# CTU Open 2022

## Presentation of solutions

November 5, 2022

# Journals

# Journals

- Rewrite as sequence of journal types (up or down).
- `UUUUDDDDUUUDDDDUUUUDDDDUU`
- In one operation:
  - Reverse a substring
  - Swap types of journals
- `UUUUDDDDUUUDDDDUUUUDDDDUU`

# Journals

- Rewrite as sequence of journal types (up or down).
- `UUUUDDDDUUUDDDDUUUUDDDDUU`
- In one operation:
  - Reverse a substring
  - Swap types of journals
- `UUUUDDDDUUUDDDDUUUUDDDDUU`
- `UUU | UDDDDUUU | DDDDUUUUDDDDUU`

# Journals

- Rewrite as sequence of journal types (up or down).
- UUUUDDDDUUUDDDDUUUUDDDDUU
- In one operation:
  - Reverse a substring
  - Swap types of journals
- UUUUDDDDUUUDDDDUUUUDDDDUU
- UUU | UDDDDUUU | DDDDUUUUDDDDUU
- UUU | UUUDDDDU | DDDDUUUUDDDDUU

# Journals

- Rewrite as sequence of journal types (up or down).
- `UUUUDDDDUUUDDDDUUUUDDDDUU`
- In one operation:
  - Reverse a substring
  - Swap types of journals
- `UUUUDDDDUUUDDDDUUUUDDDDUU`
- `UUU | UDDDDUUU | DDDDUUUUDDDDUU`
- `UUU | UUUDDDDU | DDDDUUUUDDDDUU`
- `UUU | DDDUUUUD | DDDDUUUUDDDDUU`

# Journals

- Rewrite as sequence of journal types (up or down).
- UUUUDDDDUUUDDDDUUUUDDDDUU
- In one operation:
  - Reverse a substring
  - Swap types of journals
- UUUUDDDDUUUDDDDUUUUDDDDUU
- UUU | UDDDDUUU | DDDDUUUUDDDDUU
- UUU | UUUDDDDU | DDDDUUUUDDDDUU
- UUU | DDDUUUUD | DDDDUUUUDDDDUU
- UUUDDDUUUUDDDDDUUUUDDDDUU

# Journals

- Note the splits between blocks of same types. Need to remove all of them.
- `UUUU | DDDD | UUU | DDDD | UUUU | DDDD | UU`

# Journals

- Note the splits between blocks of same types. Need to remove all of them.
- `UUUU | DDDD | UUU | DDDD | UUUU | DDDD | UU`
- Can not remove more than two splits
  - Any split with its both elements outside the operation or inside the operation remains a split

# Journals

- Note the splits between blocks of same types. Need to remove all of them.
- `UUUU | DDDD | UUU | DDDD | UUUU | DDDD | UU`
- Can not remove more than two splits
  - Any split with its both elements outside the operation or inside the operation remains a split
- We can remove the optimal number of splits with each operation.
  - Use the operation on the second block.
  - If only two blocks remain, than it's final move.
  - Otherwise we remove two splits.
- Answer is half the number of splits rounded up.

# Patio

# Patio

- The pavement must use $k^2$ tiles for some integer $k \geq 3$.
- $k^2 \leq n$
- $k \leq \sqrt{n}$, thus need to try only $\sqrt{n}$ different sizes.
- In total, only $n \cdot \sqrt{n}$ candidates for the nice pavement.
- Solution in time $\mathcal{O}(n \cdot \sqrt{n})$ will pass easily.

- Let $r$ be the number of red tiles in the block, $b$ be the number of blue ones.
- The block is valid if $r = (k-2)^2$ and $b = 4k - 4$ (or with $r$ and $b$ swapped).
- Try all values $3 \leq k \leq \sqrt{n}$ and all starting positions.
- Quickly maintain the values of $r$ and $b$.

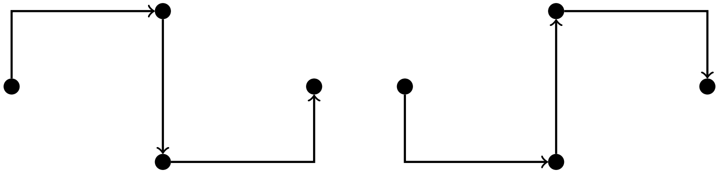
# Volcanoes

# Volcanoes

- If there won't be any any point with common $x$ coordinate, we would only sort the points and go from left to right.

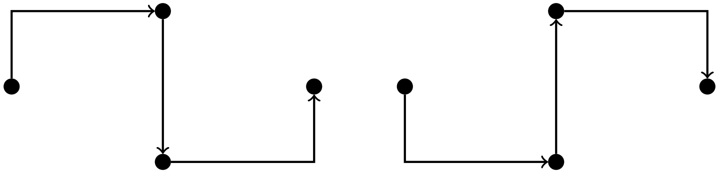
# Volcanoes

- If there won't be any any point with common $x$ coordinate, we would only sort the points and go from left to right.
- Observation: The only interesting points with similar $x$ coordinates are the lowest and highest.



# Volcanoes

- If there won't be any any point with common $x$ coordinate, we would only sort the points and go from left to right.
- Observation: The only interesting points with similar $x$ coordinates are the lowest and highest.



- We can build DAG from each of the bottommost/topmost node of each $x$ coordinate to the bottommost/topmost node of the following $x$ coordinate.
- Use dynamic programming: $\mathcal{O}(N)$
- Alternatively use Dijkstra: $\mathcal{O}(N \log_2(N))$

# Wagon

# Wagon

- Naive solution:
  - If you don't have any item try to buy any of the items (and carry it futher) or none.
  - If you have an item try to either sell it or carry it futher.
- Complexity: $\mathcal{O}(M^N)$

# Wagon

- Naive solution:
  - If you don't have any item try to buy any of the items (and carry it futher) or none.
  - If you have an item try to either sell it or carry it futher.
- Complexity: $\mathcal{O}(M^N)$
- Optimization - use dynamic programming. If you remember which item you bought the complexity would be $\mathcal{O}(MN^2)$

# Wagon

- Naive solution:
  - If you don't have any item try to buy any of the items (and carry it futher) or none.
  - If you have an item try to either sell it or carry it futher.
- Complexity: $\mathcal{O}(M^N)$
- Optimization - use dynamic programming. If you remember which item you bought the complexity would be $\mathcal{O}(MN^2)$
- This can be futher optimized if you jump through bought items only if you build one.
- To do this you can build some kind of "next" array.
- Complexity: $\mathcal{O}(MN + N\log_2(N))$

# Mower

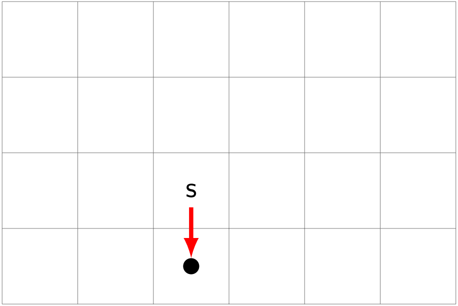
# Mower

- 2-player snake-like game
- decide whether the first player wins

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | s | | | |
| | | | | | |

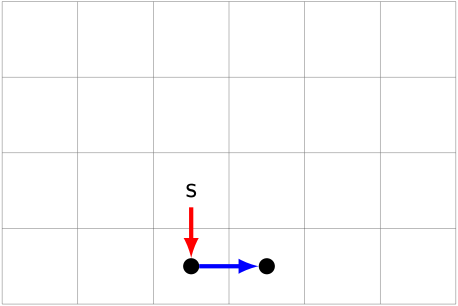
# Mower

- 2-player snake-like game
- decide whether the first player wins



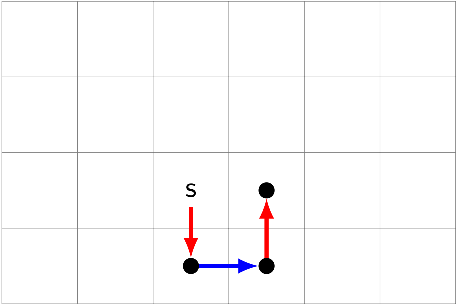
# Mower

- 2-player snake-like game
- decide whether the first player wins



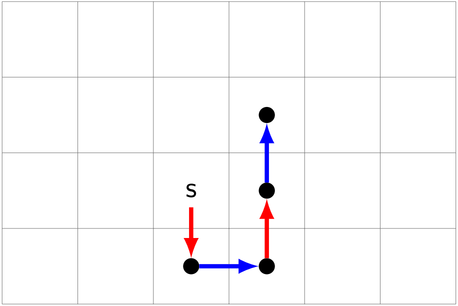
# Mower

- 2-player snake-like game
- decide whether the first player wins



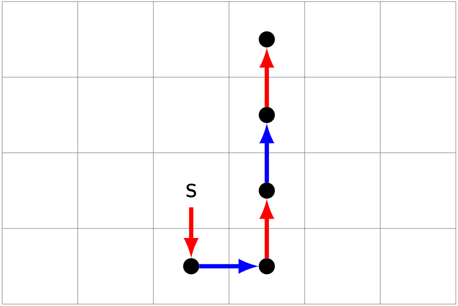
# Mower

- 2-player snake-like game
- decide whether the first player wins



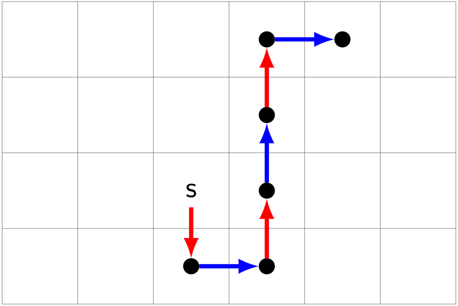
# Mower

- 2-player snake-like game
- decide whether the first player wins



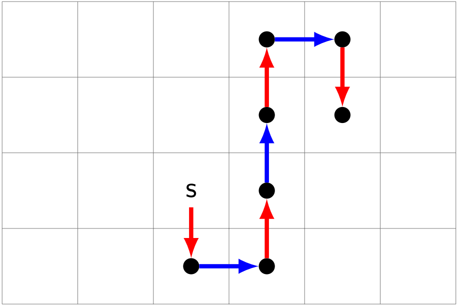
# Mower

- 2-player snake-like game
- decide whether the first player wins



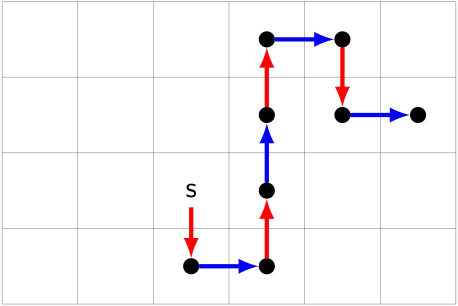
# Mower

- 2-player snake-like game
- decide whether the first player wins



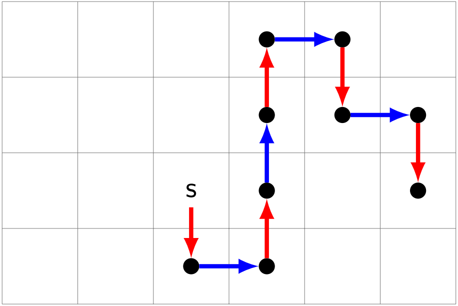
# Mower

- 2-player snake-like game
- decide whether the first player wins



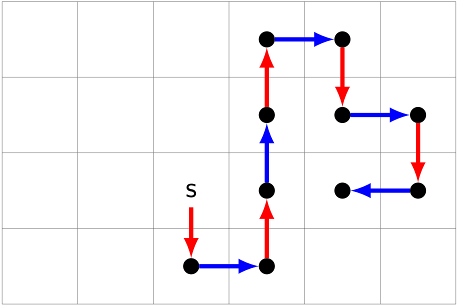
# Mower

- 2-player snake-like game
- decide whether the first player wins



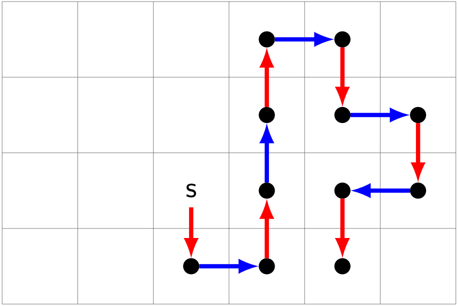
# Mower

- 2-player snake-like game
- decide whether the first player wins



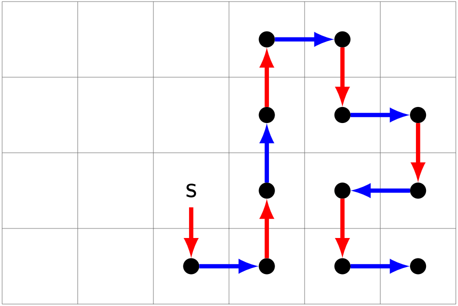
# Mower

- 2-player snake-like game
- decide whether the first player wins



# Mower

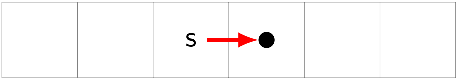
- 2-player snake-like game
- decide whether the first player wins



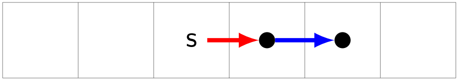
# Mower

| | | | | | |
|---|---|---|---|---|---|
| | | s | | | |

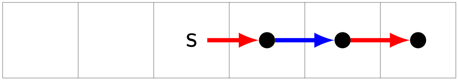
# Mower



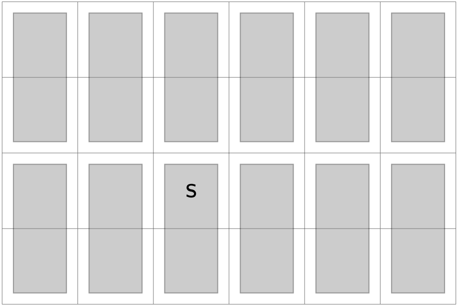
# Mower



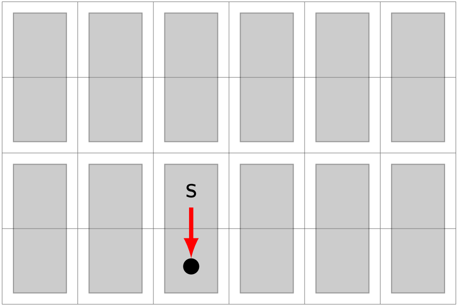
# Mower



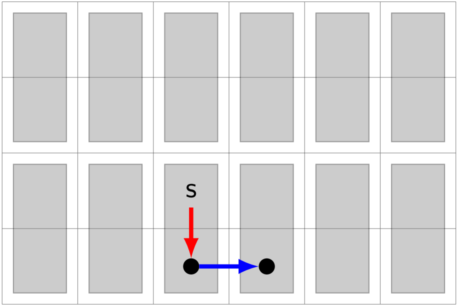
# Mower



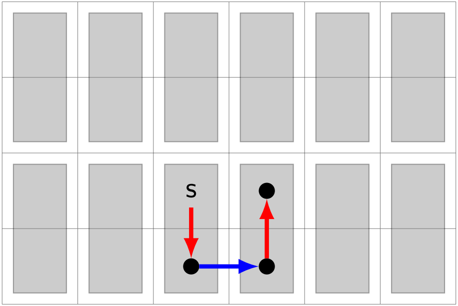
# Mower



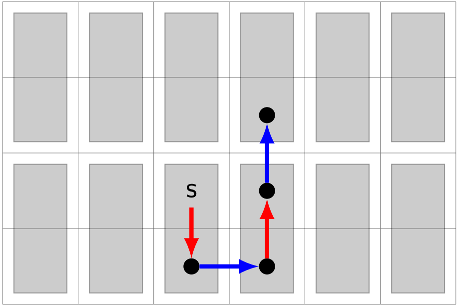
# Mower



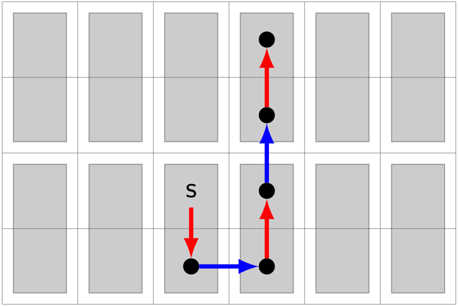
# Mower



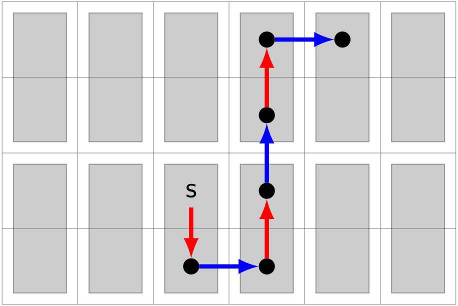
# Mower



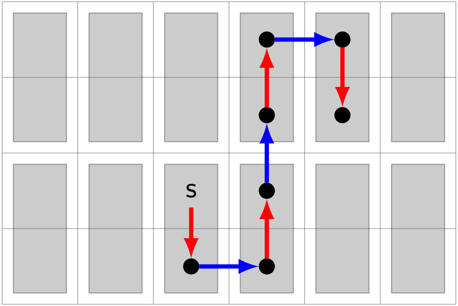
# Mower



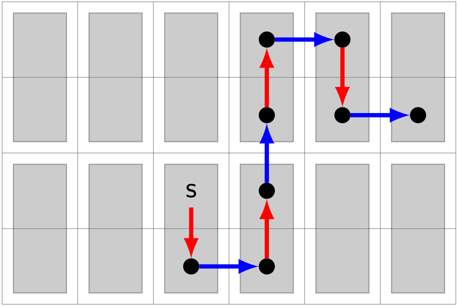
# Mower



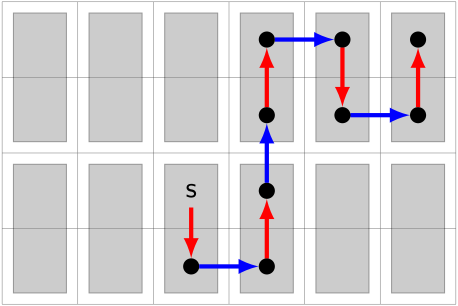
# Mower



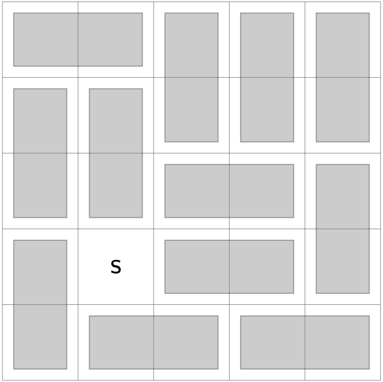
# Mower



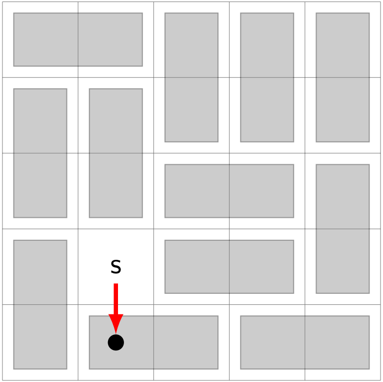
# Mower



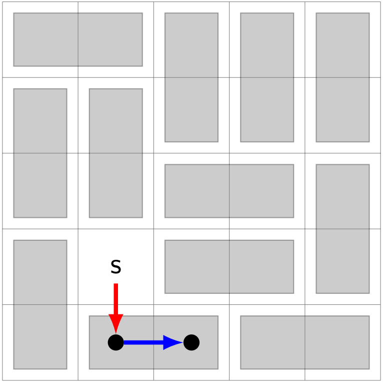
# Mower



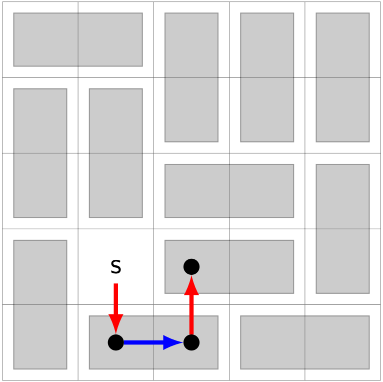
# Mower



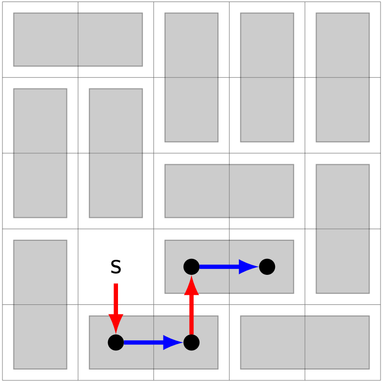
# Mower



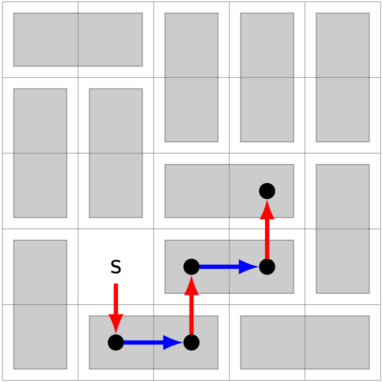
# Mower



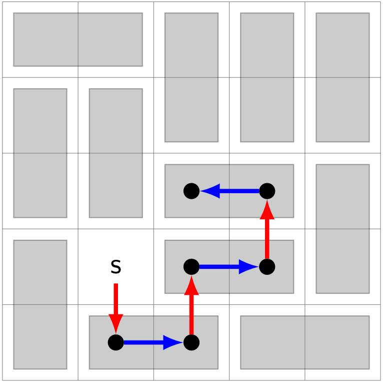
# Mower



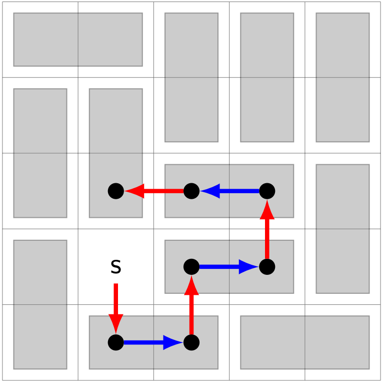
# Mower



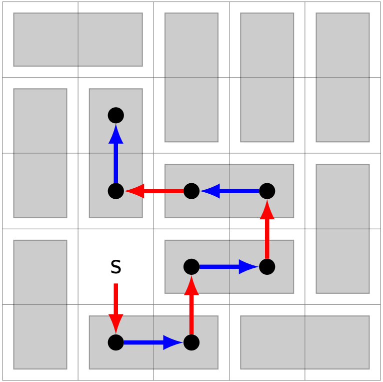
# Mower



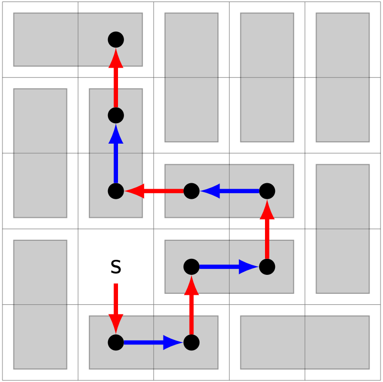
# Mower



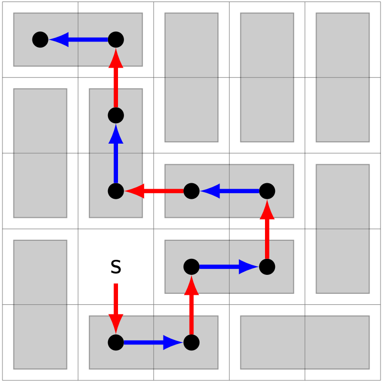
# Mower



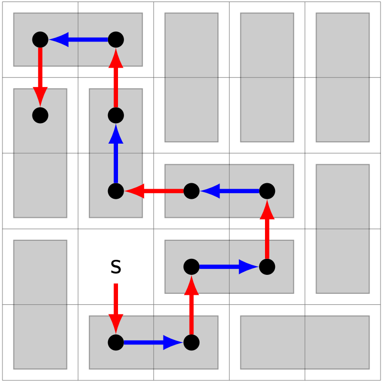
# Mower



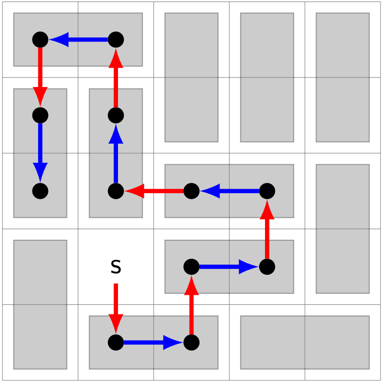
# Mower



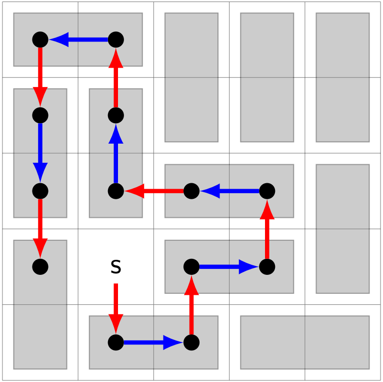
# Mower



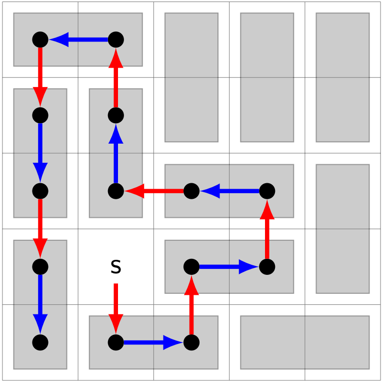
# Mower



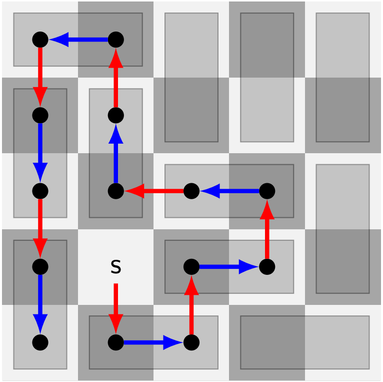
# Mower



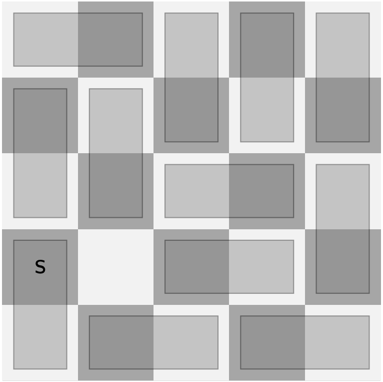
# Mower



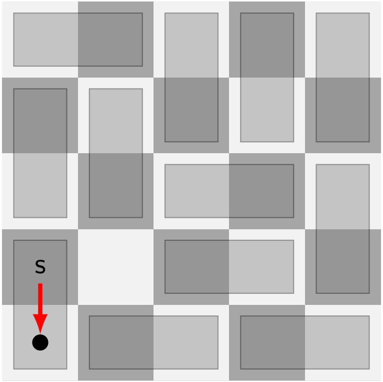
# Mower



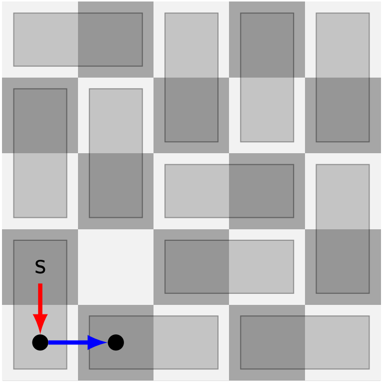
# Mower



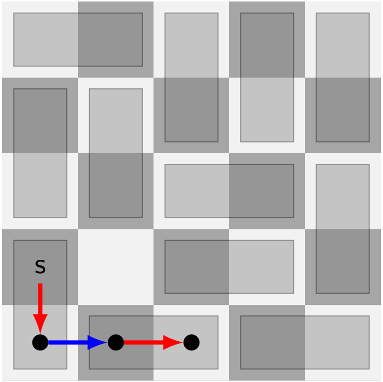
# Mower



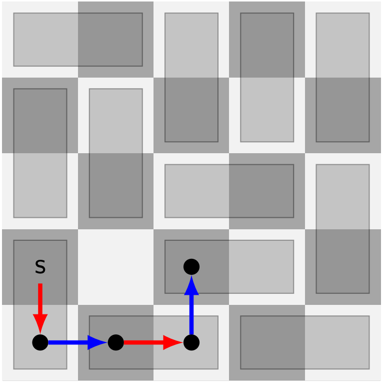
# Mower



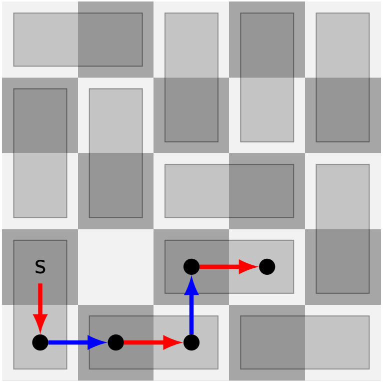
# Mower



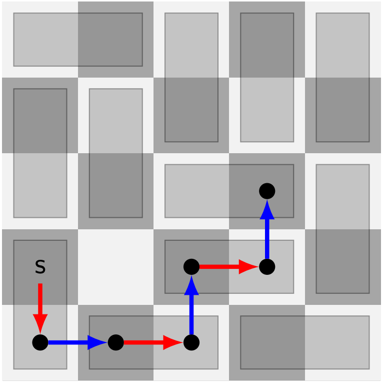
# Mower



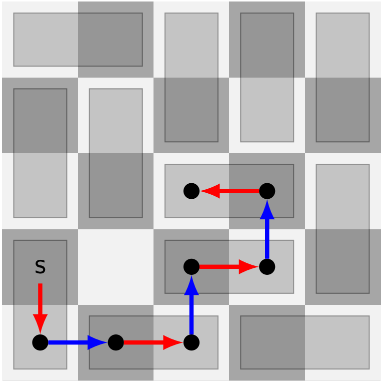
# Mower



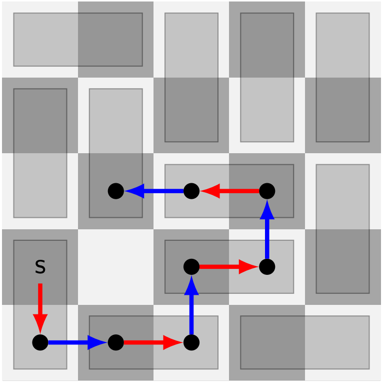
# Mower



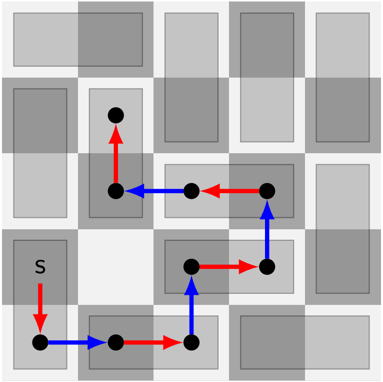
# Mower



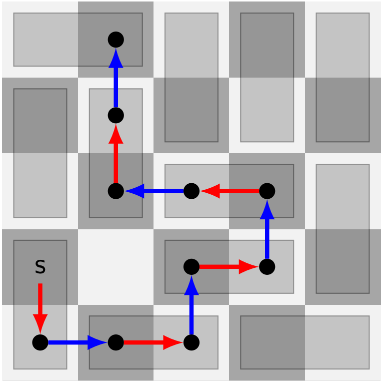
# Mower



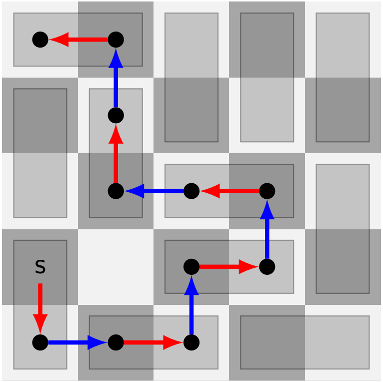
# Mower



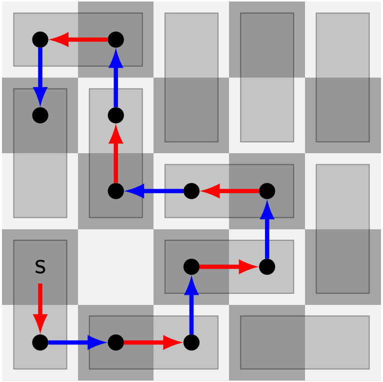
# Mower



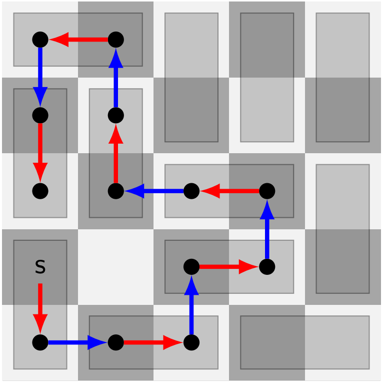
# Mower



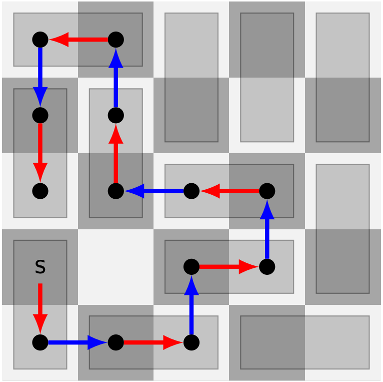
# Mower



# Mower



# Mower




Solution:

```
ll W, H, X, Y; cin >> W >> H >> X >> Y;
cout << ((W%2==0)||(H%2==0)||((X+Y)%2!=0)?"Win":"Lose");
```

# Earthquake

# Earthquake

- Checking all numbers from the old list for each number from the stained list is slow

# Earthquake

- Checking all numbers from the old list for each number from the stained list is slow
- Instead, we build a reverse index

# Earthquake

- Checking all numbers from the old list for each number from the stained list is slow
- Instead, we build a reverse index
- For each number from the old list, generate all possible stained numbers that may correspond to it and increment the counter of each by one

# Earthquake

- Checking all numbers from the old list for each number from the stained list is slow
- Instead, we build a reverse index
- For each number from the old list, generate all possible stained numbers that may correspond to it and increment the counter of each by one
- Upon inspection of a stained number, just return the value in its counter

# Earthquake

- How many possible stained numbers for a particular number from the old list are there?

# Earthquake

- How many possible stained numbers for a particular number from the old list are there?
- Not stained – 1 728147956

# Earthquake

- How many possible stained numbers for a particular number from the old list are there?
- Not stained – 1 728147956
- Coffee stained

Once: $\binom{9}{1} = 9$

?28147956
7?8147956
72?147956
...

Twice: $\binom{9}{2} = 36$

??8147956
?2?147956
?28?47956
...

# Earthquake

- How many possible stained numbers for a particular number from the old list are there?
- Not stained – 1 728147956
- Coffee stained

Once: $\binom{9}{1} = 9$

?28147956
7?8147956
72?147956
...

Twice: $\binom{9}{2} = 36$

??8147956
?2?147956
?28?47956
...

- Juice stained – number of continuous subsequences, that are omitted: $9 + 8 + \ldots + 1 = 45$

$\underbrace{7}_{*}28147956 \quad \underbrace{72}_{*}8147956 \quad \underbrace{728}_{*}147956 \quad \ldots$

# Earthquake

- In total, this is at most 91 possible stained numbers per a number in the old list $= 91 \cdot 10^4 \Rightarrow$ at most $\sim 10^6$ possible stained numbers to be preprocessed

# Robots

# Robots

- Observation - as soon as AtlasTiger is able to get to a field, he can get there every second turn.

# Robots

- Observation - as soon as AtlasTiger is able to get to a field, he can get there every second turn.
- For every node we want to know the first time tiger can get there in "odd" time and in "even" time.

# Robots

- Observation - as soon as AtlasTiger is able to get to a field, he can get there every second turn.
- For every node we want to know the first time tiger can get there in "odd" time and in "even" time.
- Duplicate all nodes (odd/even) and process BFS from node of AtlasTiger.

# Robots

- Observation - as soon as AtlasTiger is able to get to a field, he can get there every second turn.
- For every node we want to know the first time tiger can get there in "odd" time and in "even" time.
- Duplicate all nodes (odd/even) and process BFS from node of AtlasTiger.
- Try to get (by BFS) from start to end.
  - If you step on node earlier then AtlasTiger - you can enter.
  - If you step on node after the first odd occurence of tiger but before first even occurence of tiger, you can enter if and only if the time is even.
  - If you step on node after the first even occurence of tiger but before first odd occurence of tiger, you can enter if and only if the time is odd.
  - If you step on node after first occurence of tiger in both, odd and even times, you can't step on the node.

# Robots

- Observation - as soon as AtlasTiger is able to get to a field, he can get there every second turn.
- For every node we want to know the first time tiger can get there in "odd" time and in "even" time.
- Duplicate all nodes (odd/even) and process BFS from node of AtlasTiger.
- Try to get (by BFS) from start to end.
  - If you step on node earlier then AtlasTiger - you can enter.
  - If you step on node after the first odd occurence of tiger but before first even occurence of tiger, you can enter if and only if the time is even.
  - If you step on node after the first even occurence of tiger but before first odd occurence of tiger, you can enter if and only if the time is odd.
  - If you step on node after first occurence of tiger in both, odd and even times, you can't step on the node.
- Complexity $\mathcal{O}(N)$

# Array

# Array

- Pascal triangle, $i$-th entry on $n$-th row is $\binom{n-1}{i-1}$.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| row 1: | | | | | | 1 | | | | | |
| row 2: | | | | | 1 | | 1 | | | | |
| row 3: | | | | 1 | | 2 | | 1 | | | |
| row 4: | | | 1 | | 3 | | 3 | | 1 | | |
| $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | |
| row $n$: | 1 | | $n-1$ | | $\Theta(n^2)$ | $\cdots$ | $\Theta(n^2)$ | | $n-1$ | | 1 |

- Task: Find topmost occurrence of a number $\leq 10^9$.
- Observation: There is relatively small number of small Pascal numbers (with exception of the obvious ones - on borders).
  - On row $n \geq 44723$, only one new value not greater than $10^9$: $n-1$.
  - On row $n \geq 1820$, only two: $n-1$ and $\binom{n-1}{2}$.

# Array

- Pascal triangle, $i$-th entry on $n$-th row is $\binom{n-1}{i-1}$.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| row 1: | | | | | | 1 | | | | | |
| row 2: | | | | | 1 | | 1 | | | | |
| row 3: | | | | 1 | | 2 | | 1 | | | |
| row 4: | | | 1 | | 3 | | 3 | | 1 | | |
| $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | | $\dots$ | |
| row $n$: | 1 | | $n-1$ | | $\Theta(n^2)$ | $\cdots$ | $\Theta(n^2)$ | | $n-1$ | | 1 |

- Task: Find topmost occurrence of a number $\leq 10^9$.
- Observation: There is relatively small number of small Pascal numbers (with exception of the obvious ones - on borders).
  - On row $n \geq 44723$, only one new value not greater than $10^9$: $n-1$.
  - On row $n \geq 1820$, only two: $n-1$ and $\binom{n-1}{2}$.
- Generate all numbers, store them in map/dictionary and then swiftly answer for each query. If number $n$ is not in map, reply row $n+1$.

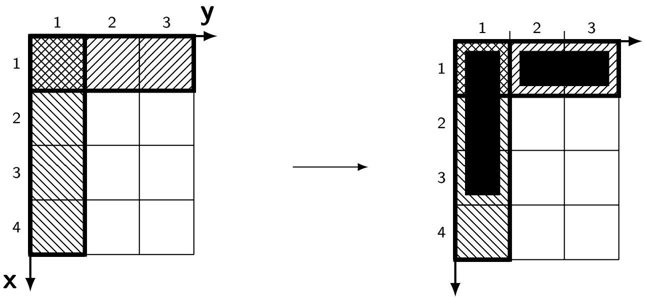
# Canoes

# Canoes

- First we make observations about glaringly impossible cases

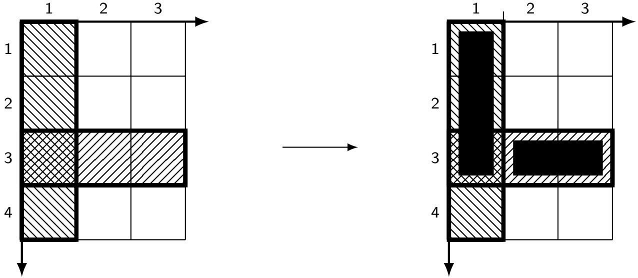
# Canoes

- First we make observations about glaringly impossible cases
- Intersections at the ends of docks are OK ✓



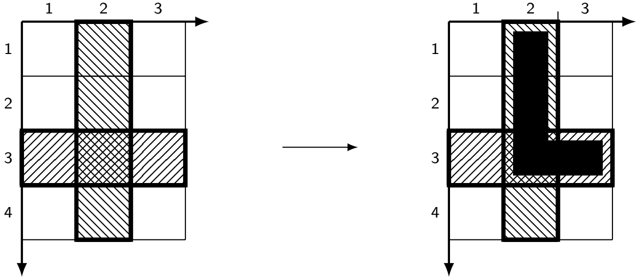
# Canoes

- Intersections of the middle of a dock with an end of a dock are OK ✓



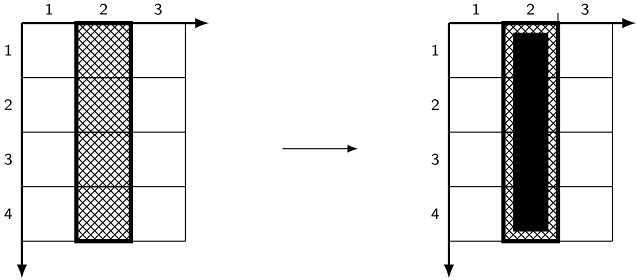
# Canoes

- Intersections of the middle of a dock with the middle of a dock are not OK ✗



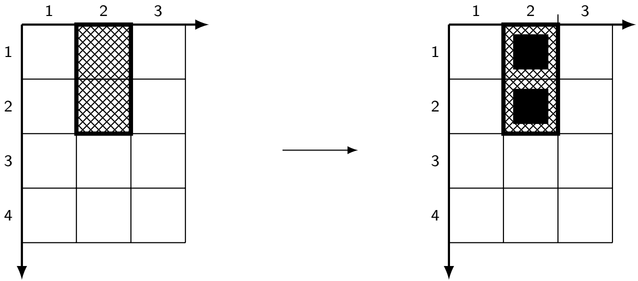
# Canoes

- Intersections of the middle of a dock with the middle of a dock are not OK X
- Also not when two docks coincide



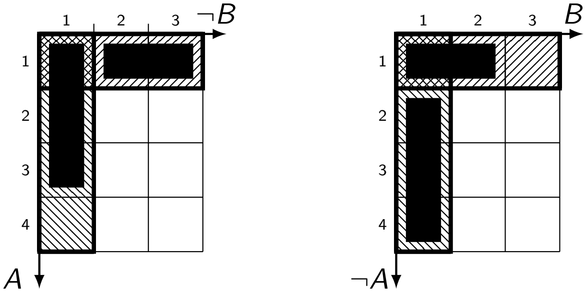
# Canoes

- Intersections of the middle of a dock with the middle of a dock are not OK ✗
- Also not when two docks coincide
- With the exception of square boats ✓



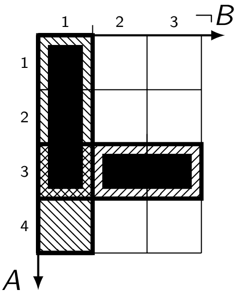
# Canoes

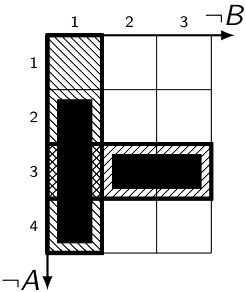
- We model the configuration as implications with the use of the following key: $\uparrow X$, $\downarrow \neg X$, $\leftarrow X$, $\rightarrow \neg X$




- Yields $(A \Rightarrow \neg B) \Leftrightarrow (B \Rightarrow \neg A) \Leftrightarrow (\neg A \vee \neg B)$

# Canoes




- Yields $(\neg B) \Leftrightarrow (\neg B \vee \neg B) \Leftrightarrow (B \Rightarrow \neg B)$

# Canoes

- For $N$ docks, we obtain 2-SAT with $\mathcal{O}(N)$ variables and $\mathcal{O}(N)$ clauses

$$(A \vee \neg B) \wedge (C \vee C) \wedge (\neg C \vee \neg D) \wedge \ldots$$

# Canoes

- For $N$ docks, we obtain 2-SAT with $\mathcal{O}(N)$ variables and $\mathcal{O}(N)$ clauses

$$(A \vee \neg B) \wedge (C \vee C) \wedge (\neg C \vee \neg D) \wedge \ldots$$

- We employ a SCC-based 2-SAT algorithm, which provides solution in $\mathcal{O}(N + M)$ for $N$ variables and $M$ clauses

# Canoes

- For $N$ docks, we obtain 2-SAT with $\mathcal{O}(N)$ variables and $\mathcal{O}(N)$ clauses

$$(A \vee \neg B) \wedge (C \vee C) \wedge (\neg C \vee \neg D) \wedge \dots$$

- We employ a SCC-based 2-SAT algorithm, which provides solution in $\mathcal{O}(N + M)$ for $N$ variables and $M$ clauses
- Complexity: $\mathcal{O}(N)$

# Transmitters

# Transmitters

- Cost of the block of strings: the sum of lengths of longest common prefixes for all pairs of strings.
- `aaabc`
- `abbc`
- `aaabx`

# Transmitters

- Cost of the block of strings: the sum of lengths of longest common prefixes for all pairs of strings.
- `aaabc`
- `abbc`
- `aaabx`

# Transmitters

- Cost of the block of strings: the sum of lengths of longest common prefixes for all pairs of strings.
- `aaabc`
- `abbc`
- `aaabx`

# Transmitters

- Cost of the block of strings: the sum of lengths of longest common prefixes for all pairs of strings.
- `aaabc`
- `abbc`
- `aaabx`

# Transmitters

- For every $i$, find the minimum index $j$ such that block $[i, j]$ has cost at least $K$.
- Use sliding window: Note that as $i$ increases, $j$ can not decrease.

# Transmitters

- For every $i$, find the minimum index $j$ such that block $[i, j]$ has cost at least $K$.
- Use sliding window: Note that as $i$ increases, $j$ can not decrease.
- Use *trie* to keep track of cost:
  - Contains all the strings in block $[i, j]$.
  - Count how many times each prefix appears.
  - Make sure to update count when adding/removing strings.
- Linear complexity.

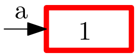
# Transmitters

→ aaabc

abbc

aaabx

Cost: 0



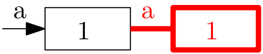
# Transmitters

$\rightarrow$ `aaabc`

`abbc`

`aaabx`

Cost: 0



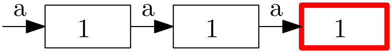
# Transmitters

$\rightarrow$ `aaabc`

`abbc`

`aaabx`

Cost: 0



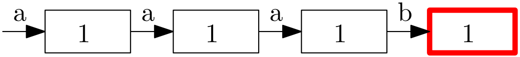
# Transmitters

$\rightarrow$ `aaabc`

`abbc`

`aaabx`

Cost: 0



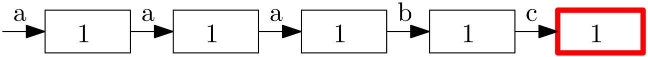
# Transmitters

$\rightarrow$ `aaabc`

`abbc`

`aaabx`

Cost: 0



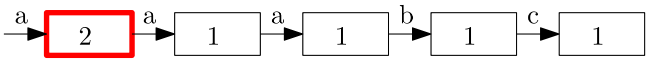
# Transmitters

→ `aaabc`

→ `abbc`

`aaabx`

Cost: 1



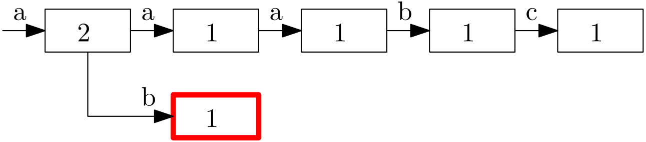
# Transmitters

→ `aaabc`

→ `abbc`

`aaabx`

Cost: 1



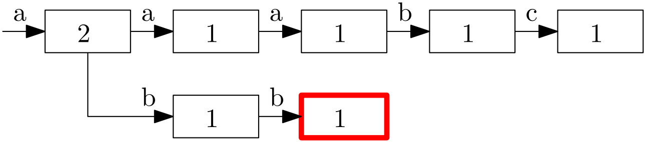
# Transmitters

$\rightarrow$ `aaabc`

$\rightarrow$ `abbc`

`aaabx`

Cost: 1



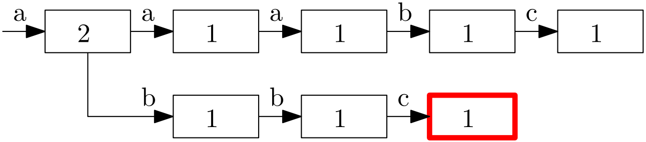
# Transmitters

$\rightarrow$ `aaabc`

$\rightarrow$ `abbc`

`aaabx`

Cost: 1



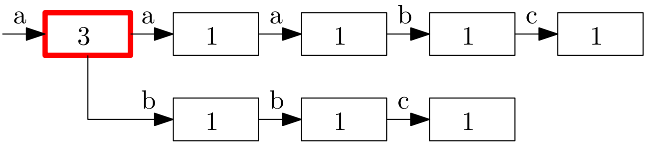
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 3



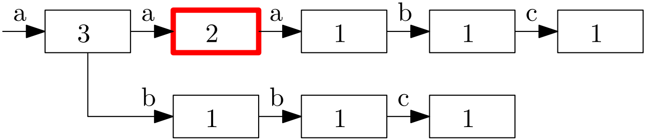
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 4



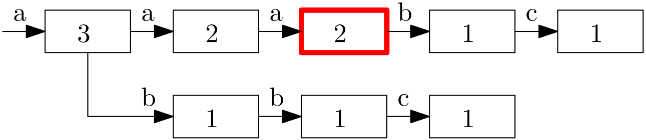
# Transmitters

$\rightarrow$ `aaabc`

$\rightarrow$ `abbc`

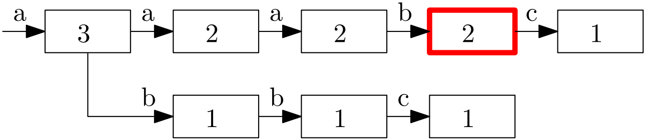
$\rightarrow$ `aaabx`

Cost: 5



# Transmitters

→ aaabc
→ abbc
→ aaabx
Cost: 6



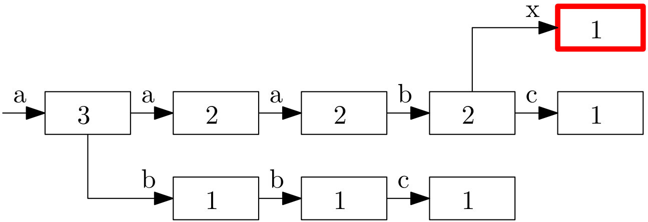
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 6



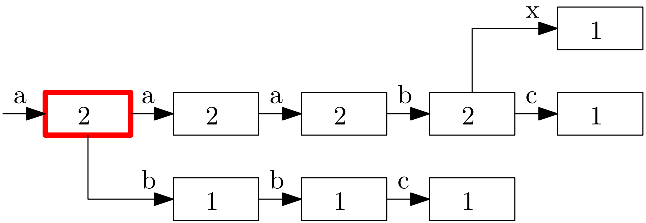
# Transmitters

→ aaabc

→ abbc

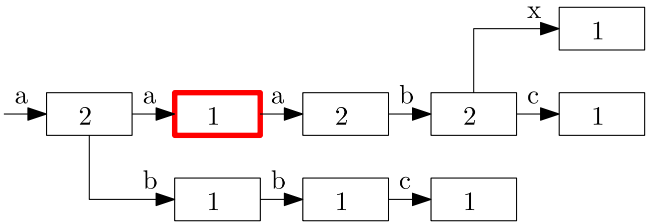
→ aaabx

Cost: 4



# Transmitters

→ aaabc
→ abbc
→ aaabx
Cost: 3



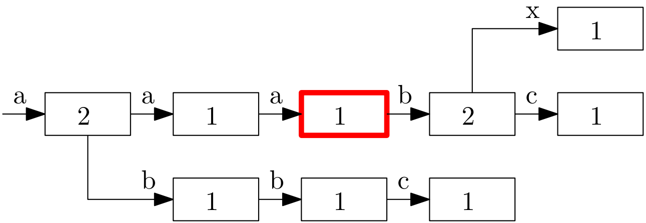
# Transmitters

→ aaabc

→ abbc

→ aaabx

Cost: 2



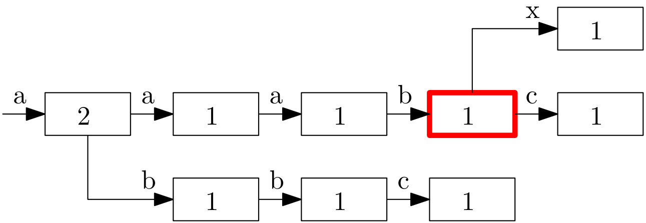
# Transmitters

$\rightarrow$ `aaabc`

$\rightarrow$ `abbc`

$\rightarrow$ `aaabx`

Cost: 1



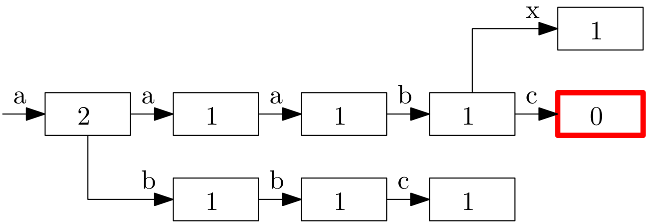
# Transmitters

→ aaabc

→ abbc

→ aaabx

Cost: 1



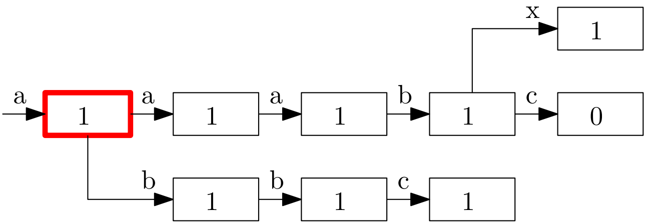
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



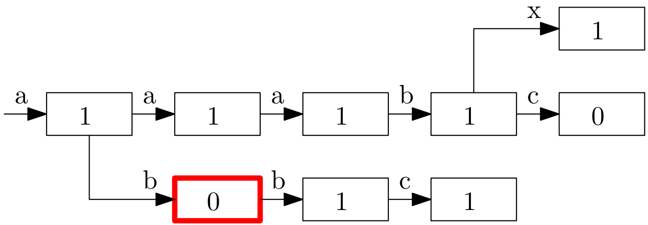
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



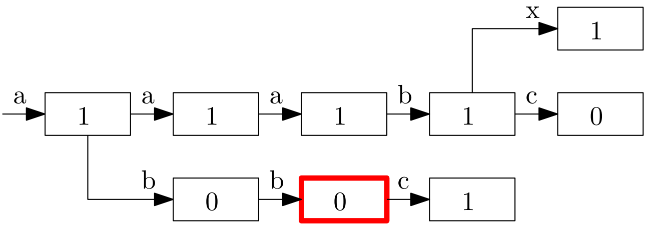
# Transmitters

→ `aaabc`

→ `abbc`

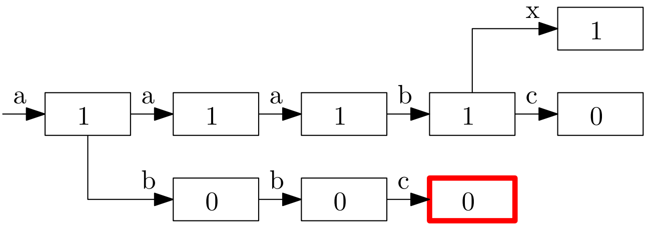
→ `aaabx`

Cost: 0



# Transmitters

$\rightarrow$ `aaabc`
$\rightarrow$ `abbc`
$\rightarrow$ `aaabx`
Cost: 0



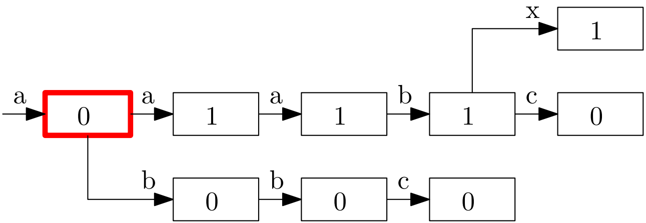
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



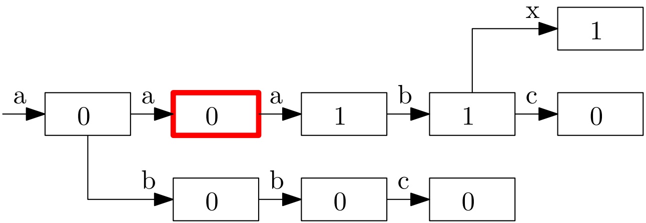
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



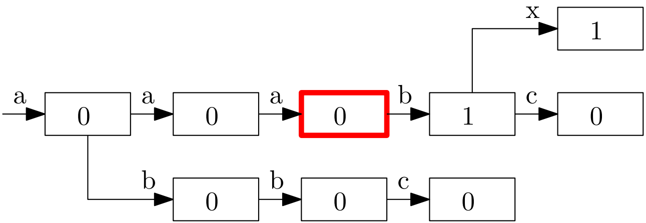
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



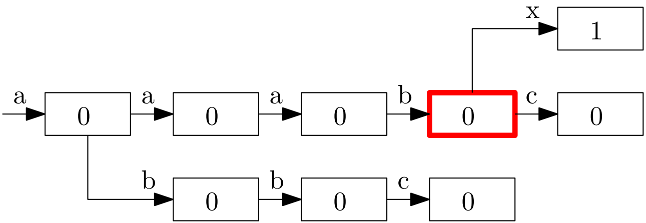
# Transmitters

→ `aaabc`

→ `abbc`

→ `aaabx`

Cost: 0



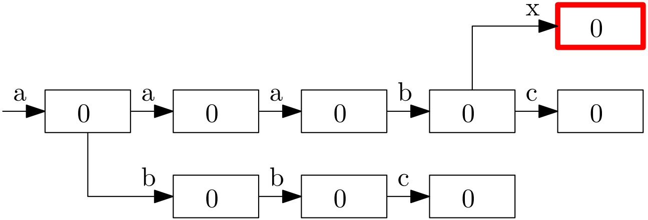
# Transmitters

$\rightarrow$ `aaabc`

$\rightarrow$ `abbc`

$\rightarrow$ `aaabx`

Cost: 0



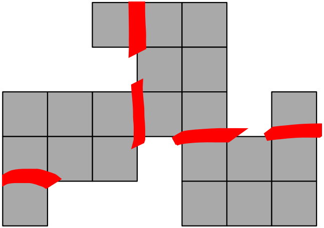
# Transmitters

- Alternatively use hashing!
- For each prefix, keep track of how many times it is in the sliding window.
- Use rolling hash to quickly compute the next hash.
- Linear solution.
- Watch out for collisions!

# Shamans

# Shamans

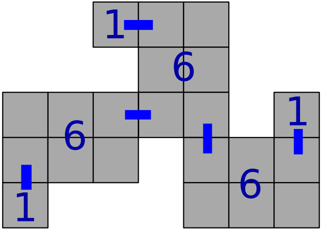
- Construct a graph: each tile is a vertex, connect by edges tiles sharing an edge.
- We can cut two tiles if their edge is a **bridge** (its removal makes the graph disconnected).



- We can identify bridges in $\mathcal{O}(n + m)$.

# Shamans

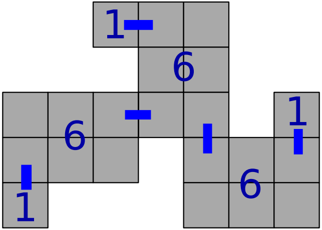
- Try all possible sizes of the cut parchments.
  - Must be a divisor of $n$, thus only at most $2 \cdot \sqrt{n}$ possibilities.
- First pick the size of the cut parchments. Then check if it's valid.




- 21 blocks in total. Try sizes 1, 3, **7**, 21.
- Go bottom up: merge biconnected components until they reach the correct size.
- Then check its shape and remove the component.

# Shamans

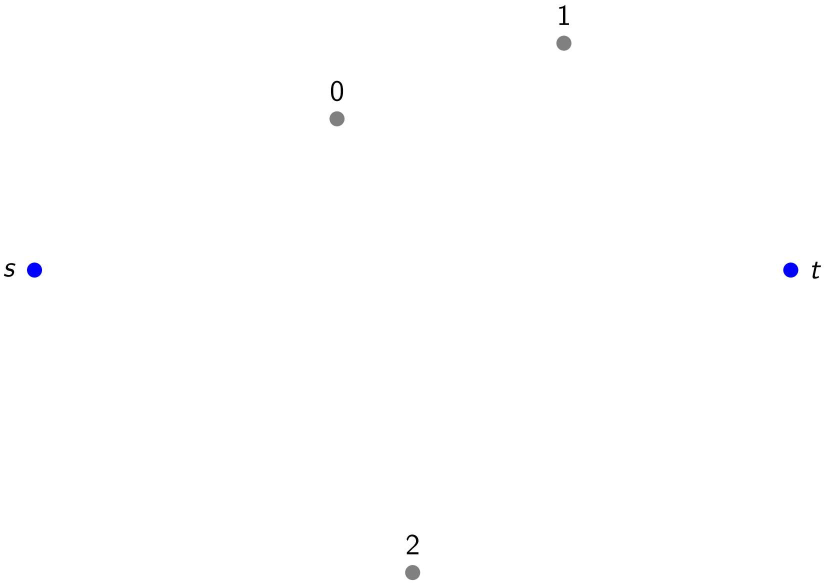
- Try all possible sizes of the cut parchments.
  - Must be a divisor of $n$, thus only at most $2 \cdot \sqrt{n}$ possibilities.
- First pick the size of the cut parchments. Then check if it's valid.



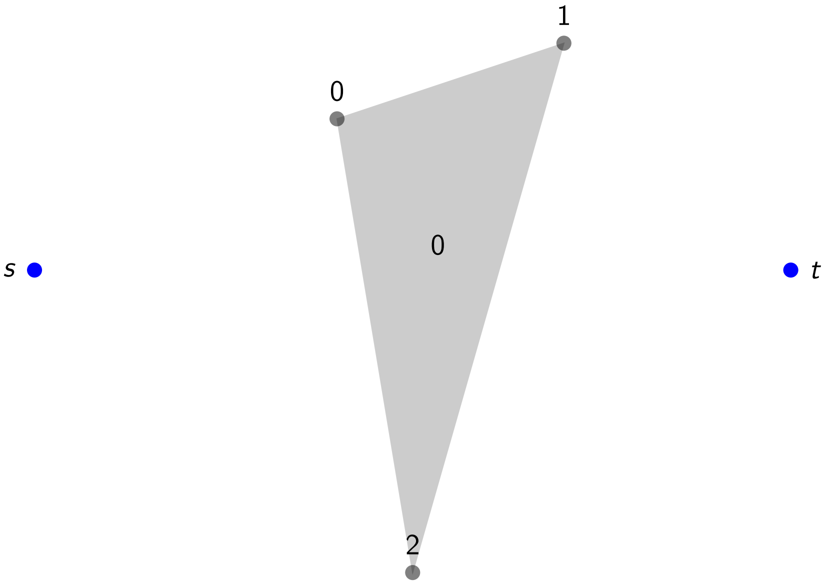

- 21 blocks in total. Try sizes 1, 3, **7**, 21.
- Go bottom up: merge biconnected components until they reach the correct size.
- Then check its shape and remove the component.
- $\mathcal{O}(n)$ for one size of the cut parchments, total running time $\mathcal{O}(n\sqrt{n})$.

# Needle

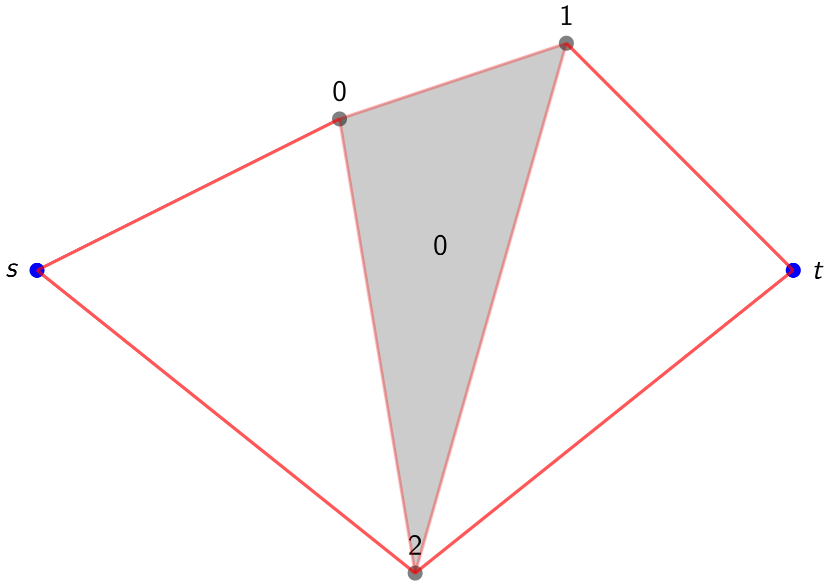
# Needle



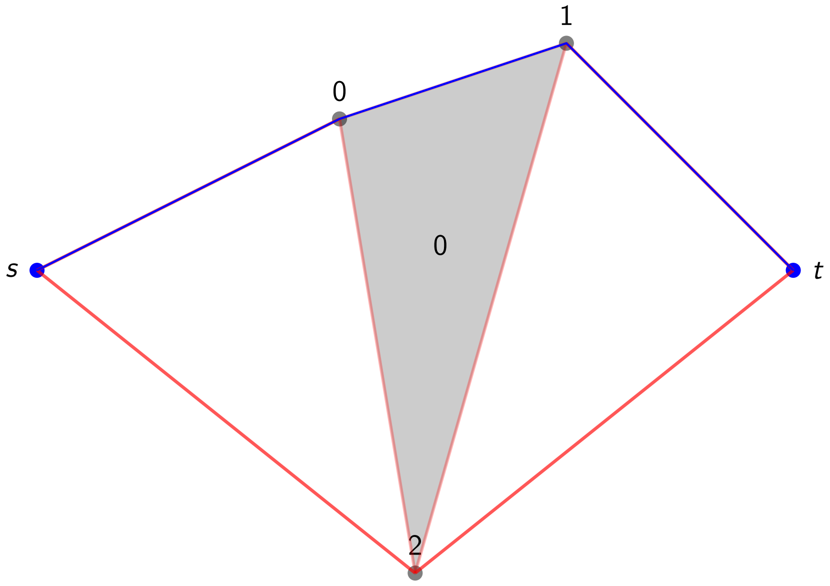
# Needle



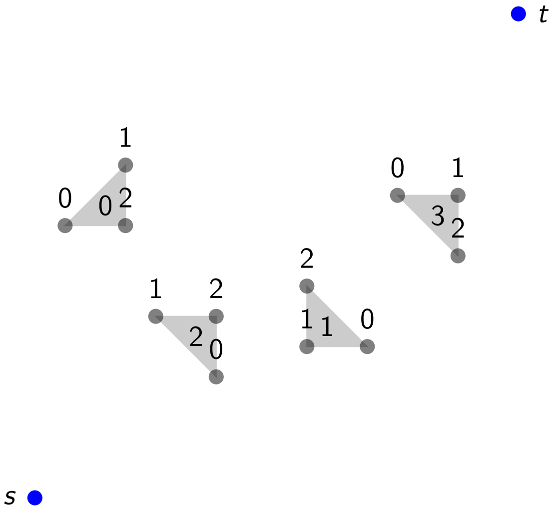
# Needle



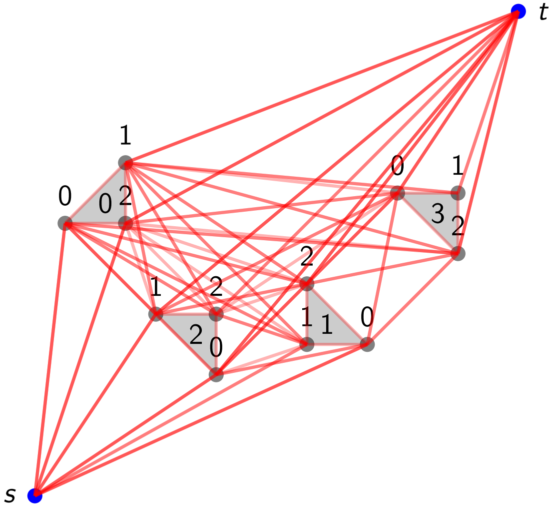
# Needle



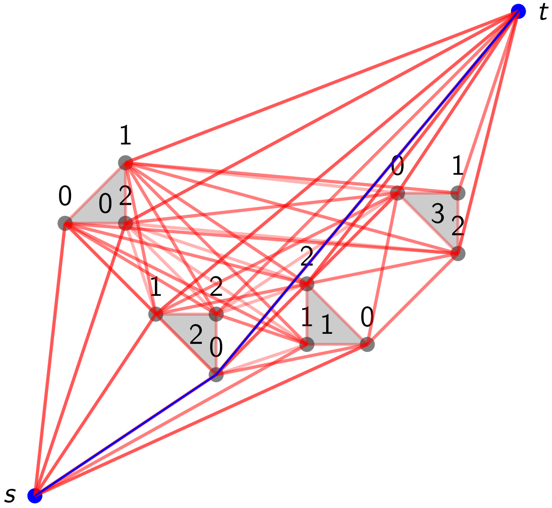
# Needle



# Needle



# Needle



# Needle

- given point clouds find shortest path from $s$ to $t$

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments
- use Dijkstra to find the shortest path

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments
- use Dijkstra to find the shortest path

but to find all viable line segments

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments
- use Dijkstra to find the shortest path

but to find all viable line segments

- find convex hull of every point clouds

# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments
- use Dijkstra to find the shortest path

but to find all viable line segments

- find convex hull of every point clouds
- test every viable line segment on intersection of convex hull's sides

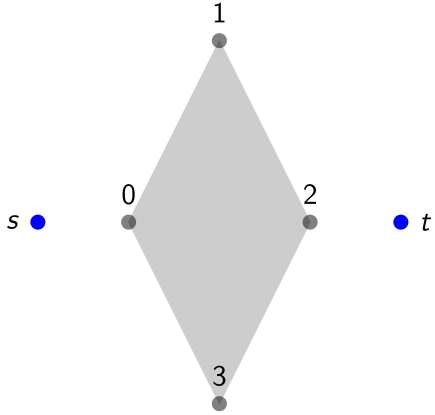
# Needle

- given point clouds find shortest path from $s$ to $t$
- point clouds constitute convex non-touching shapes
- path consists of line segments
- identify all viable line segments
- use Dijkstra to find the shortest path

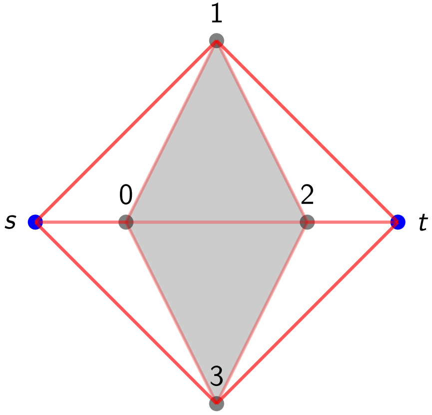
but to find all viable line segments

- find convex hull of every point clouds
- test every viable line segment on intersection of convex hull's sides
- ignore sides adjacent to the segment that is being tested

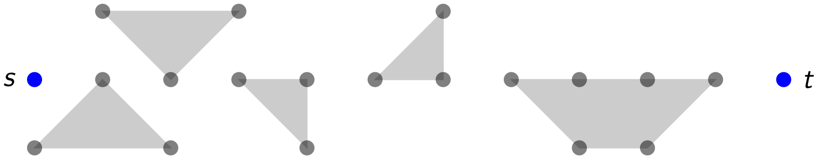
# Needle



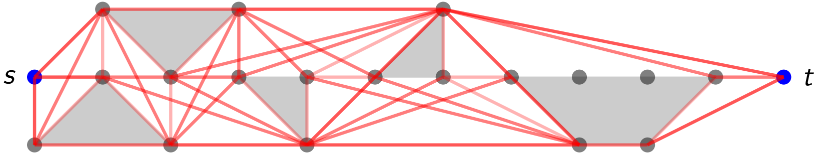
# Needle



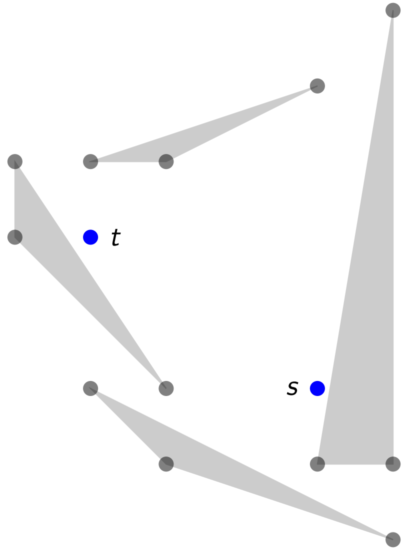
# Needle



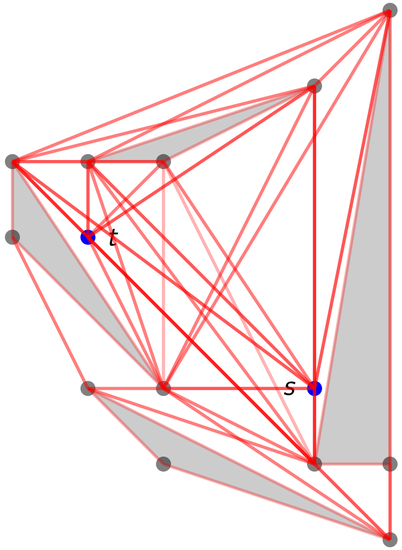
# Needle



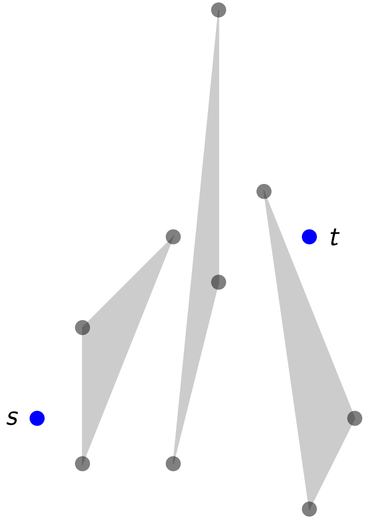
# Needle



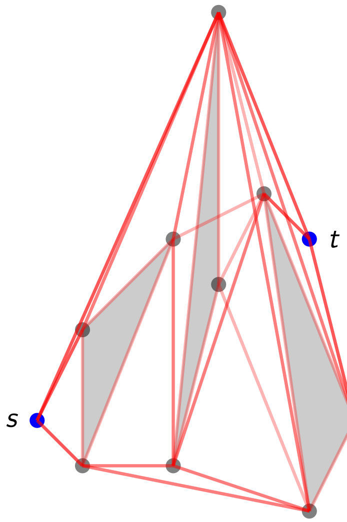
# Needle



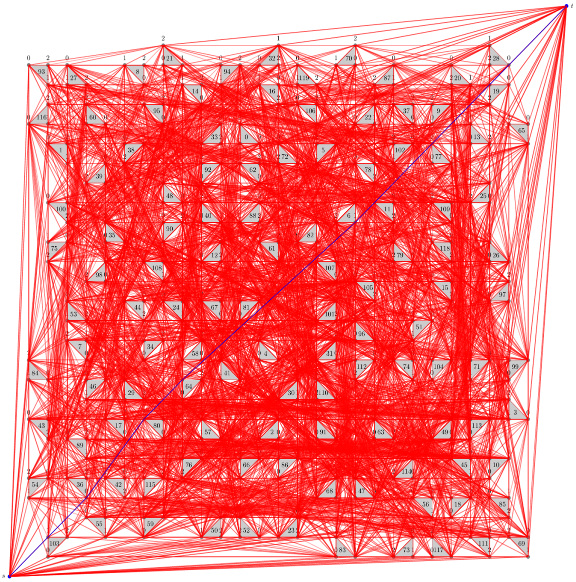
# Needle



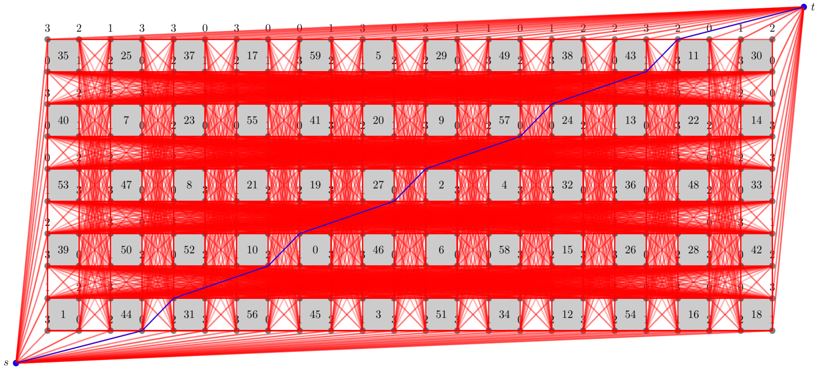
# Needle



# Needle



# Needle


t
s
35 25 37 17 59 5 29 49 38 43 11 30
40 7 23 55 41 20 9 57 24 13 22 14
53 47 8 21 19 27 2 4 32 36 48 33
39 50 52 10 0 46 6 58 15 26 28 42
1 44 31 56 45 3 51 34 12 54 16 18

Thank you for your attention!