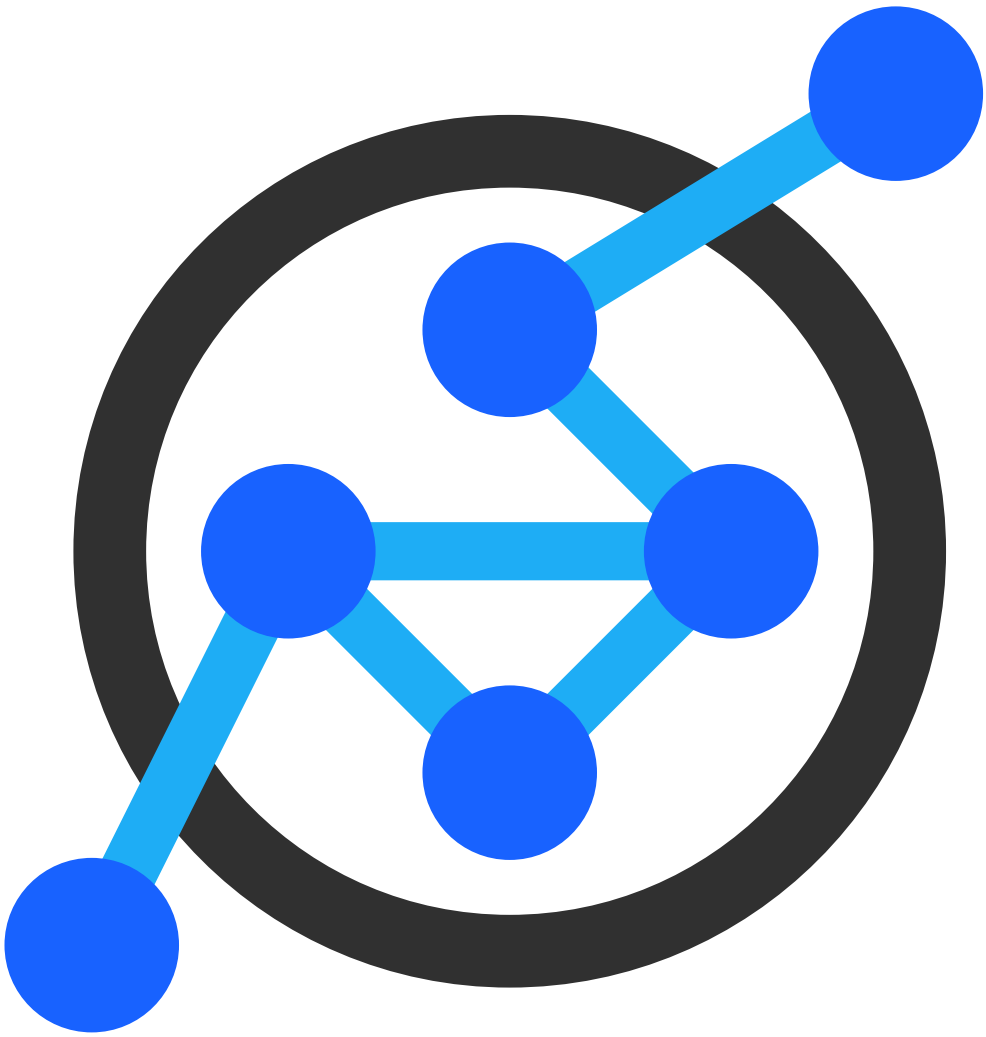
Second Round Theoretical

Solutions



Swiss Olympiad in Informatics

March 6–7, 2021

Lasercut

Task Idea	Daniel Rutschmann
Task Preparation	Christopher Burckhardt
Description English	Christopher Burckhardt
Description German	Jan Schär
Description French	Florian Gatignon
Solution	Christopher Burckhardt

Subtask 1: (10 Points)

The minimal number of cuts needed is 5.

Subtask 2: (35 Points)

Two main observations were needed to solve this subtask.
Lemma 1: The minimum number of cuts needed for a larger piece is never less than for a smaller piece.
Lemma 2: After a cut, the amount of cuts still needed, is equal to the single piece that needs maximal cuts.
Proof of Lemma 1: Assume there exists a larger piece, size $1 \times m + 1$, which needs less cuts than a smaller piece, size $1 \times m$. We simply use the same sequence of cuts and reorderings used for the larger piece on the smaller piece, while disregarding any reorderings of the single missing square. The minimal number of cuts needed for the smaller piece is now equal to that of the larger piece, which is a contradiction and concludes the proof.
Proof of Lemma 2: Since in every cut we may split arbitrarily many pieces. Using the maximum number of cuts needed for a single piece, will also be enough to fully split any other piece.

Using these two observations we see that after any cut the largest piece should be minimized. This is achieved by cutting a $1 \times m$ in half, the largest piece will then be $1 \times \lceil \frac{m}{2} \rceil$. This can immediately be implemented as a recusrsive algorithm.

```
// recursive implementation
int C(int m){              // Calculates number of cuts needed for a 1 x m bar
    if(m == 1){
        return 0;          // Base Case 1 x 1 bar needs 0 cuts.
    }
    return 1 + C((m+1)/2); // Cut the current bar in half, add the cuts needed for the larger half.
}
```

Since for every function call m is halved, it will be called $O(\log m)$ times. Each function call takes $O(1)$ time. This results in a runtime of $O(\log m)$. By rewriting the function to be iterative we can optimize the space usage to $O(1)$. In fact the compiler will do this automatically.

```
// iterative implementation
int C(int m){
    int ret = 0;
    while(m != 1){
        ret++;
        m = (m+1)/2;
    }
    return ret;
}
```

Additionally the recurrence may be solved which yields the closed form solution of $\lceil \log_2 m \rceil$. However the closed form solution can also only be evaluated in $O(\log m)$ time [1] and $O(1)$ space.

[1] No points were deducted for assuming that $\lceil \log_2 m \rceil$ may be evaluated in $O(1)$

Subtask 3: (55 Points)

To extend the previous solution to an $n \times m$ bar we make the observation that both dimensions can be treated independently. This is because in a single cut on a single piece we can only halve either the number of rows or the number of columns. And due to lemma 2. above, we reduced the problem of finding the number of cuts for a set of pieces to finding the maximum number of cuts for a single piece.
Since the dimensions are independant we can first split the $n \times m$ bar into n bars of size $1 \times m$ and then split each of these bars into single pieces. For both parts we use the same splitting process as in subtask 2. The closed form solution is thus $\lceil \log_2 n \rceil + \lceil \log_2 m \rceil$. Calculating the answer with either the closed form solution or the recurrence relation has a runtime of $O(\log n + \log m)$ and uses $O(1)$ space.

Planeseating

Task Idea	Johannes Kapfhammer
Task Preparation	Monika Steinová
Description English	Monika Steinová
Description German	Tobias Feigenwinter
Description French	Florian Gatignon
Solution	Bibin Muttappillil

If we model the mice as nodes and their likeness as edges then the task boils down to checking if a bidirectional graph is a grid graph and if so relabeling the nodes to form a grid.

There are two conditions holding if and only if for feasible seating arrangements (given from the task):

1. We have a perfect matching between mice and seats.
   a) Every seat has a mouse.
   b) Every mouse has a seat.
2. A pair of mice are neighbors $\iff$ the pair likes each other (combination of the last two bullet points).
   a) Every pair of mice liking each other sits next to each other.
   b) No pair mice not liking each other sits next to each other.

For analysis, we assume that $\texttt{num(m)} \leq 4$ to ensure that `likes(m)` runs in constant time and space. We can assume that as we could simply check every mouse and see if any mouse fulfills $\texttt{num(m)} > 4$, in $O(R \cdot C)$ time and constant space, and this wouldn't be a possible grid graph.

Also, it is not explicitly excluded that a mouse can like itself, but such a setting can't be a grid graph (as you can't be your own neighbor with (1). This can also be simply avoided by checking every mouse on $m \in \texttt{likes}(m)$, in $O(R \cdot C)$ time and constant space (follows from above).

Subtask 1: Concrete example (20 Points)

Here we are given a list of the like pairs. The input according to the description should be:

- $\texttt{likes}(1) = \{2, 3, 8\}$
- $\texttt{likes}(2) = \{1, 4, 6, 9\}$
- $\texttt{likes}(3) = \{1, 4\}$
- $\texttt{likes}(4) = \{2, 3, 5\}$
- $\texttt{likes}(5) = \{4, 6\}$
- $\texttt{likes}(6) = \{2, 5, 7\}$
- $\texttt{likes}(7) = \{6, 9\}$
- $\texttt{likes}(8) = \{1, 9\}$
- $\texttt{likes}(9) = \{2, 7, 8\}$

One possible solution is:

Every other solution is just a rotation or reflection of the above.

3	1	8
4	2	9
5	6	7

Subtask 2: A plane with a single seat in a row (30 points)

This subtask boils down to see if the graph is a simple path.

Idea

We first find a corner, by checking every mouse and see if one of them has degree 1. There should exist exactly two, one for the front seat and one for the back, otherwise it is impossible. Then taking an arbitrary one of them we set it on the first row and follow the liked mouse. After that we set the mouse to the next seat and follow the liked mouse from which we didn't just come from. As every *internal* mouse should have exactly two neighbors (and therefore two in their `likes` set) and from one of them we came from the next one should be unique, otherwise it is impossible. So we repeat this process until we've reached the last row. [1]

Pseudocode

```
# assume that impossible() terminates the program

# helper function to check and return the unique element from a set
def singleton(s):
    if size of s == 1: return next(iter(s))  # return the single element from s
    impossible()

def find_corner():
    for m from 1 to R:
        if num(m) == 1: return m
    impossible()

last = -1
cur = find_corner()

for r from 1 to R:
    assign(cur, r, 1)
    last, cur = cur, singleton(likes(cur) - {last})
```

Analysis

- Finding a corner is a simple iteration with a constant check: $O(R)$ runtime and constant memory.
- In the main loop everything has constant runtime and space, especially `likes` as we checked it beforehand: $O(R)$ runtime and constant memory.

So in total we have $O(R)$ runtime and constant memory.

Subtask 3: General case (50 points)

Idea & Pseudocode

First *wlog* we assume that $C \leq R$. We can do that as rows and columns are symmetrical, except for the `assign` function where we can simply swap the arguments if $C > R$.

```
# instead of assign(m, r, c)
if C > R: assign(m, c, r)
else:     assign(m, r, c)
```

[1] Another idea would be to do some kind of graph traversal (dfs, bfs) and checking for loops and that everything is visited. This approach though needs more memory, as it doesn't use the fact that we check for a simple path.

We try to find a corner by checking every mouse on their degree and returning one that has degree 2 (resp. 1 like if $C = 1$).

```
def find_corner():
    for m from 1 to R * C:
        if (C == 1 && num(m) == 1) || (C >= 2 && num(m) == 2): return m
    impossible()

corner = find_corner()
```

Then we walk along the border in both directions until we hit the first corner (for $C = 1$ we skip the border step). We can do that buy following the mice with degree 3 which wasn't the one we've come from.

```
# helper function to find next node in border
def next_border(last, cur):
    # only works if cur and the next ones are internal border nodes
    # -> require C >= 4
    return singleton(likes(cur) - {last} where num == 3)

# follow both borders simultaneously
let border1 = [corner, one of likes(corner)]
let border2 = [corner, other one of likes(corner)]
do C-3 times:  # number of edges in-between internal border nodes
    border1 += next_border(last two of border1)
    border2 += next_border(last two of border2)
```

We take the (resp. any) full border with their corners of length C and set it to the first row. The first row will be our current row and also initialize the previous row to sentinels.

```
def singleton_opt(s):
    if size of s == 1: return single element from s
    return -1

# helper function to return corner if available
def potential_corner(last, cur):
    # only works if there is a second corner
    # -> require C >= 2
    return singleton_opt(likes(cur) - {last} where num == 2)

potential = potential_corner(last two of border1)
cur = border1 + potential
if potential == -1:  # try the other border
    potential = potential_corner(last two of border2)
    cur = border2 + potential
if potential == -1:
    impossible()

prv = [-1] * (C + 2)
nxt = [-1] * (C + 2)
```

Then we go over every mouse in the current row, assign them a seat, and add their remaining mouse that is not in the current or previous row to the next row. During the process we also check that the neighbors coincide with the `likes` set, and we also check if the start corner gets assigned a second seat.

```
for r from 1 to R:
    for c from 1 to C:
        assign(cur[c], r, c)
        nxt[c] = singleton(likes(cur[c]) - {cur[c-1], cur[c+1], prv[c]})

        # likes check
        if likes(cur[c]) != ({prv[c], cur[c-1], cur[c+1], nxt[c]} - {-1}):
            impossible()

        # loop check (necessary?)
        if mid[c] == corner:
            impossible()

    prv, cur, nxt = cur, nxt, [-1] * (C + 2)
```

If there is no suitable corner, unique node to progress the border or the next row, or a check fails then we call `impossible()`.

Correctness Sketch

We give a correct seat assignment for grid graphs: If we have a grid graph as an input we can rotate and flip it such that our chosen corner is top left and our chosen border is on top (this obviously preserves the grid property). Then our algorithm will assign the seats corresponding to this orientation. This is the case as we start on the top left (as we oriented it that way). Then we walk the borders and this works as the next border node different from the last will be unique in the graph. Our chosen border must end in a corner as we walked on $C-2$ internal border nodes and the last one has only the corner remaining. And in the last step we always choose the neighbor which hasn't been visited yet on each node (as we explicitly exclude all other neighbors). The next row will get the correct coordinates relative to our current row, and with the fact that our first row is correct we see that every node gets a correct assignment.

We dismiss non grid graphs: This is the same as saying that every graph that passes is a grid graph. From our *likes check* we can directly follow (2) as our seat neighbors have to match the liked mice to not call `impossible()`. From the fact that we iterate over all pairs (r, c) and call `assign(m, r, c)` exactly once every iteration we can follow (1a). The only thing left is (1b): This is only possible if one mouse has two seats (otherwise we would have more seats than mice). If we have such a mouse m we can show that there exists a mouse with smaller seat coordinate (ordered lexicographically). If we look at the seat with the smaller coordinate and look at either the neighbor top the top or to the left we can see that they will also have a second seat in the neighboring area of m. This has to be the case as it supposedly passed our *likes check* so both seats of m have the same neighbors. We can repeatedly use this argument. The only way to stop is to be at the start corner (as it is the only node without a top or left neighbor). Thus, if we have a mouse with two seats then the start corner mouse also has two seats, but then our *loop check* would fail, so this can't happen. [2] Thus, the input need to fulfill all the four conditions and is therefore a grid graph. □

Analysis

- Finding a corner is a simple loop check, without any containers: $O(R \cdot C)$ runtime and constant space.
- `likes` is bounded by 4, so all the checking and returning is in constant time and space.
- As the functions inside the border walking loop take constant time and space (see above) we will have a runtime and space of $O(C)$.

[2] I think that the loop check is actually not necessary, but I didn't find a nice proof for that. So for simplicity of the proof we rather just check it.

  - Note that we only add one element in each iteration and we can extend the list in (amortized) constant time.
- Checking and adding the corner also takes the same runtime and space as for one iteration as for the reason above.
- Setting up `prv` and `nxt` take $O(C$ time and space.
- The last part with the nested loops has $O(R \cdot C)$ runtime and $O(C)$ space:
  - everything in the loop has constant runtime as we only look at constant amount of memory in every iteration, especially the *likes check*
  - the only non-constant bounded containers are `prv`, `cur` and `nxt` each of which is bounded by $O(C)$
- We assumed $C \leq R$, which actually means that we take the minimum of these two instead of just C.

Putting it all together we have $O(R \cdot C)$ runtime and $O(\min(C, R))$ space.

Mug stacking

Task Idea	Daniel Rutschmann
Task Preparation	Tobias Feigenwinter
Description English	Tobias Feigenwinter
Description German	Jan Schär
Description French	Florian Gatignon
Solution	Tobias Feigenwinter

Subtask 1: Solve an example (10 points)

The minimal time possible is seven seconds. The following is a possible way of achieving that time:

- Place the first mug on the second mug (1 second)
- Place the third mug on the fourth mug (1 second)
- Place the fifth mug on the sixth mug (1 second)
- Place the ninth mug on the eighth mug (1 second)
- Place the tenth mug on the seventh mug (3 seconds)

Subtask 2: Perfect pairing (40 points)

Because exactly half of the mugs have a curved base, we always have to stack a curved mug on top of a flat mug. One optimal solution is to place the i-th curved mug on top of the i-th flat mug. We can prove this using the following inductive construction.

We start by showing that the first flat mug can always be paired with the first round mug. One of these two cups must be the first mug overall and is therefore at position 0. Let a be the position of the first mug of the opposite type, i.e. the mug that our proposed solution would pair with the first mug. Now assume that our solution is not optimal, and an optimal solution would instead pair the cups at positions 0 and i as well as the cups at positions a and j. Since the cups at positions 0 and a are the leftmost ones of their respective types, we know that $j > 0$ and $i > a$. We will now distinguish the two cases of $j < a$ and $j > a$.

Case $j < a$: We also know that $j > 0$ and $i > a$, so the order of the four mug positions is $0 < j < a < i$. The optimal solution uses $|i - 0| + |a - j| = i + a - j$ seconds to stack the four mugs. Our proposed solution also uses $|a - 0| + |i - j| = i + a - j$ seconds. Therefore, we can pair the mugs at positions 0 and a and the mugs at position i and j and still be optimal.

Case $j > a$: We still know that $j > 0$ and $i > a$, so the order of the four mug positions is either $0 < a < j < i$ or $0 < a < i < j$. The (supposedly) optimal solution uses $|i - 0| + |a - j| = i + j - a$ seconds to stack the four mugs. Our proposed solution uses $|a - 0| + |i - j|$ seconds. If $i > j$, this is equal to $a + i - j$, otherwise it's equal to $a + j - i$. In both cases, this is less, so such a configuration cannot exist in an optimal solution.

This concludes the proof that we can always stack the first round mug on the first flat mug. This also shows our proposed solution is optimal for the entire problem if we remove the first round and the first flat mug (e.g. the mugs that we just stacked) and inductively repeating the above proof on the modified problem until no mugs remain. The new problems slightly differs from the original one in that there might be gaps between some of the mugs, but this proof still holds because no argument assume that there are no gaps. In conclusion, we have shown that it is optimal to stack the i-th curved mug on top of the i-th flat mug.

The following code shows a simple scanline approach to calculate the time of this optimal solution in linear time and constant memory:

```
int flat = -1, curved = -1;
int result = 0;
for (int i = 0; i < n; i++) { //there are 2n mugs
    // find the position of the next flat mug
    do {
        flat++;
    } while (type(flat) != "flat");

    // find the position of the next curved mug
    do {
        curved++;
    } while (type(curved) != "curved");

    result += abs(curved - flat);
}
return result;
```

Subtask 3: Stofl's cupboard (50 points)

Contrary to the last subtask, now some curved mugs are at the bottom of a stack. There is a Dynamic Programming approach that works on prefixes of the solution. A prefix of the solution consists of a prefix of the mugs and all stacks our solution makes. First, we'll show an useful property of such prefixes.

We call some mug "unmatched" for some prefix if the mug it is stacked with is not within that prefix. We claim that a prefix may contain either unmatched flat or unmatched curved mugs, but not both. This can be proven in almost the same way we've previously proven that it is optimal to stack the first curved mug on the first flat mug in the second subtask.

Using this property, we can now define the DP table `DP[p][u]`, where p is the size of the observed prefix and there are $|u|$ unmatched mugs. If $u < 0$, the unmatched mugs are flat, otherwise they are curved. The value of the DP is the cost of the prefix of the solution, but this also includes the cost of moving $|u|$ mugs from/to the unmatched positions to/from the end of the prefix. The basecases are `DP[0][0] = 0` (it does not take any time to do nothing) and `DP[0][u]` $= \infty$ when $u \neq 0$ (It is not possible to have unmatched mugs on an empty prefix).

At any other state `DP[p][u]`, we have two options:

- We do not match any new mugs. This is only an option if the type of the current mug matches the sign of u. In this case, `DP[p][u]=DP[p-1][u-1] + (u-1)` if the current mug has a curved base and $u > 0$, and `DP[p][u] = DP[p-1][u+1] + (-u-1)` if the current mug has a flat base and $u < 0$.
- We match a new pair of mugs. Again, the types of mugs have to match. In particular, if the current mug has a flat base, then the unmatched mugs must have a curved base. The possible DP values are `DP[p][u] = DP[p-1][u+1] + (u+1)` if $u >= 0$, and `DP[p][u] = DP[p-1][u-1] + (-u+1)` if $u <= 0$ and the current mug has a curved base.

When the conditions for multiple values of $DP[p][u]$ are fulfilled, we take the minimum, and if none are valid, we return infinity. We also return infinity when some request to the DP table goes out of bounds. After calculating all table entries, the end result is the value of `DP[2n][0]` (The prefix of size 2n is the whole solution, and we do not want any unmatched mugs in the end).

We have to calculate $O(n^2)$ table entries in constant time each, so the total runtime is $O(n^2)$. A trivial implementation also uses $O(n^2)$ memory, but this can be reduced to $O(n)$ by only storing the columns for the current and previous prefix length.

Astrophysics

Task Idea	Johannes Kapfhammer
Task Preparation	Timon Gehr
Description English	Timon Gehr
Description German	Christopher Burckhardt
Description French	Anonymous
Solution	Timon Gehr

Subtask 1: A very small galaxy (10 points)

In the example, the black hole consists of locations $\{3, 5\}$ and the white hole is empty: $\{\}$.

Subtask 2: A small black hole (15 points)

We use the following algorithm:

```
a = 0
for b = 1, ..., n−1:
  if simply_reachable(a, b):
    a = b
return a
```

If the black hole is at location 0, this is correct because there is no $b \in \{1, \ldots, n-1\}$ such that there is an edge $(0, b)$, and the algorithm will correctly return 0. Otherwise, at some point in the loop, b will be the location of the black hole. At this point, a will be updated to b, because no matter the value of a, there is an edge $a \to b$. After this point, a cannot be updated again because there are no edges that lead out of the black hole. Therefore, this algorithm computes the location of the black hole using exactly $n - 1$ measurements.

Running time is $O(n)$ and we use $O(1)$ memory.

Subtask 3: Small black hole and white hole (20 points)

Given the above, there is an easy solution using $2 \cdot n - 2$ measurements: Just run the algorithm above twice, once using measurements with the two arguments swapped.

However, we can improve on this: First, note that if we find some set B of locations such that we know that the black hole is in it, we can find the black hole using $|B| - 1$ measurements using the solution from subtask 1 on B. This is true because as long as the single black hole is in B, all other members of the set have at least one edge leading out of them (into the black hole). The analogous observations are true for the white hole.

Then, note that if there is an edge $a \to b$, this means a is not the black hole and b is not the white hole. Otherwise, if there is no edge $a \to b$, then a is not the white hole and b is not the black hole.

Therefore, we will group the locations into pairs of two. Using exactly one measurement per pair, we can create a set of candidates for the black hole and a set of candidates for the white hole consisting of elements that cannot be excluded using the above observation, both of size $n/2 + O(1)$ (if there is an odd number of locations, we will put one of the locations into both sets of candidates). This uses $n/2 + O(1)$ measurements.

After this step, we are left with two sets of size $n/2 + O(1)$ and we have to find exactly one hole in both of them. We can therefore compute the locations of the two holes using the solution from subtask 1 using additional $n/2 + O(1)$ measurements for each hole.

In total, we use $\frac{3}{2}n + O(1)$ measurements.

Running time is $O(n)$ and we use $O(n)$ memory. (We can optimize memory to $O(1)$ by doing comparisons of candidates on the fly as they would be added to the respective sets instead of storing the full sets.)

Subtask 4: A large black hole (25 points)

We can first use the solution from subtask 1 to find some location a that is inside the black hole. This works because we are guaranteed to enter the black hole at some point, and after that, we will never leave it. This uses $n-1$ measurements. We can then find the set of locations b that are inside the black hole by finding all edges (a, b), using another $n-1$ measurements.

However, this approach is again not optimal. We will modify the algorithm from subtask 1 such that it stores the sequence s of visited locations:

```
a = 0
s = [0]
for b = 1, ..., n-1:
  if simply_reachable(a, b):
    a = b
    s.append(a)
```

Now note that the black hole is always the set of locations of some suffix of sequence s. Some visited location will be the first one we see that is inside the black hole. After that, we will only move to other locations inside the black hole, and we will see each of them exactly once.

We can find this suffix using binary search:

```
(l, r) = (-1, len(s-1))
while l+1 != r:
  m = ⌊(l+r)/2⌋
  if simply_reachable(a, s[m]):
    r = m
  else:
    l = m
return s[r], ..., s[len(s-1)]
```

The loop invariant is that s[r] is inside the black hole and s[l] is outside the black hole, where we use $l = -1$ as a sentinel value allowing for the case that all values are inside the black hole. After the loop terminates, r will point to the first index into s such that $s[r]$ is inside the black hole.

In this way, we can find all locations inside the black hole using only $n + O(\log n)$ measurements.

Running time is $O(n)$ and we use $O(n)$ memory.

Subtask 5: Large black hole and white hole (30 points)

Running the inefficient solution from the previous subtask twice, we get a solution that uses $4n - 4$ measurements.

We can use the solution from subtask 2 to find a location in the black hole and a location in the white hole. This is correct because if both measured locations of a pair are inside the same type of hole, it suffices to keep one candidate. This takes $\frac{3}{2}n$ measurements. We can then find the remaining locations in each hole using $n + O(1)$ measurements each by exhaustively measuring reachability from the found locations from respectively to all other locations. This results in a total of $\frac{7}{2}n + O(1)$ measurements.

There is a solution in $3n + O(1)$ measurements: build sequences of visited locations for both the algorithm from subtask 1 and its analogue for white holes, then find the black hole and the white hole using linear search for the respective suffix. The linear searches will take about as many measurements as the size of the respective hole, therefore this step uses only around n measurements in total.

We can also just use the algorithm using binary search from above twice. This uses $2n + O(\log n)$ measurements.

Running time is $O(n)$ and we use $O(n)$ memory.