ICPC 2020 - Gwalior-Pune Regionals

Presentation of solutions

Credits!

- **Chief Judge**: Balajiganapathi **Balajiganapathi** S
- **Contest Admin**: Vichitr **Vichitr** Gandas
- **Problem Coordinators**: Ashish **Ashishgup** Gupta, Kevin **kevinsogo** Atienza, Vaibhav **xennygrimmato** Tulsyan
- **Setters**: Kevin **kevinsogo** Atienza, Jatin **jtnydv25** Yadav, Vichitr **Vichitr** Gandas, Vaibhav **xennygrimmato** Tulsyan, Saarang **saarang123** Srinivasan
- **Testers**: Kshitij **kshitij_sodani** Sodani, Kevin **kevinsogo** Atienza, Jatin **jtnydv25** Yadav, Ashish **Ashishgup** Gupta, Ajinkya **ajinkya1p3** Parab, Udit **T1duS** Sanghi, Divyansh **failed_coder** Verma, Naveen **mnaveenkumar2009** Kumar, Soumyaditya **socho** Choudhuri, Shashwat **SleepyShashwat** Chandra, Suneet **ScarletS** Mahajan, Vichitr **Vichitr** Gandas, Ritul **ritul_kr_singh** Singh, Varad **playing_it_my_way** Kulkarni, Aswin **aswinashok44** Ashok, Saarang **saarang123** Srinivasan, Vaibhav **xennygrimmato** Tulsyan, Het **hetshah1998** Shah, Mayank **katana_handler** Pugalia
- **Platform Coordinator**: Deepa **deepa_panwar** Panwar
- **RCD**: Dr. Joydip Dhar

Cakewalk	1. Jeff
Simple	2. Vichitrödinger's Cat
Easy	3. Apple Uniformity
Easy	4. Dimitrescu
Easy-Medium	5. Buy n Large
Easy-Medium	6. Fermod
Easy-Medium	7. Shinobu Seven
Medium	8. Neelu in Wanderland
Medium	9. La La Langton's Ant
Medium	10. Pass the Message
Medium	11. House Hunting
Medium-Hard	12. Wakka and Molly

Problem 1

Jeff

No. of accepted solutions: 509

Author: Kevin Charles Atienza

Image Credits: Sarah

Jeff

- If you’re in that situation:
  - There’s no reason to go left.
  - Go right if you could.
  - Clear debris if you couldn’t, so you could afterward.
- *Simulate it*. If you get eaten at any point, the answer is **JEFF**. Otherwise, it’s **JAY**.
- You can also just count debris to the right of `j`. If there are too many, the answer is **JEFF**.

Problem 2

Vichitrödinger's Cat

No. of accepted solutions: 456

Author: Vaibhav Tulsyan

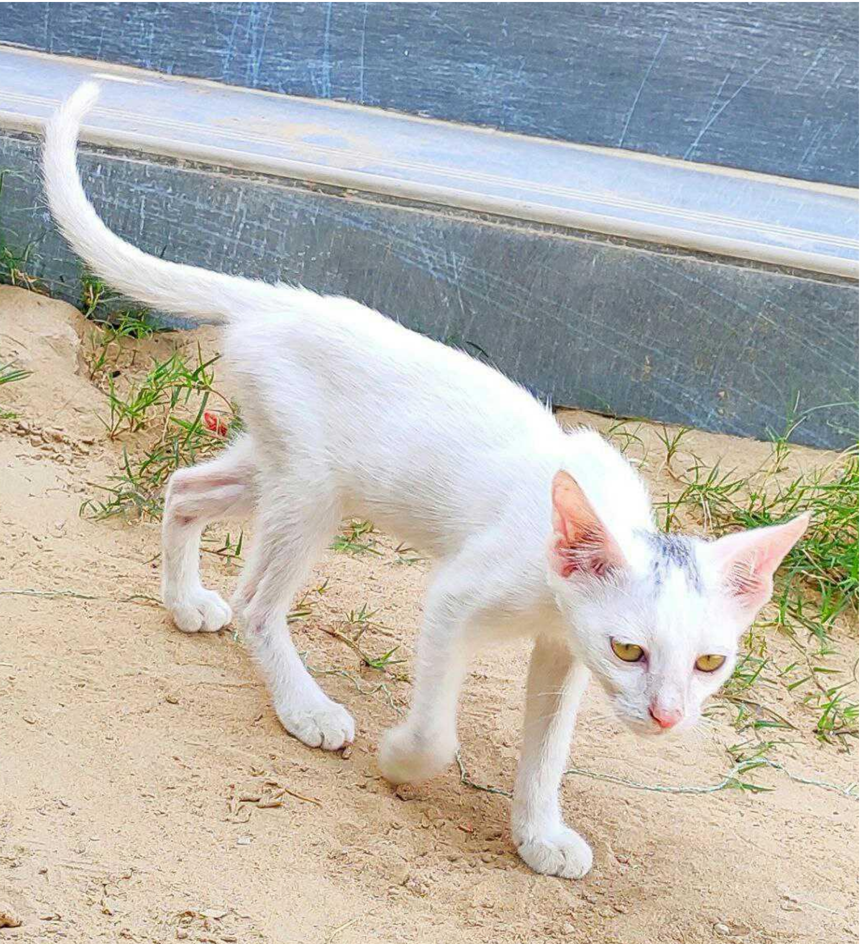
Vichitrödinger's Cat

- For each i, we want to find the largest odd-sized subsequence containing i such that $\mathrm{A[i]}$ is the median of the array.
- **Insight:** *We can “sort” the array.*
- For every valid subsequence in the sorted array, we can “unsort” it to obtain a valid subsequence in the original array with the same median.
- Thus, we can sort the array A!

Vichitrödinger's Cat

- Let S be the sorted version of A.
- Smaller values than $S[i]$ are to its left; larger ones to its right.
  - ...not exactly, because of equal values, but ignore for now.
- Thus, to get the largest odd subsequence with median $S[i]$, greedily take two items, one from each side of $S[i]$, until we can't anymore.
  - We can do this $\min(i - 1, n - i)$ times, so the subarray's size is $1 + 2\cdot\min(i - 1, n - i)$.

Vichitrödinger's Cat



- The answer for $\mathrm{A[i]}$ should now be
  - $1 + 2\cdot\min(j - 1, n - j)$ where j is the location of $\mathrm{A[i]}$ in the array S, i.e., $\mathrm{A[i]} = \mathrm{S[j]}$.
- Hence, the “answers” can be computed in $O(n \log n)$ time.
- **BUT** there may be duplicate values! So it can also be
  - $1 + 2\cdot\min(j' - 1, n - j')$ if j' is another location such that $\mathrm{A[i]} = \mathrm{S[j']}$.
- In fact, the answer must be the largest among all such j.
- But doing so increases the running time to $O(n^2)$!

Vichitrödinger's Cat

- **Insight:** Solve the problem for every *distinct value* $\mathrm{A[i]}$ only once.
- For every other $\mathrm{i'}$ such that $\mathrm{A[i']} = \mathrm{A[i]}$, the answer will be the same.
- The running time goes back to $\mathrm{O(n \log n)}$ again!

Problem 3

Apple Uniformity

No. of accepted solutions: 368

Author: Saarang Srinivasan

Apple Uniformity

- **Observation:** the subarray with the minimum uniformity is always of **length 2**, i.e., $r = l + 1$.
- *Proof:*
    - When you add new elements to a subarray, the maximum can only increase and the minimum can only decrease.
    - Formally, if $S \subseteq T$ then $\max(S) \leq \max(T)$ and $\min(S) \geq \min(T)$.
    - Thus, their difference (i.e., the uniformity) cannot decrease.
- Therefore for a fixed array, the answer will be the minimum difference between adjacent elements of the array.

Apple Uniformity

- When $A[x]$ is updated, only 2 subarrays which change:
  - $[x - 1, x]$ and $[x, x + 1]$.
- So, when $A[x]$ is updated, the old values of $|A[x] - A[x - 1]|$ and $|A[x] - A[x + 1]|$ get removed and replaced by their new values.
- Thus, we want a *data structure* that can store our several values, and such that the following operations can be done quickly:
  - Find the minimum element.
  - Remove some elements.
  - Insert some elements.
- A **multiset** does the job!

Apple Uniformity

- *Alternative data structures:*
    - A *map* where the values are the counts.
    - *Two priority queues*, where the second priority queue represents the “removed” elements.
- These solve the problem in $O((N + Q) \log N)$ time.

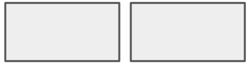
Problem 4

Dimitrescu

No. of accepted solutions: 141

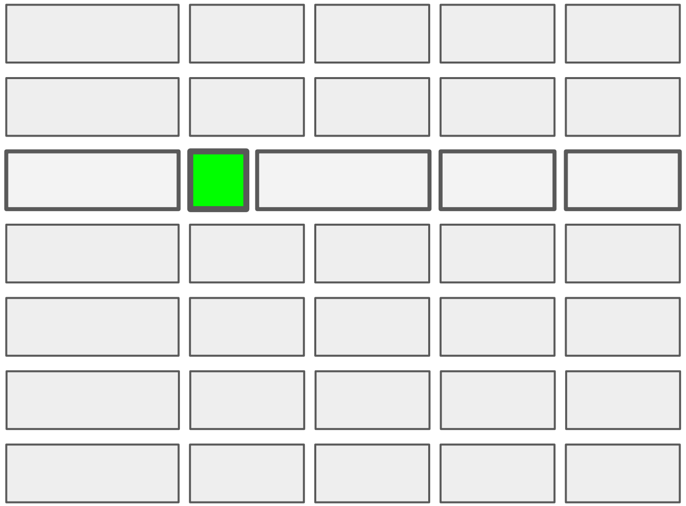
Author: Kevin Charles Atienza

Dimitrescu

- Any row can be tiled with dominoes or triominoes, except if there is just 1 cell.
  - If odd, use 1 triomino, else, use 0 triominoes.
  - 2: 
  - 3: 
  - 4: 
  - 5: 
  - 6: 
  - 7: 

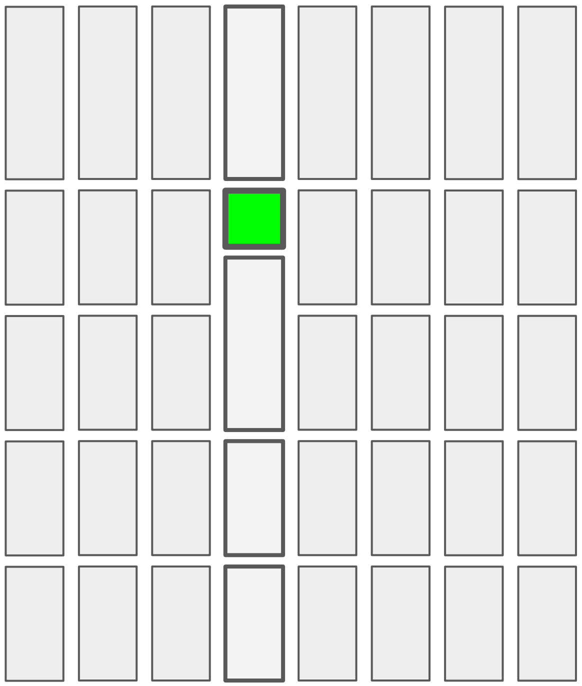
Dimitrescu

- Thus, if (i, j) is not the second cell from the left or right in row i,
  - then we can tile both sides of (i, j),
  - and we can also tile the remaining rows.



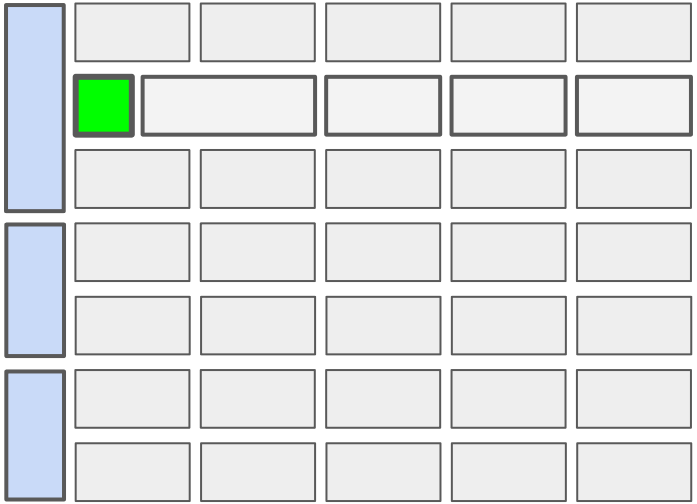
Dimitrescu

- Similarly, if (i, j) is not the second cell from the top or bottom in column j, we can do the tiling.



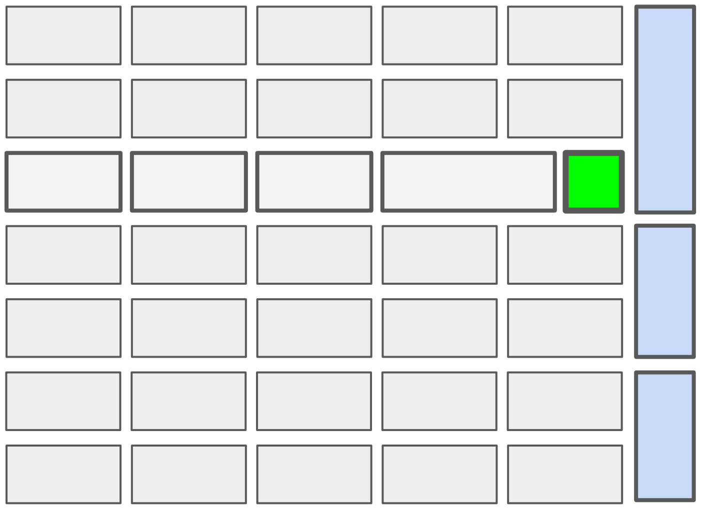
Dimitrescu

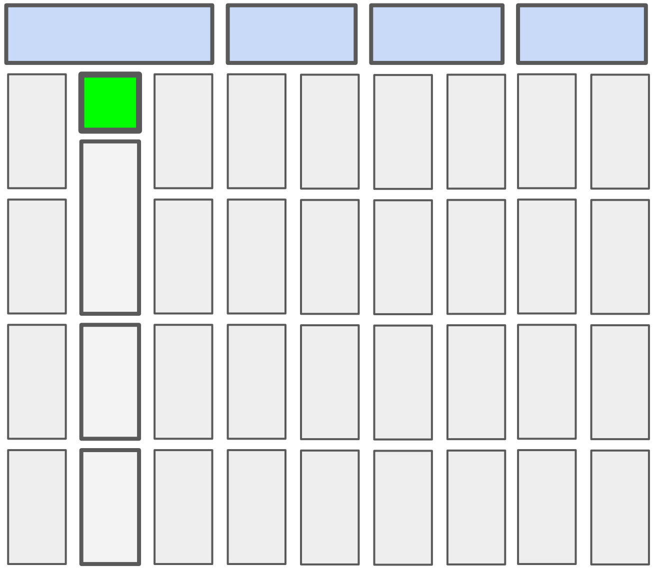
- If (i, j) is the second from the left in row i, then we can tile the first column first.
- Cell (i, j) will become the leftmost, and then we can finish the tiling...
  - ...if $c > 3$, otherwise it is still the second from the right.



Dimitrescu

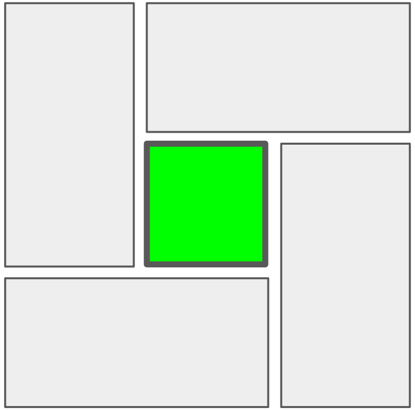
- A similar thing is true if (i, j) is the second from the right in row i.
- Similarly, if (i, j) is second from the top or bottom in column j, then we can finish the tiling if $r > 3$.





Dimitrescu

- The only remaining case is $r = 3$ and $c = 3$, and (i, j) is the middle cell.



Dimitrescu

- *Implementation tips:*
  - Instead of handling all orientations separately, handle it in one orientation (e.g., horizontal, and second from-the-left, not right), and reduce everything else to it via rotations, reflections, etc.
  - Pre-assign letters to different parts of the tiling
    - e.g., tiling a full row, or tiling the row containing (i, j)
  - Alternatively, greedily choose a letter whenever you add a tile to be different from its neighboring tiles.
    - There is always an available letter to choose.

Problem 5

Buy n Large

No. of accepted solutions: 229

Author: Kevin Charles Atienza

Buy n Large

- Two items can only be eventually related if they belong to the same connected component in the initial state.
- Now, if all connected components are *complete* (all items related), then the best solution is to pair them up in reverse order, i.e., cheapest to most expensive, and so on.

Buy n Large

- If the sorted costs are $\mathrm{C}[1], \mathrm{C}[2], \ldots, \mathrm{C}[\mathrm{k}]$, then the cost is
    - $\mathrm{C}[1] + \mathrm{C}[1] + \mathrm{C}[2] + \mathrm{C}[2] + \ldots$ for k terms.
- In other words,
    - the lower half are weighted 2,
    - the upper half are weighted 0,
    - the middle one is weighted 1 (if k is odd).
- Let’s call the items in these categories **low**, **high**, **middle**.

Buy n Large

- Can we achieve this minimum cost, without assuming the component is complete?
- To achieve this cost, we can only pair a low item with a high item.
- We can just buy the middle one first if it exists.
- However, there's always a low item adjacent to a high item, simply because the component is connected!
- Furthermore, this property is still satisfied after the purchase, because the component remains connected.
- Thus, the minimum cost is achievable!
- $\mathbf{O(n \log n)}$

Problem 6

Fermod

No. of accepted solutions: 16

Author: Kevin Charles Atienza

Fermod

- Lots of answers! We can’t hope to describe everything.
- I’ll describe a solution that’s moderately interesting.

Fermod

- In many cases, there are easy-to-find answers.
  - If $m = x^2$, then $(x, x, x, 3)$ is an answer (if $x > 2$).
    - because x^3 is just $0 \pmod{x^2}$
- In fact, if m is not squarefree and $m = p^2k$, then $(pk, pk, pk, 3)$ is an answer.
  - Thus, what remains are squarefree m's.

Fermod

- Let's consider when m is prime first.
- We have Fermat's little theorem: $\mathrm{x}^{\mathrm{m}} \equiv \mathrm{x} \pmod{\mathrm{m}}$.
    - so we use *Fermat's little theorem* to solve *Fermod's last theorem*!
- Therefore, $\mathrm{x}^{\mathrm{m}} + \mathrm{y}^{\mathrm{m}} \equiv (\mathrm{x} + \mathrm{y})^{\mathrm{m}} \pmod{\mathrm{m}}$, so $(\mathrm{x}, \mathrm{y}, \mathrm{x} + \mathrm{y}, \mathrm{m})$ is a "solution" for any x, y.
- However, it doesn't work because the exponent must be $< \mathrm{m}$.

Fermod

- How about the exponent $m - 1$, then? We have $x^{m-1} \equiv 1$,
  - except when $x = 0$.
- But $x^{m-1} + y^{m-1} \equiv 2 \not\equiv 1 \equiv z^{m-1}$ for any x, y, z (except with 0s),
  - so it doesn't work.

Fermod

- What about $m - 2$? Well, $x^{m - 2} \equiv 1/x$ (i.e., the modular inverse).
- But now, we can find easy solutions!
- e.g., $2^{m - 2} + 2^{m - 2} \equiv 1^{m - 2}$, purely because $1/2 + 1/2 = 1$!
- Any fractional equation will do, e.g.,
  - $1/6 + 1/3 = 1/2$,
  - $(-1/2) + (-1/2) = -1$.
- Choose one that will ensure $2 < x, y, z < m$.

Fermod

- The remaining case is when m is squarefree and not prime.
- But we can piggyback on the prime case:
  - If $m = pk$ and (x, y, z, n) is a solution for p, then (xk, yk, zk, n) is a solution for m.
- Alternatively, use the exponent $\varphi(m) - 1$ for the modular inverse.
  - This actually solves the non-squarefree case sometimes!
  - e.g., $(-2)^{\varphi(m) - 1} + (-2)^{\varphi(m) - 1} \equiv (-1)^{\varphi(m) - 1}$
    - if m is not divisible by 2.

Fermod

- We’ve solved everything...with some caveats.
  - Some constructions will not yield $2 < x, y, z < m$.
    - Can sometimes be fixed by scaling $x, y, z \pmod{m}$.
    - Usually, there’s a small multiplier that works, e.g., < 20.
  - For really small m, there may be no multiplier.
    - Just solve small m separately!
    - In fact, only $m = 4$ is impossible.

Problem 7

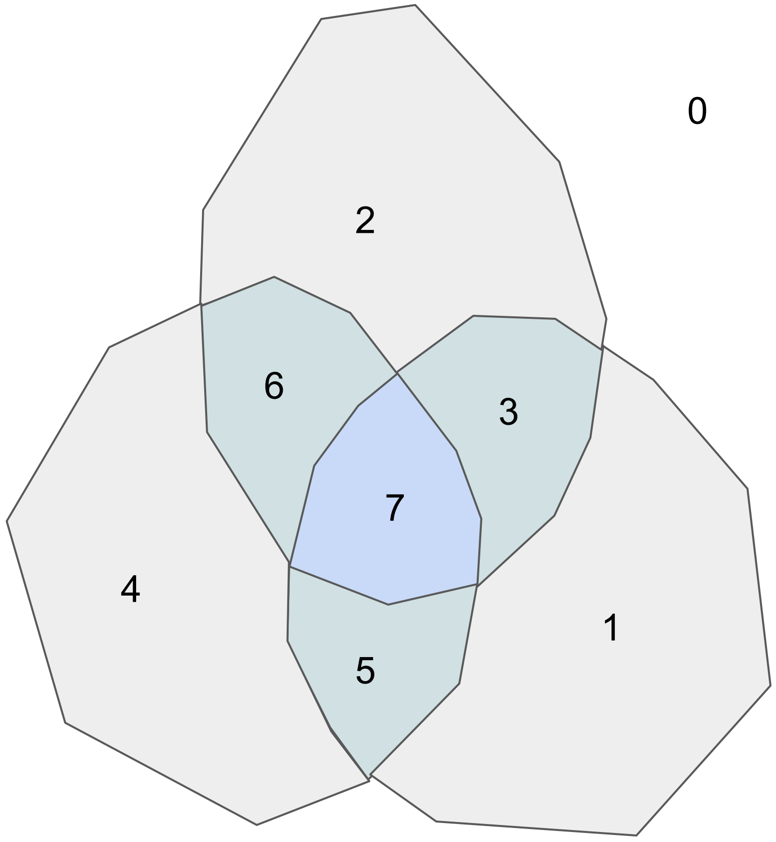
Shinobu Seven

No. of accepted solutions: 1

Author: Kevin Charles Atienza

Shinobu Seven

- k regions partition the plane into 2^{k} **disjoint** regions, one for each combination of (*inside/outside region 1, inside/outside region 2,* etc.)
- So $\mathrm{k} = 3$ polygons partition the plane into 8 regions. One of them is the infinite "outside", so we have 7 finite regions.



Shinobu Seven

- It is much easier to think about these 7 *disjoint* regions than the seven overlapping “union” regions from the statement.
- Given the areas of the 7 overlapping regions, the areas of the 7 disjoint regions can be extracted.
- *Notation:*
  - $\mathbf{X}$ for the area of X (boldface).
  - $\mathrm{A} \cup \mathrm{B}$ for the union of A and B.
  - AB for the intersection of A and B.
  - A' for the complement of A.

Shinobu Seven

The 7 disjoint (finite) regions are:

- ABC
- ABC'
- $AB'C$
- $AB'C'$
- $A'BC$
- $A'BC'$
- $A'B'C$

The 7 overlapping regions are:

- A
- B
- C
- $A \cup B$
- $A \cup C$
- $B \cup C$
- $A \cup B \cup C$

Shinobu Seven

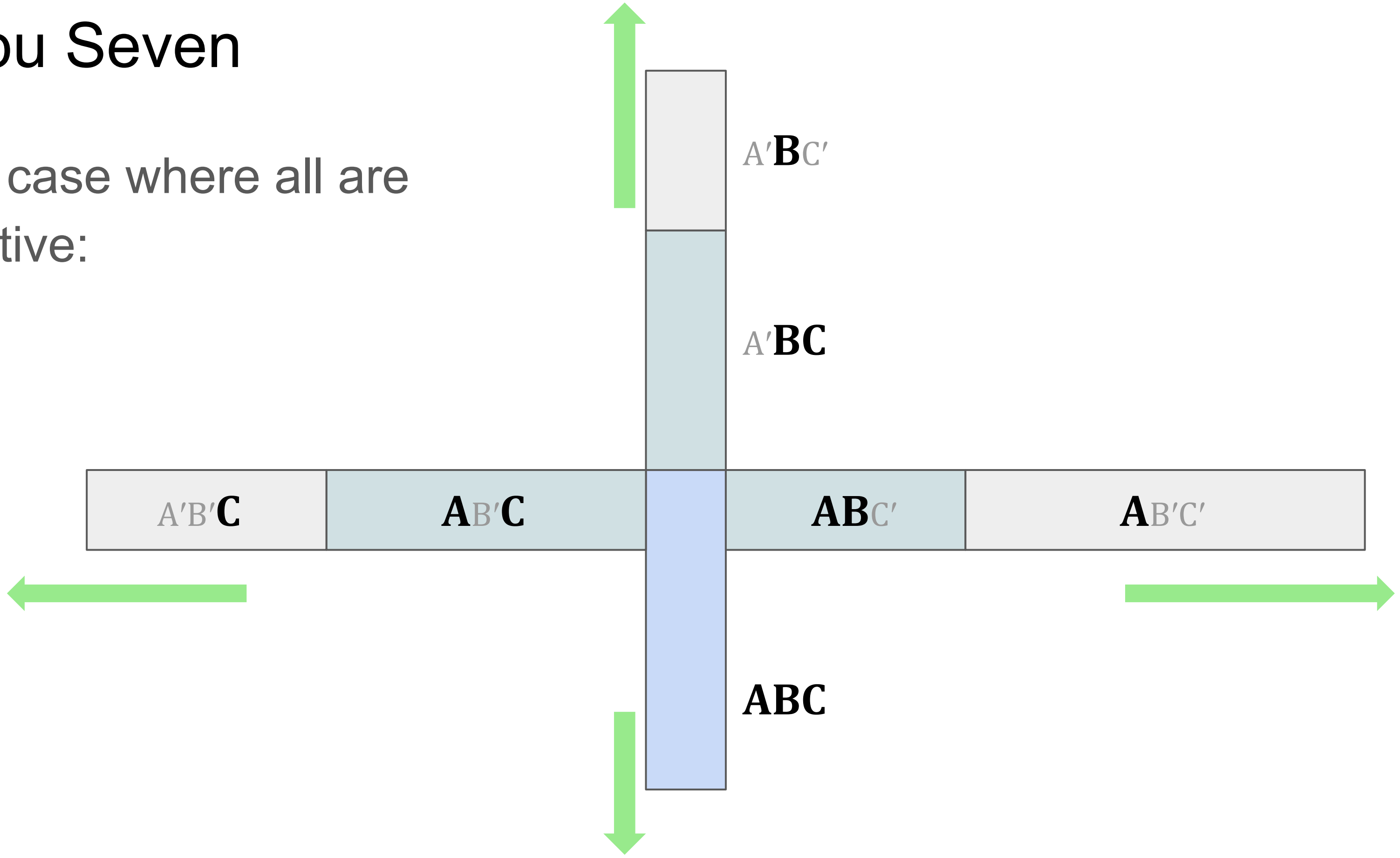
- Then we have the following system:
  - $\mathbf{A} = \mathbf{ABC} + \mathbf{ABC'} + \mathbf{AB'C} + \mathbf{AB'C'}$
  - $\mathbf{B} = \mathbf{ABC} + \mathbf{ABC'} + \mathbf{A'BC} + \mathbf{A'BC'}$
  - $\mathbf{C} = \mathbf{ABC} + \mathbf{AB'C} + \mathbf{A'BC} + \mathbf{A'B'C}$
  - $\mathbf{A} \cup \mathbf{B} = \mathbf{A} \cup \mathbf{B} \cup \mathbf{C}$ - $\mathbf{A'B'C}$
  - $\mathbf{A} \cup \mathbf{C} = \mathbf{A} \cup \mathbf{B} \cup \mathbf{C}$ - $\mathbf{A'BC'}$
  - $\mathbf{B} \cup \mathbf{C} = \mathbf{A} \cup \mathbf{B} \cup \mathbf{C}$ - $\mathbf{AB'C'}$
  - $\mathbf{A} \cup \mathbf{B} \cup \mathbf{C} = \mathbf{ABC} + \mathbf{ABC'} + \mathbf{AB'C} + \mathbf{AB'C'} + \mathbf{AB'C'} + \mathbf{AB'C'} + \mathbf{AB'C'}$
- We can solve for the areas of the disjoint regions!

Shinobu Seven

- Now, the area of each disjoint region must be nonnegative.
    - Otherwise, it is impossible.
- But if all are nonnegative, maybe we could construct a solution.
- Consider a simple case where all 7 regions have positive area.
- The $\mathbf{ABC}$ part will be the “core”, and the 6 remaining regions will be “offshoots”.
- All areas are integers, so we can think in terms of a *grid*, or *building blocks*.

Shinobu Seven

- The case where all are positive:



Shinobu Seven

- In fact, the same solution should work even if some regions are zero. The only important thing is $\mathbf{ABC}$ is positive.
- Thus, the remaining cases are when $\mathbf{ABC} = 0$.
  - For these, the “core” disappears, and we’ll have to do something else.

Shinobu Seven

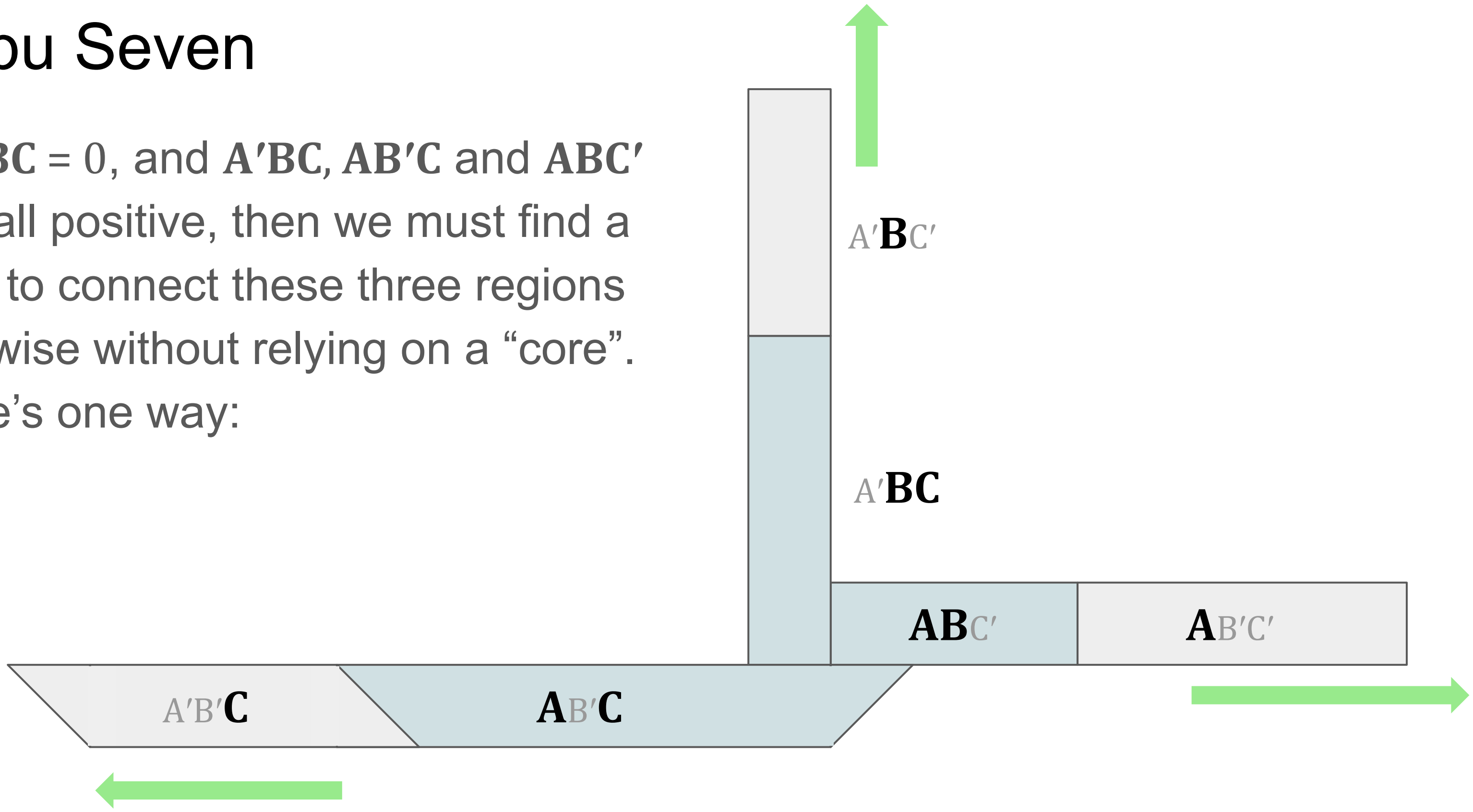
- The case where one of $\mathbf{A'BC}$, $\mathbf{AB'C}$ or $\mathbf{ABC'}$ is 0 is a bit easier.
  - e.g., if $\mathbf{AB'C} = 0$ (and $\mathbf{ABC} = 0$), we can just put them all in a line:

$\mathbf{A}B'C'$	$\mathbf{AB}C'$	$A'\mathbf{B}C'$	$A'\mathbf{BC}$	$A'B'\mathbf{C}$

- Some of the parts here may be 0, but the solution still works.
- Thus, the only remaining case is when $\mathbf{ABC} = 0$, and $\mathbf{A'BC}$, $\mathbf{AB'C}$ and $\mathbf{ABC'}$ are all positive.

Shinobu Seven

- If $\mathbf{ABC} = 0$, and $\mathbf{A'BC}$, $\mathbf{AB'C}$ and $\mathbf{ABC'}$ are all positive, then we must find a way to connect these three regions pairwise without relying on a “core”.
- Here’s one way:



Shinobu Seven

- All cases are now solved.
- There are definitely other solutions.
  - Some may have fewer cases, and some may have more.
- It can be implemented in $O(1)$, but slower solutions (such as building an actual grid) may pass.

Problem 8

Neelu in Wanderland

No. of accepted solutions: 7

Author: Vichitr Gandas

Neelu in Wanderland

- There are four types of $(A[i], S[i])$:
  - (v, d) where $1 \leq v \leq 6$ and $d \in$ {'L', 'R'}
  - $(v, ?)$ where $1 \leq v \leq 6$
  - $(?, d)$ where $d \in$ {'L', 'R'}
  - $(?, ?)$
- The possible offsets are:
  - (v, d): $\{-v\}$ or $\{v\}$ (depending on d)
  - $(v, ?)$: $\{-v, v\}$
  - $(?, d)$: $\{-6, -5, \ldots, -1\}$ or $\{1, 2, \ldots, 6\}$ (depending on d)
  - $(?, ?)$: $\{-6, -5, \ldots., -1, 1, 2, \ldots, 6\}$

Neelu in Wanderland

- Let's make sure the offset *sets* are nonnegative.
- Compute the leftmost position p. Then the possible offsets from p are:
  - (v, d): $\{0\}$
  - $(\mathrm{v}, ?)$: $\{0, 2\mathrm{v}\}$.
  - $(?, \mathrm{d})$: $\{0, 1, \ldots, 5\}$
  - $(?, ?)$: $\{0, 1, \ldots, 5, 7, 8, \ldots, 11\}$.
- For example, $(\mathrm{v}, ?)$ changes from
  "*move* v *to the left or right*" to
  "*stay, or move* $2\mathrm{v}$ *to the right*".

Neelu in Wanderland

- We can convert all offset *sets* to the form $\{0, \mathrm{x}\}$ for some $\mathrm{x} \leq 12$:
  - $\{0, 1, \ldots, 5\}$ is equivalent to 5 sets $\{0, 1\}$.
    - i.e., a movement of up to 5 is equivalent to 5 optional steps to the right.
  - $\{0, \ldots, 5, 7, \ldots, 12\}$ is equivalent to 5 sets $\{0, 1\}$ *and* 1 set $\{0, 7\}$.
    - The "7" jump corresponds to moving to $\{7, 8, \ldots, 12\}$.

Neelu in Wanderland

- Thus, our offset sets now look like $\{0, 1\}, \{0, 2\}, \ldots, \{0, 12\}$.
  - In fact, only 8 among these are possible.
- We want to find all possible locations.
- We process each of these types one by one.
- Let S_x be the set of possible endpoints if we only consider the offset types $\{0, 1\}, \{0, 2\}, \ldots, \{0, x\}$.
  - Initially, S_0 is just $\{0\}$.
  - We want S_{12}.
  - We will compute S_x assuming we already have S_{x-1}.

Neelu in Wanderland

- For offset type $\{0, x\}$, let's say there are c of them.
- Then the new offset set, S_x, is simply
  - $S_{x-1} \cup (S_{x-1} + x) \cup (S_{x-1} + 2x) \cup \ldots \cup (S_{x-1} + cx)$
  - where $S + v$ is defined as $S + v := \{s + v \mid s \in S\}$.
- In other words, $s \in S_{x-1}$ generates these locations:
  - $s, s + x, s + 2x, \ldots, s + cx$
- But this is very similar to a *range update*!

Neelu in Wanderland

- So, given S_{x-1} and offset type $\{0, x\}$ (c of them), our algorithm is now:
  - Create an array `A` of size $|S_{x-1}| + (c + 1)x$.
  - For each $s \in S$: `A[s]++, A[s + (c + 1)x]--`
    - this represents a range update
  - Compute the “accumulation with offset x”:
    - `for i = 0, 1, 2, …: A[i + x] += A[i]`
  - S_x is now the set of indices with a positive value.
- Everything takes $O(d^2n)$ time where $d = 6$ is the size of a die.
  - Better running times are definitely possible.

Neelu in Wanderland

- Everything can be phrased in terms of polynomials.
  - (v, d): x^{-v} or x^{v}
  - $(v, ?)$: $x^{-v} + x^{v}$
  - $(?, d)$: $x^{-6} + x^{-5} + \ldots + x^{-1}$ or $x^{1} + x^{2} + \ldots + x^{6}$
  - $(?, ?)$: $x^{-6} + x^{-5} + \ldots + x^{-1} + x^{1} + x^{2} + \ldots + x^{6}$

Neelu in Wanderland

- Extracting the leftmost offset p corresponds to factoring x^p, so they become:
  - (v, d): 1
  - $(v, ?)$: $1 + x^{2v}$
  - $(?, d)$: $1 + x + \dots + x^5$
  - $(?, ?)$: $1 + x + \dots + x^5 + x^7 + x^8 + \dots + x^{12}$
- And the conversion from $\{0, \dots, 5\}$ to $5\ \{0, 1\}$ is just the fact that $1 + x + \dots + x^5$ has the same terms with positive coefficients as $(1 + x)^5$, etc.

Problem 9

La La Langton's Ant

No. of accepted solutions: 0

Author: Kevin Charles Atienza

La La Langton’s Ant

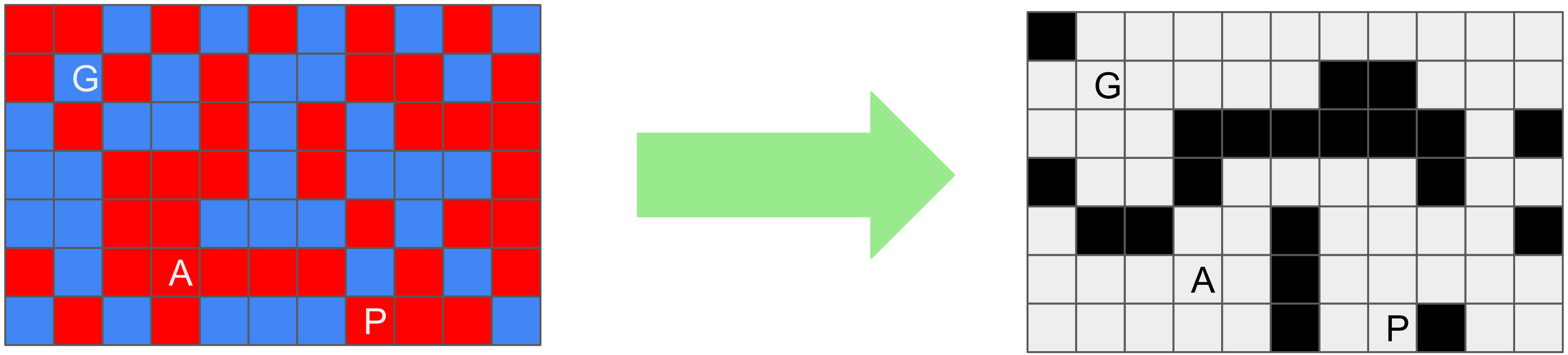
- A normal grid traversal seems hopeless, because the state of the grid changes while you walk, so the number of possible states is exponential.

La La Langton’s Ant

- **Insight:** A cell can only be visited once.
- *Proof:*
  - Returning to a cell always takes even steps,
  - so the required color will be the same as the first visit,
  - but the color will have flipped,
  - so you can’t return!

La La Langton's Ant

- Which cells are reachable?
  - A cell is reachable iff there is a path to it from the starting point with alternating colors (except the first cell).
- So, we can simply look at the grid of "reachable cells":
  - On the right, reachable cells are **white** & nonreachable cells are **black**.



La La Langton's Ant

- This is just a regular grid maze!
- Thus, our problem is now to find a path in a grid that:
  - goes $\mathrm{A} \rightarrow \mathrm{G} \rightarrow \mathrm{P}$ or $\mathrm{A} \rightarrow \mathrm{P} \rightarrow \mathrm{G}$, and
  - that goes through each cell at most once.
- There are only two possibilities, so we can just try both.
  - So we can just look at one, say $\mathrm{A} \rightarrow \mathrm{G} \rightarrow \mathrm{P}$.

La La Langton’s Ant

- Normally, traversing a grid can be done with just BFS or DFS.
- However, since we need to visit two cells, the two paths $\mathrm{A} \rightarrow \mathrm{G}$ and $\mathrm{G} \rightarrow \mathrm{P}$ may overlap.
- We need to limit the number of visits to each cell by 1.
- Limiting the “capacity” of nodes and edges is what maximum flow does best!

La La Langton’s Ant

- Instead of finding two disjoint paths $\mathrm{A} \rightarrow \mathrm{G}$ and $\mathrm{G} \rightarrow \mathrm{P}$, let’s find two disjoint paths $\mathrm{G} \rightarrow \mathrm{A}$ and $\mathrm{G} \rightarrow \mathrm{P}$ instead.
- We can now create a flow network:
  - The nodes are cells, and the edges are between adjacent cells.
  - G is a source, and A and P are sinks.
  - There is a capacity of 1 at *nodes*, not *edges*.
- Now, we want to find a flow of 2! The path $\mathrm{A} \rightarrow \mathrm{G} \rightarrow \mathrm{P}$ can be extracted from this flow.

La La Langton's Ant

- There are standard ways to:
  - Convert a multi-sink network to a single-sink network.
  - Convert node capacities to edge capacities.
- Finally, doing two Ford-Fulkerson augmentations is enough!
  - It just takes $O(rc)$ per augmentation.
- Thus, the running time is $O(2rc) = O(rc)$ overall.
- No need to worry about exceeding the $4rc$ limit: any valid path only takes rc steps anyway.

Problem 10

Pass the Message

No. of accepted solutions: 0

Author: Kevin Charles Atienza

Pass the Message

- If a message m is written in binary, e.g., as $\Sigma\, \mathrm{a_i 2^i}$ where $\mathrm{a_i} \in \{0, 1\}$, then its square is:
  - $\mathrm{m^2} = (\Sigma_\mathrm{i}\, \mathrm{a_i 2^i})^2 = (\Sigma_\mathrm{i}\, \mathrm{a_i 2^i})(\Sigma_\mathrm{j}\, \mathrm{a_j 2^j}) = \Sigma_\mathrm{i}\Sigma_\mathrm{j}\, \mathrm{a_i a_j 2^{i+j}}$.
- Thus, there are “interaction terms” $\mathrm{a_i a_j}$ between bits i and j.
- Note that $\mathrm{a_i}$ depends only on the i’th bit of the nodes, and nothing else.

Pass the Message

- List all messages as $m_1, m_2, \ldots, m_p$ where $p = n(n-1)/2$.
- We want to find $\Sigma_k\, m_k^{\,2}$.
- Write m_i in binary: $m_k = \Sigma_i\, a_{ki} 2^i$. Then:
    - $\Sigma_k\, m_k^{\,2}$
    - $= \Sigma_k\, (\Sigma_i\, a_{ki} 2^i)^2$
    - $= \Sigma_k\, \Sigma_i \Sigma_j\, a_{ki} a_{kj}\, 2^{i+j}$
    - $= \Sigma_i \Sigma_j\, \Sigma_k a_{ki} a_{kj}\, 2^{i+j}$
    - $= \Sigma_i \Sigma_j\, 2^{i+j} \cdot (\Sigma_k a_{ki} a_{kj})$
- Thus, we want to compute $\Sigma_k a_{ki} a_{kj}$ for all pairs of bits (i, j).

Pass the Message

- For each pair (i, j) of bits, a_{ki} and a_{kj} only depends on the i'th and j'th bits of the nodes, respectively.
- Thus, for each node, we only *keep* the two associated bits.
  - In other words, let's assume that only the numbers $0, 1, 2, 3$ are on the nodes.
- We can now compute how many paths with XOR $0, 1, 2, 3$ are there.
- These can all be done with something like DP on the tree, in linear time.
- Message bits reach up to $\lg M$, so it takes $O(n \lg^2 M)$ overall.

Pass the Message

- The naive $O(n \lg^2 M)$ probably won't pass, but there are strides of optimization that can be made:
  - The answer for (i, j) is the same for (j, i), so only do it once.
    - This saves half the work.
  - Bits $> \lg n$ are special: the corresponding bits of the nodes are 0, so they mostly can be done *all at once*.
    - This reduces the running time to $O(n \lg^2 n)$.
  - Precompute the tree traversal information, so you only have to compute the DP every iteration.
    - This improves the constant factor.

Problem 11

House Hunting

No. of accepted solutions: 0

Author: Jatin Yadav

House Hunting

- Given a tree T, you want to choose k nodes such that the maximum pairwise distance is minimized.
- Consider two nodes a and b with the maximum distance, say D, and let c be the midpoint of this path.
    - c could be a *node* or a *midpoint of an edge*.
- All k nodes must be at a distance $\leq D/2$ from c.
    - Otherwise, there will be a pair of points with distance $> D$.
- Converse also true:
    - If there is a vertex or an edge midpoint from which all k nodes are at a distance $\leq D/2$, then the maximum pairwise distance is $\leq D$

House Hunting

- Based on the previous observations, we can consider an *equivalent problem*:
  - Transform tree T to a new tree T' by adding nodes at midpoints of edges.
    - There are $2\mathrm{n}\text{-}1$ nodes in this tree.
    - Each edge of T' connects an edge midpoint of T to one of its endpoints.
  - Let $\mathrm{F}(\mathrm{c}, \mathrm{D})$ be the number of nodes of T that are at a “distance” ≤ D from c in T'
    - where “distance” refers to distance in T'
  - Find the minimum D for which there exists a node c with $\mathrm{F}(\mathrm{c}, \mathrm{D}) \geq \mathrm{k}$

House Hunting

- Precompute in $O(n \log n)$ to allow LCA, and hence distance queries, to be answered in $O(1)$.
- We then perform *centroid decomposition*.
- Then, for each node x in the *centroid decomposition tree* of T', store
  - $\text{prefix}[x][d]$ = number of nodes of T in the subtree of x (in centroid tree) at a distance $\leq d$ from x.
- Similarly, for every child x' of x, store
  - $\text{prefix2}[x'][d]$ = number of nodes of T in the subtree of x' (in centroid tree) at a distance $\leq d$ from x.
- This takes $O(n \log n)$ time since $d \leq$ (size of subtree of x).

House Hunting

- Given prefix and $\mathrm{prefix2}$, $\mathrm{F}(\mathrm{c}, \mathrm{D})$ can be computed in $\mathrm{O}(\log \mathrm{n})$:
  - Iterate over the ancestors of c in the centroid tree.
  - For an ancestor x such that $\mathrm{dist}(\mathrm{x}, \mathrm{c}) = \mathrm{d}$,
    - add $\mathrm{prefix}[\mathrm{x}][\mathrm{D}\text{-}\mathrm{d}]$ - $\mathrm{prefix2}[\mathrm{x}'][\mathrm{D}\text{-}\mathrm{d}]$ to the answer,
    - where x' is the child of x that contains c.
- Binary search on D to find the smallest one for which there exists a node c with $\mathrm{F}(\mathrm{c}, \mathrm{D}) \geq \mathrm{k}$
- This takes $\mathrm{O}(\mathrm{n} \log^2 \mathrm{n})$ time, enough to get AC

House Hunting

- The solution can be improved by using a two-pointers like approach.
- If we know the answer is $\leq \mathrm{D}$, then for each new node c, we can start checking from $\mathrm{F}(\mathrm{c}, \mathrm{D} - 1)$.
- Iterate over c and do:
  - `while(F(c, D - 1) ≥ k) D--`
- In every computation either D decreases by 1 or we move to the next node, hence not wasting more than one call to F for any node c.
- The overall complexity is $\mathrm{O}(\mathrm{n} \log \mathrm{n})$

Problem 12

Wakka and Molly

No. of accepted solutions: 2

Author: Kevin Charles Atienza

Wakka and Molly

- Let's first assume they *cooperate* to clear the whole array.
- When is it possible to clear the whole array?
- It's possible iff the number of visible cells is odd.
- *Proof:*
    - If there’s a connected piece with even visible cells, then there will always be, after every move.
    - But the end state doesn’t have such a connected piece.

Wakka and Molly

- Now, back to the *competition*.
- For Wakka to possibly win, there must be an odd number of visible cells.
- Also, it's clear that Molly wants to make a piece with an even number.
- But Molly's job is quite easy!
  - If there is a piece with more than two visible cells, then smacking the second from the left does the job.
  - Only one move needed to ruin it for Wakka!

Wakka and Molly

- Therefore, Wakka must leave each piece to contain exactly 1 visible cell. (0 is not allowed because it is even)
- But that is also sufficient: If every piece has 1 visible cell, then that will always be true, no matter what *anyone* does.
- Thus, a state is winnable iff you can smack f times to make sure every piece has 1 visible cell.
  - 's' doesn't matter!

Wakka and Molly

- Let $p_0 < p_1 < \ldots < p_{2k}$ be the initial visible cells.
- The odd-indexed ones, $p_1, p_3, p_5, \ldots$ cannot be smacked; however, they must be *flipped* somehow. (Why?)
- So for odd i, p_i is either flipped from the left or right side (or both).
  - i.e., a “wave” of smacks from p_{i-1} or p_{i+1} will flip it.
- Intuitively, it should come from the “closer” one.
- So the minimum number of smacks needed is approx. the sum of the distances to each closest neighbor, for odd i.

Wakka and Molly

- Not completely correct, because a cell can generate two "waves" of smacks going on both sides.
- The condition is a little bit complicated, but they can be managed.
- We can now count the states with this property with DP. Let:
  - $\mathrm{count\_odd}(n, f)$ = number of states with odd visible cells that takes at most f smacks to "fix".
  - $\mathrm{count\_even}(n, f)$ = number of states with even visible cells that takes at most f smacks to "fix", *and* also assumes the cell to the left of the leftmost cell is secretly a visible cells that's already been smacked.

Wakka and Molly

- A mutual recurrence can now be created for $\mathrm{count\_odd}$ or $\mathrm{count\_even}$.
- There are $O(n)$ states, each requiring $O(n^2)$ transition, so overall, it takes $O(n^4)$ time.
- Use more standard techniques e.g.,
  - Store prefix sums in the DP. The transition becomes $O(n)$.
  - Precompute everything at the start.
- It now takes $O({n_{max}}^3)$ time, and each test case is answered in $O(1)$.

Wakka and Molly

- *Alternative approaches:*
  - Other DP states are possible.
  - One can also set up an "automaton" that can recognize valid strings. It will have a state for each "f" and something else to remember the past string with (having $O(n_{max})$ values).
    - Also takes $O(n_{max}{}^3)$ time.
  - A closed-form formula also exists, though it may be a bit tricky to prove correct.
    - It takes $O(n)$ time per query.