Delft Algorithm Programming Contest 2026

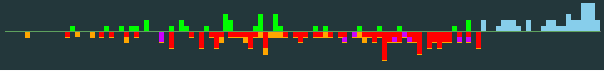
Solutions presentation

The Benelux Algorithm Programming Contest (BAPC) 2026 Jury

September 26, 2026

A: Ambagaata

Problem author: Mike de Vries

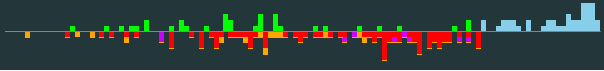


Problem: Given: dictionary with n words, messages where all vowels except maybe `y` are replaced by `a`. At most 4 letters per word are a `y`. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

A: Ambagaata

Problem author: Mike de Vries



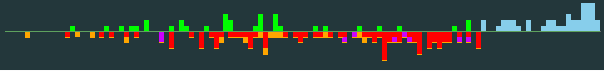
Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

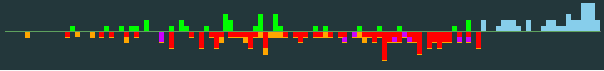
- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

Setup: Create a map from those words to their corresponding dictionary words.

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

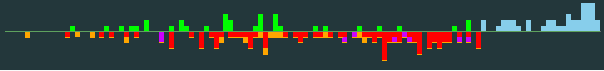
- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

Setup: Create a map from those words to their corresponding dictionary words.
Any message containing a string corresponding to two words is ambiguous or impossible, so keep only one dictionary word (or a note that the string is ambiguous).

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

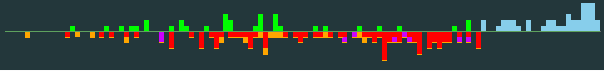
Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

Setup: Create a map from those words to their corresponding dictionary words.
Any message containing a string corresponding to two words is ambiguous or impossible, so keep only one dictionary word (or a note that the string is ambiguous).

Solution: Go over all messages and use the map to determine the translation, or that the message is impossible (if a string in the message is not in the map) or ambiguous.

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

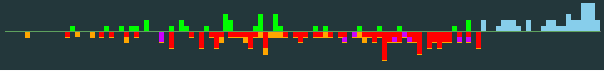
Setup: Create a map from those words to their corresponding dictionary words.
Any message containing a string corresponding to two words is ambiguous or impossible, so keep only one dictionary word (or a note that the string is ambiguous).

Solution: Go over all messages and use the map to determine the translation, or that the message is impossible (if a string in the message is not in the map) or ambiguous.

Pitfall: Messages with an ambiguous word can become impossible due to a later word.

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

Setup: Create a map from those words to their corresponding dictionary words.
Any message containing a string corresponding to two words is ambiguous or impossible, so keep only one dictionary word (or a note that the string is ambiguous).

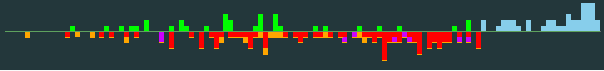
Solution: Go over all messages and use the map to determine the translation, or that the message is impossible (if a string in the message is not in the map) or ambiguous.

Pitfall: Messages with an ambiguous word can become impossible due to a later word.

Running time: $\mathcal{O}(2^Y D + M)$, where D is the total length of words in the dictionary, M is the total length of the messages, and Y is the maximum number of ys in a single word.

A: Ambagaata

Problem author: Mike de Vries



Problem: Given: dictionary with n words, messages where all vowels except maybe y are replaced by a. At most 4 letters per word are a y. Determine for each message whether

- it corresponds to a unique message consisting of words from the dictionary, or
- it is ambiguous (could correspond to multiple messages), or
- it is impossible to construct from the dictionary.

Setup: For each word in the input, replace all vowels with a and if the word contains Y times the letter y, generate all 2^Y possible ways to replace a y with an a or not.

Setup: Create a map from those words to their corresponding dictionary words.
Any message containing a string corresponding to two words is ambiguous or impossible, so keep only one dictionary word (or a note that the string is ambiguous).

Solution: Go over all messages and use the map to determine the translation, or that the message is impossible (if a string in the message is not in the map) or ambiguous.

Pitfall: Messages with an ambiguous word can become impossible due to a later word.

Running time: $\mathcal{O}(2^Y D + M)$, where D is the total length of words in the dictionary, M is the total length of the messages, and Y is the maximum number of ys in a single word.

Statistics: 176 submissions, 33 accepted, 43 unknown

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Problem: Given numbers x and o, construct a valid, reachable game state in *Tic-tac-toe*, with x crosses and o circles.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Problem: Given numbers x and o, construct a valid, reachable game state in *Tic-tac-toe*, with x crosses and o circles.

Condition 1: The board is 3×3, so at most 9 moves can be played, so it must hold that $x + o \leq 9$.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Problem: Given numbers x and o, construct a valid, reachable game state in *Tic-tac-toe*, with x crosses and o circles.

Condition 1: The board is 3×3, so at most 9 moves can be played, so it must hold that $x + o \leq 9$.

Condition 2: Players alternate turns, so the number of crosses and circles can differ by at most 1, so $|x - o| \leq 1$.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Problem: Given numbers x and o, construct a valid, reachable game state in *Tic-tac-toe*, with x crosses and o circles.

Condition 1: The board is 3×3, so at most 9 moves can be played, so it must hold that $x + o \leq 9$.

Condition 2: Players alternate turns, so the number of crosses and circles can differ by at most 1, so $|x - o| \leq 1$.

Observation: If no player has won yet, and Conditions 1 & 2 hold, it must be reachable.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

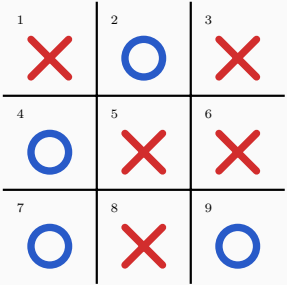
Problem: Given numbers x and o, construct a valid, reachable game state in *Tic-tac-toe*, with x crosses and o circles.

Condition 1: The board is 3×3, so at most 9 moves can be played, so it must hold that $x + o \leq 9$.

Condition 2: Players alternate turns, so the number of crosses and circles can differ by at most 1, so $|x - o| \leq 1$.

Observation: If no player has won yet, and Conditions 1 & 2 hold, it must be reachable.

Construction for draw:



B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

Running time: $\mathcal{O}(x + o)$ (basically, $\mathcal{O}(1)$).

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

Running time: $\mathcal{O}(x + o)$ (basically, $\mathcal{O}(1)$).

Fun fact: The secret test cases covered every single possible input.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

Running time: $\mathcal{O}(x + o)$ (basically, $\mathcal{O}(1)$).

Fun fact: The secret test cases covered every single possible input.

Fun fact 2: Within the time limit, your solution can play all possible games of *Tic-tac-toe*.

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

Running time: $\mathcal{O}(x + o)$ (basically, $\mathcal{O}(1)$).

Fun fact: The secret test cases covered every single possible input.

Fun fact 2: Within the time limit, your solution can play all possible games of *Tic-tac-toe*.

Bonus: What if instead of a 3×3 board, the board was $n \times n$?
(See also: "Crosses and Circles" from GCPC 2026)

B: Butter, Cheese, and Eggs

Problem author: Jeroen Op de Beek

Solution: Take the construction for a draw, alternate removing crosses and circles, until left with the correct total.

Edge case: At the end, may need to replace all circles for crosses and vice versa.

Running time: $\mathcal{O}(x + o)$ (basically, $\mathcal{O}(1)$).

Fun fact: The secret test cases covered every single possible input.

Fun fact 2: Within the time limit, your solution can play all possible games of *Tic-tac-toe*.

Bonus: What if instead of a 3×3 board, the board was $n \times n$?
(See also: "Crosses and Circles" from GCPC 2026)

Statistics: 99 submissions, 68 accepted

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Problem: Given an undirected graph. Each edge ij has a probability p_{ij} that it can be used during each second, and crossing an edge takes exactly one second. What is the minimum expected time to go from s to t?

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Problem: Given an undirected graph. Each edge ij has a probability p_{ij} that it can be used during each second, and crossing an edge takes exactly one second. What is the minimum expected time to go from s to t?

Strategy: Each vertex i has some minimum expected time E_i to go from i to B. For t, we have $E_t = 0$. For any other vertex i, each second, we only cross an edge if we can move to a vertex j with $E_j < E_i$, and if we can, then we pick the smallest E_j.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Problem: Given an undirected graph. Each edge ij has a probability p_{ij} that it can be used during each second, and crossing an edge takes exactly one second. What is the minimum expected time to go from s to t?

Strategy: Each vertex i has some minimum expected time E_i to go from i to B. For t, we have $E_t = 0$. For any other vertex i, each second, we only cross an edge if we can move to a vertex j with $E_j < E_i$, and if we can, then we pick the smallest E_j.

Algebra: For example, if vertex 4 neighbors vertices $1, 2, 3$ with $E_1 < E_2 < E_3 < E_4$, we get

$$E_4 = 1 + p_{41}E_1 + (1 - p_{41})p_{42}E_2 + (1 - p_{41})(1 - p_{42})p_{43}E_3 + (1 - p_{41})(1 - p_{42})(1 - p_{43})E_4.$$

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Problem: Given an undirected graph. Each edge ij has a probability p_{ij} that it can be used during each second, and crossing an edge takes exactly one second. What is the minimum expected time to go from s to t?

Strategy: Each vertex i has some minimum expected time E_i to go from i to B. For t, we have $E_t = 0$. For any other vertex i, each second, we only cross an edge if we can move to a vertex j with $E_j < E_i$, and if we can, then we pick the smallest E_j.

Algebra: For example, if vertex 4 neighbors vertices $1, 2, 3$ with $E_1 < E_2 < E_3 < E_4$, we get

$$E_4 = 1 + p_{41}E_1 + (1 - p_{41})p_{42}E_2 + (1 - p_{41})(1 - p_{42})p_{43}E_3 + (1 - p_{41})(1 - p_{42})(1 - p_{43})E_4.$$

If we write this as $E_4 = A_4 + B_4E_4$, then this solves to $E_4 = \frac{A_4}{1 - B_4}$.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Problem: Given an undirected graph. Each edge ij has a probability p_{ij} that it can be used during each second, and crossing an edge takes exactly one second. What is the minimum expected time to go from s to t?

Strategy: Each vertex i has some minimum expected time E_i to go from i to B. For t, we have $E_t = 0$. For any other vertex i, each second, we only cross an edge if we can move to a vertex j with $E_j < E_i$, and if we can, then we pick the smallest E_j.

Algebra: For example, if vertex 4 neighbors vertices $1, 2, 3$ with $E_1 < E_2 < E_3 < E_4$, we get

$$E_4 = 1 + p_{41}E_1 + (1 - p_{41})p_{42}E_2 + (1 - p_{41})(1 - p_{42})p_{43}E_3 + (1 - p_{41})(1 - p_{42})(1 - p_{43})E_4.$$

If we write this as $E_4 = A_4 + B_4E_4$, then this solves to $E_4 = \frac{A_4}{1 - B_4}$.

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_ip_{ij}E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1 - B_i}$. If $B_i = 1$, then t cannot be reached.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

Algorithm: Initialize $A_i = 1$ and $B_i = 1$ for each $i \neq t$. Keep a priority queue of possible valuations of the values E_i, starting with $E_t = 0$.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

Algorithm: Initialize $A_i = 1$ and $B_i = 1$ for each $i \neq t$. Keep a priority queue of possible valuations of the values E_i, starting with $E_t = 0$.

Repeatedly take the lowest possible valuation $E_i = x$ from the heap and assign the value if E_i wasn't already assigned. Update all neighbors j, increasing A_j by $B_j p_{ij} E_i$ and multiplying B_j by $1 - p_{ij}$, and adding the possible valuation $E_j = \frac{A_j}{1-B_j}$.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

Algorithm: Initialize $A_i = 1$ and $B_i = 1$ for each $i \neq t$. Keep a priority queue of possible valuations of the values E_i, starting with $E_t = 0$.

Repeatedly take the lowest possible valuation $E_i = x$ from the heap and assign the value if E_i wasn't already assigned. Update all neighbors j, increasing A_j by $B_j p_{ij} E_i$ and multiplying B_j by $1 - p_{ij}$, and adding the possible valuation $E_j = \frac{A_j}{1-B_j}$.

When the queue is empty, we either found E_s, or there is no path from s to t.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

Algorithm: Initialize $A_i = 1$ and $B_i = 1$ for each $i \neq t$. Keep a priority queue of possible valuations of the values E_i, starting with $E_t = 0$.

Repeatedly take the lowest possible valuation $E_i = x$ from the heap and assign the value if E_i wasn't already assigned. Update all neighbors j, increasing A_j by $B_j p_{ij} E_i$ and multiplying B_j by $1 - p_{ij}$, and adding the possible valuation $E_j = \frac{A_j}{1-B_j}$.

When the queue is empty, we either found E_s, or there is no path from s to t.

Running time: $\mathcal{O}((n+m)\log(n+m))$ with a binary heap.

C: Chaotic Crossing

Problem author: Reinier Schmiermann

Calculation: To calculate A_i and B_i, we initialize $A_i = 1$ and $B_i = 1$. We then iterate the neighbors j with $E_j < E_i$ in increasing order of E_j, and we increase A_i by $B_i p_{ij} E_j$ and multiply B_i by $1 - p_{ij}$. Then $E_i = \frac{A_i}{1-B_i}$. If $B_i = 1$, then t cannot be reached.

Issue: The values of E_i depend on eachother, so where do we start?

Observation: The value E_i only depends on the values E_j of vertices j with $E_j < E_i$. So we can start at $E_t = 0$, and determine the rest of the values E_i in increasing order of E_i.

Algorithm: Initialize $A_i = 1$ and $B_i = 1$ for each $i \neq t$. Keep a priority queue of possible valuations of the values E_i, starting with $E_t = 0$.

Repeatedly take the lowest possible valuation $E_i = x$ from the heap and assign the value if E_i wasn't already assigned. Update all neighbors j, increasing A_j by $B_j p_{ij} E_i$ and multiplying B_j by $1 - p_{ij}$, and adding the possible valuation $E_j = \frac{A_j}{1-B_j}$.

When the queue is empty, we either found E_s, or there is no path from s to t.

Running time: $\mathcal{O}((n + m)\log(n + m))$ with a binary heap.

Statistics: 7 submissions, 1 accepted, 5 unknown

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

Solution part 1: Dynamic programming for each die separately. For each die, calculate each value of $P(i, j)$: The probability of landing on space i after j turns.

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

Solution part 1: Dynamic programming for each die separately. For each die, calculate each value of $P(i, j)$: The probability of landing on space i after j turns.

Recursion: For each i, j with j from small to large, starting on space i, determine for each possible die roll the landing space k, and add $\frac{1}{6}P(i, j)$ to $P(k, j + 1)$.

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

Solution part 1: Dynamic programming for each die separately. For each die, calculate each value of $P(i, j)$: The probability of landing on space i after j turns.

Recursion: For each i, j with j from small to large, starting on space i, determine for each possible die roll the landing space k, and add $\frac{1}{6}P(i, j)$ to $P(k, j + 1)$.

Solution part 2: For each die independently, calculate the probability of winning (with tie-break rules). For each die, determine after each turn the probability of finishing on that turn with no other player having finished yet, and after t turns, determine for each space the probability of landing there ahead of all other players.

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

Solution part 1: Dynamic programming for each die separately. For each die, calculate each value of $P(i, j)$: The probability of landing on space i after j turns.

Recursion: For each i, j with j from small to large, starting on space i, determine for each possible die roll the landing space k, and add $\frac{1}{6}P(i, j)$ to $P(k, j + 1)$.

Solution part 2: For each die independently, calculate the probability of winning (with tie-break rules). For each die, determine after each turn the probability of finishing on that turn with no other player having finished yet, and after t turns, determine for each space the probability of landing there ahead of all other players.

Running time: $\mathcal{O}(tn)$ (times #players and #sides-per-die, which are constant).

D: Dicey Decision

Problem author: Freek Henstra

Problem: Given a *Snakes and Ladders*-type board with n spaces, m shortcuts, turn limit t, and four six-sided dice to choose from. If you can pick your die first and you start playing last, what is your maximum possible probability of winning?

Naive solution: Dynamic programming on the state space. There are $\mathcal{O}(tn^4)$ possible states, which is too many. (State is number of turns to go and each player's position.)

Observation: The players don't interact with one another, so we can consider each player separately.

Solution part 1: Dynamic programming for each die separately. For each die, calculate each value of $P(i, j)$: The probability of landing on space i after j turns.

Recursion: For each i, j with j from small to large, starting on space i, determine for each possible die roll the landing space k, and add $\frac{1}{6}P(i, j)$ to $P(k, j + 1)$.

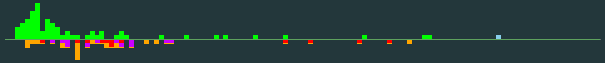
Solution part 2: For each die independently, calculate the probability of winning (with tie-break rules). For each die, determine after each turn the probability of finishing on that turn with no other player having finished yet, and after t turns, determine for each space the probability of landing there ahead of all other players.

Running time: $\mathcal{O}(tn)$ (times #players and #sides-per-die, which are constant).

Statistics: 49 submissions, 3 accepted, 38 unknown

E: Exhausted Economist

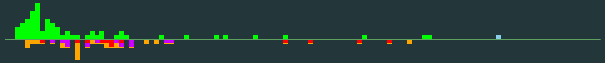
Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

E: Exhausted Economist

Problem author: Mike de Vries



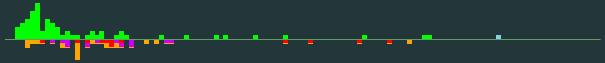
Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

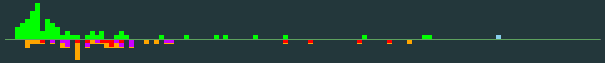
Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

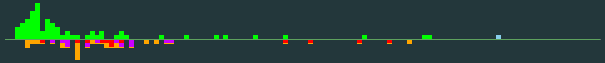
- Otherwise, you can increase the price without losing buyers.

Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

Solution: Sort all prices from low to high.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

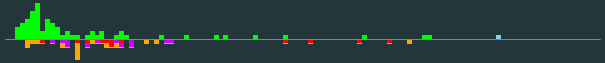
Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

Solution: Sort all prices from low to high.

Asking p_i gives revenue $i \cdot p_i$, where p_i is the i-th price in the sorted list.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

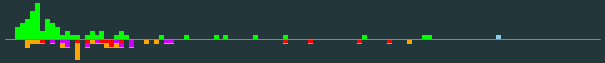
Solution: Sort all prices from low to high.

Asking p_i gives revenue $i \cdot p_i$, where p_i is the i-th price in the sorted list.

The answer is the maximum of all these values.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

Solution: Sort all prices from low to high.

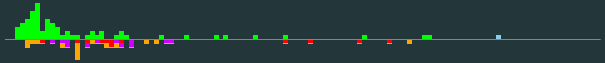
Asking p_i gives revenue $i \cdot p_i$, where p_i is the i-th price in the sorted list.

The answer is the maximum of all these values.

Running time: $\mathcal{O}(n \log n)$.

E: Exhausted Economist

Problem author: Mike de Vries



Problem: Given a list of prices that each person is willing to pay for your product, find your maximal revenue.

Observation: You should always ask a price in the list.

- Otherwise, you can increase the price without losing buyers.

Naive solution: Computing the number of buyers for each price takes $\mathcal{O}(n^2)$ time, which is too slow.

Solution: Sort all prices from low to high.

Asking p_i gives revenue $i \cdot p_i$, where p_i is the i-th price in the sorted list.

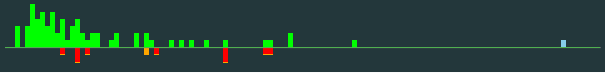
The answer is the maximum of all these values.

Running time: $\mathcal{O}(n \log n)$.

Statistics: 106 submissions, 66 accepted, 1 unknown

F: Football Forecast

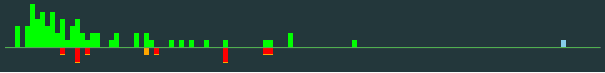
Problem author: Reinier Schmiermann



Problem: Forecast the winner of the World Cup, based on the quality score of each team.

F: Football Forecast

Problem author: Reinier Schmiermann

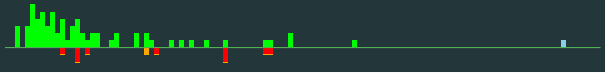


Problem: Forecast the winner of the World Cup, based on the quality score of each team.

Group phase: The top two teams in each group will continue to the elimination phase.

F: Football Forecast

Problem author: Reinier Schmiermann



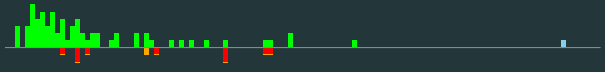
Problem: Forecast the winner of the World Cup, based on the quality score of each team.

Group phase: The top two teams in each group will continue to the elimination phase.

Elimination phase: In each match, the team with the highest quality score advances.

F: Football Forecast

Problem author: Reinier Schmiermann



Problem: Forecast the winner of the World Cup, based on the quality score of each team.

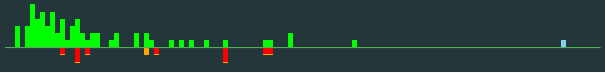
Group phase: The top two teams in each group will continue to the elimination phase.

Elimination phase: In each match, the team with the highest quality score advances.

Winner: It can be proven that the World Cup winner is the team with the highest quality score.

F: Football Forecast

Problem author: Reinier Schmiermann



Problem: Forecast the winner of the World Cup, based on the quality score of each team.

Group phase: The top two teams in each group will continue to the elimination phase.

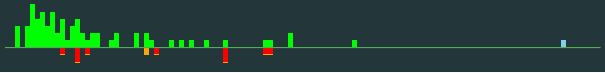
Elimination phase: In each match, the team with the highest quality score advances.

Winner: It can be proven that the World Cup winner is the team with the highest quality score.

Solution: Output the name of the team with the highest quality score.
(But the time limit is lenient enough to simulate the entire tournament.)

F: Football Forecast

Problem author: Reinier Schmiermann



Problem: Forecast the winner of the World Cup, based on the quality score of each team.

Group phase: The top two teams in each group will continue to the elimination phase.

Elimination phase: In each match, the team with the highest quality score advances.

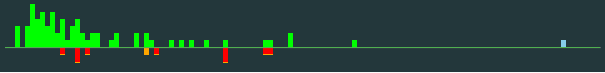
Winner: It can be proven that the World Cup winner is the team with the highest quality score.

Solution: Output the name of the team with the highest quality score.
(But the time limit is lenient enough to simulate the entire tournament.)

Running time: $\mathcal{O}(n)$.

F: Football Forecast

Problem author: Reinier Schmiermann



Problem: Forecast the winner of the World Cup, based on the quality score of each team.

Group phase: The top two teams in each group will continue to the elimination phase.

Elimination phase: In each match, the team with the highest quality score advances.

Winner: It can be proven that the World Cup winner is the team with the highest quality score.

Solution: Output the name of the team with the highest quality score.
(But the time limit is lenient enough to simulate the entire tournament.)

Running time: $\mathcal{O}(n)$.

Statistics: 79 submissions, 68 accepted, 1 unknown

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Proof: The opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that this path is removed, so nothing is gained.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Strategy: It follows that such paths only come into play once there are no other options remaining. We can thus use the parity of the total number of goods to determine which player has to tank the net loss of each such path, and we can then remove all such paths from the game.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Observation 1: Maximizing our own total is equivalent to maximizing the signed difference between our own total and the opponent's total, because the combined total is fixed. We may thus assume that we lose value whenever the opponent grabs a good.

Simpler case: Consider if each rooted tree is a path.

Observation 2: Consider a path of two vertices with the second value larger than the first. Then it is never advantageous to grab the first good.

Strategy: It follows that such paths only come into play once there are no other options remaining. We can thus use the parity of the total number of goods to determine which player has to tank the net loss of each such path, and we can then remove all such paths from the game.

Furthermore, we can do the same whenever the last value of a path is larger than the previous, since those two goods will eventually reduce to a path of length two.

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

We can furthermore do the same whenever any value besides the last value of a path is larger than the previous, since the path will eventually reduce for that value to become the second value of the path.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observation 2: We may assume that the last value of a path is never larger than the previous.

Observation 3: Consider a path with first three values a, b, c with $b > a$. Then it can only be advantageous to grab the first good if the intention is for the opponent to grab the second good so that you can grab the third good afterwards.

Proof: Again, the opponent can reply by grabbing the second good. As a result, it is again your turn, and besides the net loss you just made, the only change to the game is that you can now grab the third good, so that is the only possible advantage.

Strategy: We may thus combine the first three goods into one good with value $a - b + c$.

We can furthermore do the same whenever any value besides the last value of a path is larger than the previous, since the path will eventually reduce for that value to become the second value of the path.

Combined with Observation 2, we may thus assume that no value in any path is larger than the previous!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

Strategy: It is now always possible to grab the good with the highest value, so we can apply the greedy strategy!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Simpler case: Consider if each rooted tree is a path.

Observations 2 & 3: We may assume that no value in any path is larger than the previous.

Strategy: It is now always possible to grab the good with the highest value, so we can apply the greedy strategy!

Observation 4: Since there is never a reason to grab anything other than the good with the highest value, we can combine all the paths into a single path by sorting all the values from largest to smallest!

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Even if this path is no longer sorted from largest to smallest, it can again be reduced until it is.

Observation 4: Any collection of paths reduces to a single path with its values sorted from largest to smallest.

Next case: Consider a rooted tree for which the root is the only vertex with multiple children.

Strategy: Grabbing the good at the root reduces the tree to a collection of paths.

This collection of paths can be reduced to a single path with its values sorted from largest to smallest.

The entire tree thus reduces to the same path, but with the root added to the front.

Even if this path is no longer sorted from largest to smallest, it can again be reduced until it is.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest!

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Finally, while the first value of the path is smaller than the second, apply Observations 2 and 3 to reduce the path.

Observation 5: Any rooted tree can recursively be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Base case: A leaf with value a reduces to a path with a singular value a.

Recursion: For a non-leaf vertex with value a, first calculate for each child how its subtree reduces to a single path with its values sorted from largest to smallest.

Then merge these paths into one by sorting all combined values from largest to smallest, and add the value a to the front of the path.

Finally, while the first value of the path is smaller than the second, apply Observations 2 and 3 to reduce the path.

Running time: Since every value may need to get sorted n times, the total running time is $\Omega(n^2 \log n)$, which is too slow!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

Amortized, we thus have to move a value to a larger heap at most $n \log_2 n$ times.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a binary heap data structure, and apply small-to-large merging to the heaps.

Running time: Ignoring the fact that the heaps get smaller due to the reduction step, values can only move into a larger heap at most $\log_2 n$ times.

The fact that the heaps get smaller due to the reduction step only reduces the number of times a value moves into a larger heap.

Amortized, we thus have to move a value to a larger heap at most $n \log_2 n$ times.

Since heap operations cost $\mathcal{O}(\log n)$ time, the total running time is $\mathcal{O}(n(\log n)^2)$, which is fast enough!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

Combined with small-to-large merging, this results in a total running time of $\mathcal{O}(n \log n)$!

G: Grabbing Goods

Problem author: Jippe Hoogeveen, Mike de Vries

Algorithm: Recursively calculate for each subtree how it reduces to a single path with its values sorted from largest to smallest.

Speed-up: Store each sorted path in a heap data structure, and apply small-to-large merging to the heaps.

Running time: $\mathcal{O}(n(\log n)^2)$.

Even faster: With a custom heap implementation, you can insert all k elements of a heap of size k into a larger heap of size n in $\mathcal{O}(n \log(n/k))$ time.

Combined with small-to-large merging, this results in a total running time of $\mathcal{O}(n \log n)$!

However, in practice, the difference turns out to be immeasurable. In fact, only reading the input already already takes up half the time of the $\mathcal{O}(n(\log n)^2)$ solution!

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Running time: Must use heaps and small-to-large merging to obtain $\mathcal{O}(n \log^2 n)$.

Short version

Problem: Given a rooted forest, two players alternate grabbing the good at one of the roots, causing the rest of its tree to split into multiple new rooted trees. Going first, and with both players playing optimally to maximize the sum of values of their grabbed goods, how much total value will you accumulate?

Main idea: It can be shown that any rooted tree can be reduced to a single path with its values sorted from largest to smallest.

Algorithm: Recursively determine for each subtree how it reduces to such a path. Combine the immediate subtrees by sorting all their reduced values. Then add the root value to the front. So long as the front value is smaller than the next, we can apply certain reductions, until the path values are in a sorted order again.
With at least three values a, b, c with $b > a$, you can reduce those three values to a single value of $a - b + c$. A special case is needed for two values.

Running time: Must use heaps and small-to-large merging to obtain $\mathcal{O}(n \log^2 n)$.

Statistics: 4 submissions, 0 accepted, 2 unknown

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

Solution: First determine for each person and each of the two directions the optimal first handover, and then apply binary lifting on this.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

Solution: First determine for each person and each of the two directions the optimal first handover, and then apply binary lifting on this.

Todo: How do we determine the optimal first handovers? And how do we apply (and what even is) binary lifting?

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

Finally, reverse the array and repeat the process to get the optimal first handovers in the opposite directions.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

Finally, reverse the array and repeat the process to get the optimal first handovers in the opposite directions.

Performance: This step takes $\mathcal{O}(n \log n)$ time.

Step 2: Binary lifting.

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

Running time: The complete algorithm requires $\mathcal{O}((n + q) \log n)$ time and $\mathcal{O}(n \log n)$ memory.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

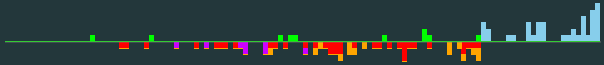
Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

Running time: The complete algorithm requires $\mathcal{O}((n + q) \log n)$ time and $\mathcal{O}(n \log n)$ memory.

Statistics: 24 submissions, 0 accepted, 17 unknown

I: Irritating Interruption I

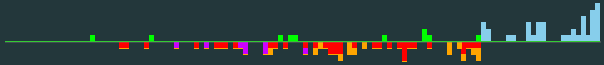
Problem author: Jonas van der Schaaf



Problem: Given a set of arrows on an $n \times m$ grid, calculate how many can be removed. An arrow can be removed when it points at nothing.

I: Irritating Interruption I

Problem author: Jonas van der Schaaf

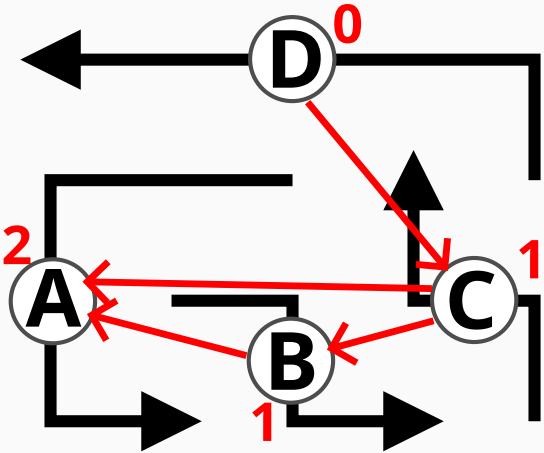


Problem: Given a set of arrows on an $n \times m$ grid, calculate how many can be removed. An arrow can be removed when it points at nothing.

Naive solution: Compute the dependency graph, then prune.

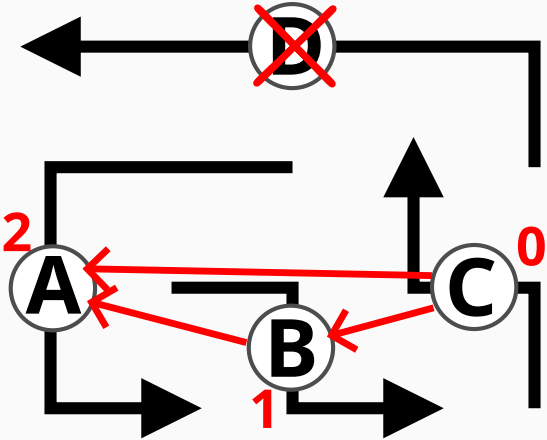
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



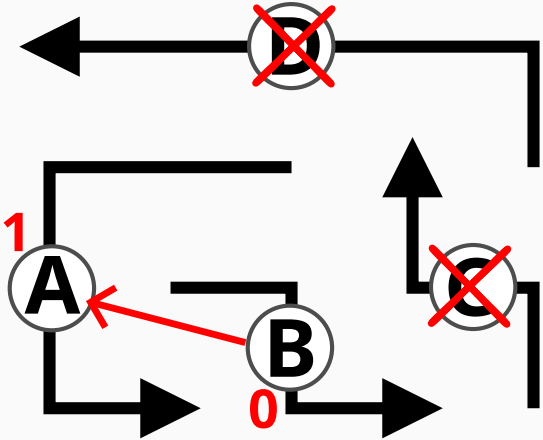
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



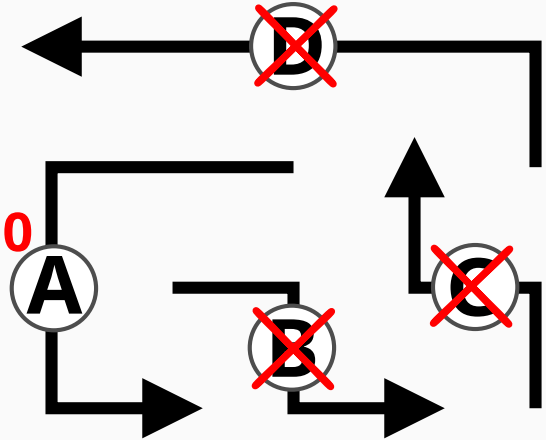
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



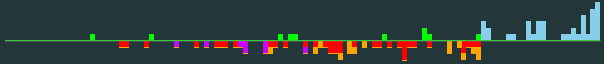
I: Irritating Interruption I

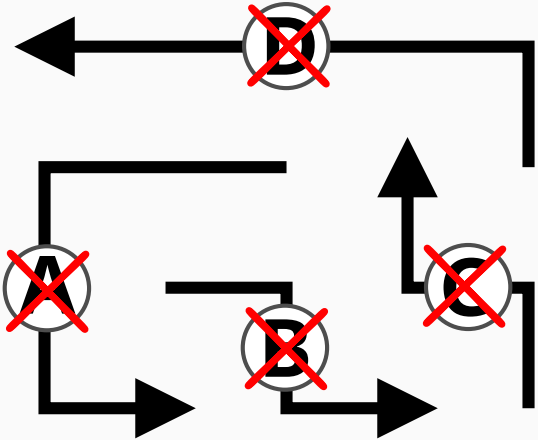
Problem author: Jonas van der Schaaf



I: Irritating Interruption I

Problem author: Jonas van der Schaaf

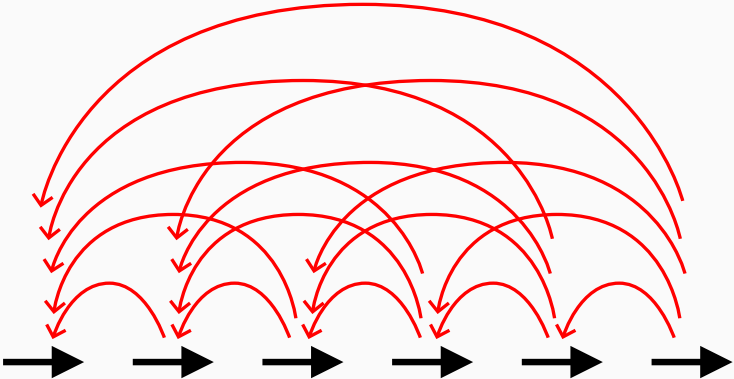




I: Irritating Interruption I

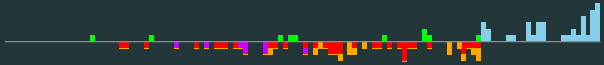
Problem author: Jonas van der Schaaf

Problem: For a arrows, there may be $\mathcal{O}(a^2)$ dependencies. This leads to a running time of $\mathcal{O}(wh(w+h))$, which is too slow.



I: Irritating Interruption I

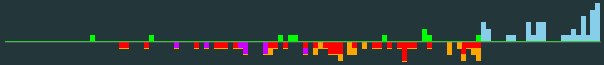
Problem author: Jonas van der Schaaf



Observation: If A points at B and C, and B points at C, then the fact that A points at C is redundant.

I: Irritating Interruption I

Problem author: Jonas van der Schaaf

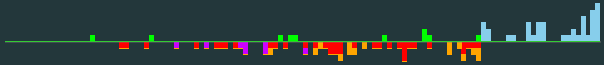


Observation: If A points at B and C, and B points at C, then the fact that A points at C is redundant.

Observation: If you start looking from an arrowhead and encounter another arrowhead pointing in the same direction, any arrows beyond this arrowhead will block both.

I: Irritating Interruption I

Problem author: Jonas van der Schaaf



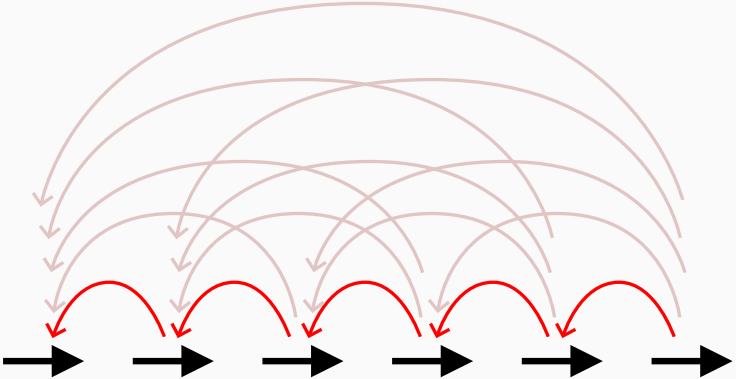
Observation: If A points at B and C, and B points at C, then the fact that A points at C is redundant.

Observation: If you start looking from an arrowhead and encounter another arrowhead pointing in the same direction, any arrows beyond this arrowhead will block both.

Fast solution: Stop looking when you encounter an arrowhead in the same direction, since all further dependencies you might find would be redundant.

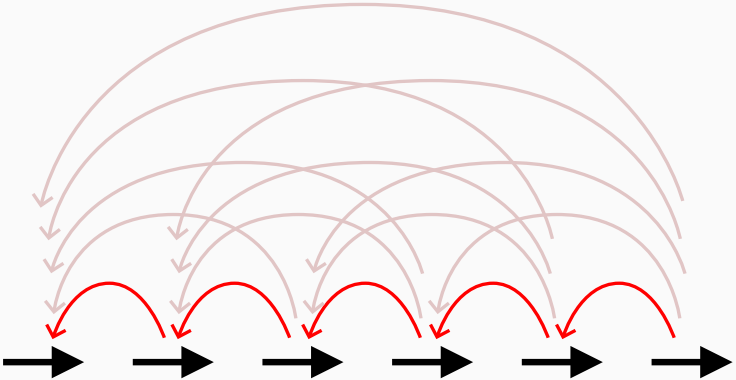
I: Irritating Interruption I

Problem author: Jonas van der Schaaf

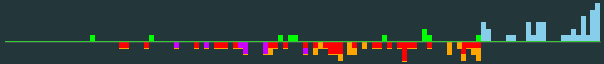


I: Irritating Interruption I

Problem author: Jonas van der Schaaf



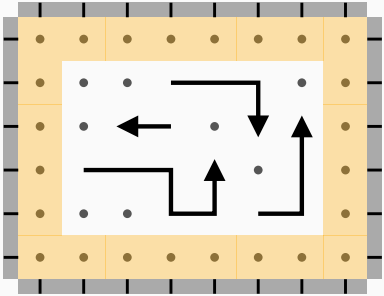
Running time: $\mathcal{O}(wh)$.



Alternative solution: Move edge bounds and remove arrows pointing outwards.

I: Irritating Interruption I

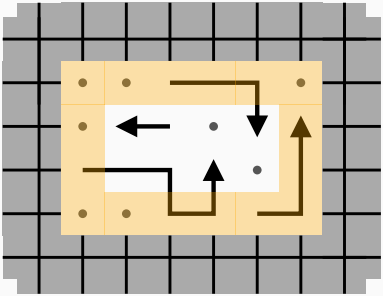
Problem author: Jonas van der Schaaf



All borders start at the edges, and cells next to the edges are marked to be updated.

I: Irritating Interruption I

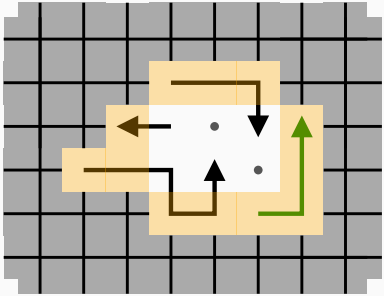
Problem author: Jonas van der Schaaf



Boundaries from all sides are moved inwards as empty cells next to the boundaries are updated.

I: Irritating Interruption I

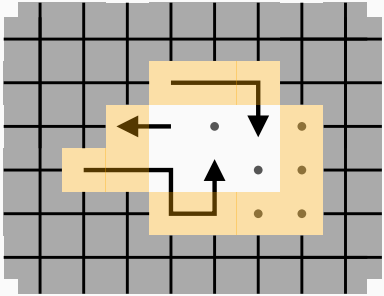
Problem author: Jonas van der Schaaf



An outwards facing arrow is found.

I: Irritating Interruption I

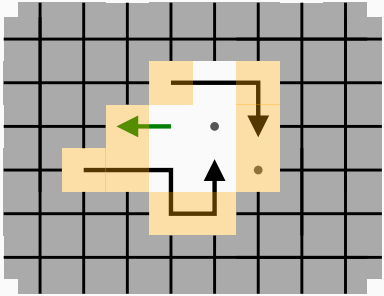
Problem author: Jonas van der Schaaf



The arrow is removed, and the empty spaces are marked for updating.

I: Irritating Interruption I

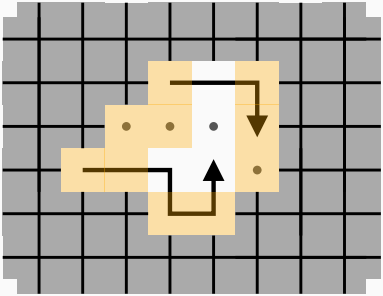
Problem author: Jonas van der Schaaf



Another free arrow is found.

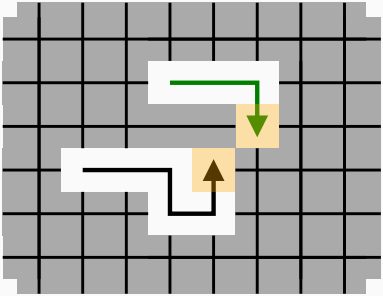
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



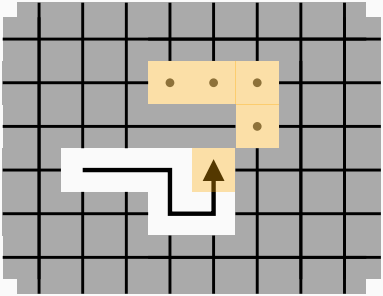
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



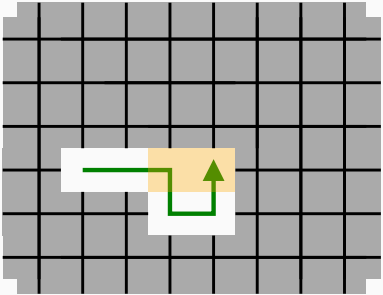
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



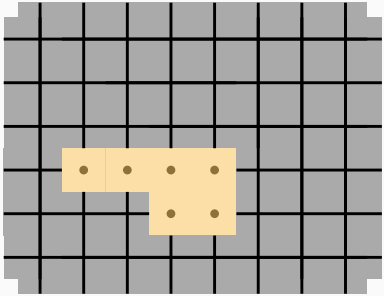
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



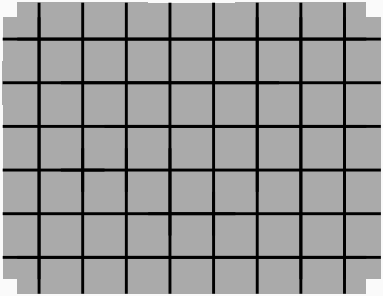
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



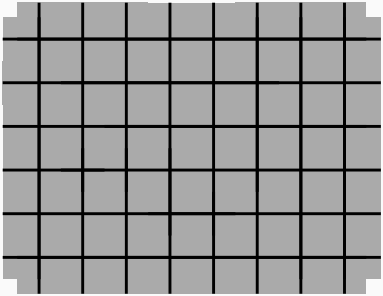
I: Irritating Interruption I

Problem author: Jonas van der Schaaf



I: Irritating Interruption I

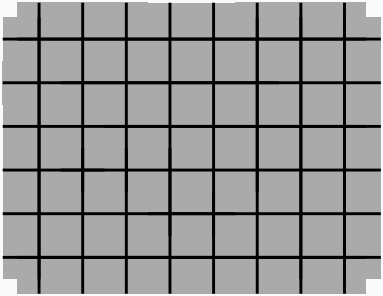
Problem author: Jonas van der Schaaf



Running time: Still $\mathcal{O}(wh)$.

I: Irritating Interruption I

Problem author: Jonas van der Schaaf



Running time: Still $\mathcal{O}(wh)$.

Statistics: 107 submissions, 10 accepted, 38 unknown

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

Equivalent problem: Find the minimum number of edges to remove to break all cycles in a connected undirected graph.

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

Equivalent problem: Find the minimum number of edges to remove to break all cycles in a connected undirected graph.

Observation: A connected graph without cycles is a tree.

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

Equivalent problem: Find the minimum number of edges to remove to break all cycles in a connected undirected graph.

Observation: A connected graph without cycles is a tree.

Solution: A tree has $n - 1$ edges, so the answer is $m - (n - 1)$.

The edges we would remove are the edges that are not part of a fixed spanning tree.

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

Equivalent problem: Find the minimum number of edges to remove to break all cycles in a connected undirected graph.

Observation: A connected graph without cycles is a tree.

Solution: A tree has $n - 1$ edges, so the answer is $m - (n - 1)$.

The edges we would remove are the edges that are not part of a fixed spanning tree.

Running time: We can stop reading the input once we know n and m, so $\mathcal{O}(1)$.

J: Joyful Jogging

Problem author: Tobias Röhr

Problem: Find the minimum number of edges to cover all cycles in a connected undirected graph.

Equivalent problem: Find the minimum number of edges to remove to break all cycles in a connected undirected graph.

Observation: A connected graph without cycles is a tree.

Solution: A tree has $n - 1$ edges, so the answer is $m - (n - 1)$.

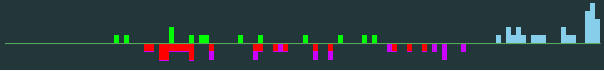
The edges we would remove are the edges that are not part of a fixed spanning tree.

Running time: We can stop reading the input once we know n and m, so $\mathcal{O}(1)$.

Statistics: 43 submissions, 23 accepted, 11 unknown

K: Khoikhoi Dralinpome

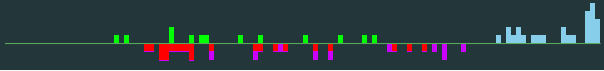
Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries

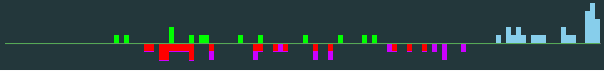


Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



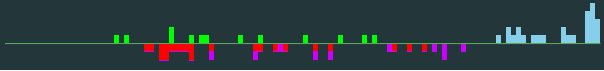
Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

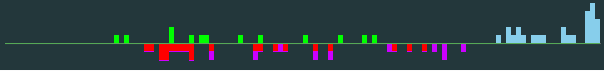
Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

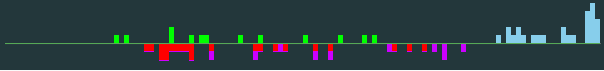
Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

Naive solution: Iterating over all 2^n subsets of nouns is too slow.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

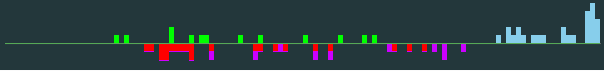
Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

Naive solution: Iterating over all 2^n subsets of nouns is too slow.

Observation 3: We can use a meet-in-the-middle approach!

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

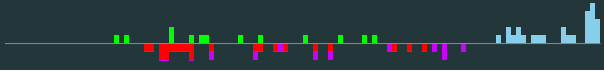
Naive solution: Iterating over all 2^n subsets of nouns is too slow.

Observation 3: We can use a meet-in-the-middle approach!

Solution: Compute the XORed letter-parity mask for each possible subset of the first and last $n/2$ nouns separately. On the second half, XOR with 0 and $1, 2, 4, \ldots, 2^{25}$, and see if any were found on the first pass. If so, joining these subsets forms a dralinpome. This finds every possible dralinpome; return the longest one.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

Naive solution: Iterating over all 2^n subsets of nouns is too slow.

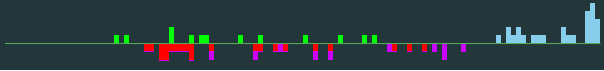
Observation 3: We can use a meet-in-the-middle approach!

Solution: Compute the XORed letter-parity mask for each possible subset of the first and last $n/2$ nouns separately. On the second half, XOR with 0 and $1, 2, 4, \ldots, 2^{25}$, and see if any were found on the first pass. If so, joining these subsets forms a dralinpome. This finds every possible dralinpome; return the longest one.

Running time: $\mathcal{O}((n + 26)2^{n/2})$.

K: Khoikhoi Dralinpome

Problem author: Mike de Vries



Problem: Given a list of nouns, determine the largest dralinpome that can be made through concatenation of a subset.

Observation 1: Dralinpome $\iff$ at most one character has an odd frequency.

Observation 2: The specific nouns only matter for the output. Besides that, all that matters is the length of a noun and which letters have odd frequency in that noun.

Idea: Precompute a 26-bit letter-parity mask for each noun. We can efficiently determine whether a subset of words forms a dralinpome by XORing these masks.

Naive solution: Iterating over all 2^n subsets of nouns is too slow.

Observation 3: We can use a meet-in-the-middle approach!

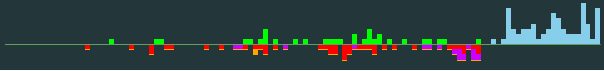
Solution: Compute the XORed letter-parity mask for each possible subset of the first and last $n/2$ nouns separately. On the second half, XOR with 0 and $1, 2, 4, \ldots, 2^{25}$, and see if any were found on the first pass. If so, joining these subsets forms a dralinpome. This finds every possible dralinpome; return the longest one.

Running time: $\mathcal{O}((n + 26)2^{n/2})$.

Statistics: 72 submissions, 13 accepted, 26 unknown

L: Lock In

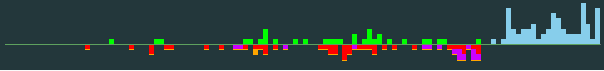
Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

L: Lock In

Problem author: Mike de Vries

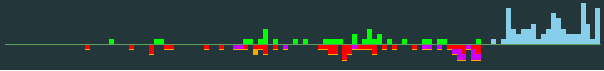


Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

L: Lock In

Problem author: Mike de Vries



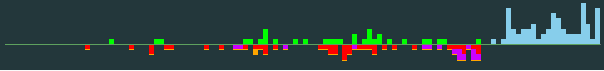
Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

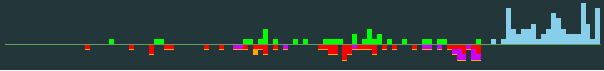
Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

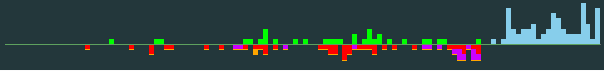
Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

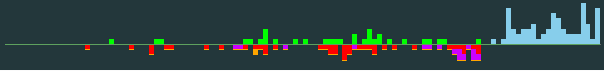
If we need to place a single number, the game is always winnable.

For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

Pre-compute the optimal strategy and follow it for each game.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

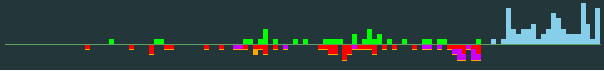
For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

Pre-compute the optimal strategy and follow it for each game.

The resulting probability of winning is $\frac{374337547685188753}{15703912738882368000} \approx 0.023837$. The resulting probability of winning less than 100 games out of 10 000 is less than 1 in 10^{24}.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

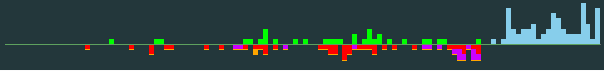
Pre-compute the optimal strategy and follow it for each game.

The resulting probability of winning is $\frac{374337547685188753}{15703912738882368000} \approx 0.023837$. The resulting probability of winning less than 100 games out of 10 000 is less than 1 in 10^{24}.

Solution: Optimal strategy is not necessary. For example, you can also linearly interpolate to make a good enough guess. There are many accepted solutions!

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

Pre-compute the optimal strategy and follow it for each game.

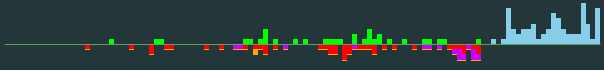
The resulting probability of winning is $\frac{374337547685188753}{15703912738882368000} \approx 0.023837$. The resulting probability of winning less than 100 games out of 10 000 is less than 1 in 10^{24}.

Solution: Optimal strategy is not necessary. For example, you can also linearly interpolate to make a good enough guess. There are many accepted solutions!

Fun fact: With 10 uniformly random real numbers in $[0, 1]$, the optimal probability of winning is about 0.020403.

L: Lock In

Problem author: Mike de Vries



Problem: Win the ranking game at least 1% of the time.

Observation: For a game with n numbers to be placed, we may always assume that we must place n random numbers chosen from $1, \ldots, m$.

Solution: We can determine the optimal solution recursively.

If we need to place a single number, the game is always winnable.

For each of these numbers, we must compute the optimal placement, placing k at index j is given the product by the winning chance of games of size $j-1$, and $n-j$, and the chance that $j-1$ of the rest of the randomly chosen numbers are at most $j-1$.

Pre-compute the optimal strategy and follow it for each game.

The resulting probability of winning is $\frac{374337547685188753}{15703912738882368000} \approx 0.023837$. The resulting probability of winning less than 100 games out of 10 000 is less than 1 in 10^{24}.

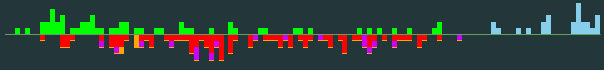
Solution: Optimal strategy is not necessary. For example, you can also linearly interpolate to make a good enough guess. There are many accepted solutions!

Fun fact: With 10 uniformly random real numbers in $[0, 1]$, the optimal probability of winning is about 0.020403.

Statistics: 160 submissions, 32 accepted, 70 unknown

M: Manic Ceramic

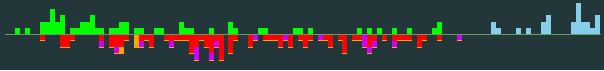
Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm

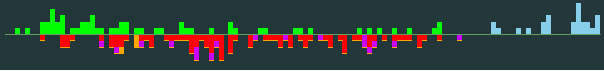


Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



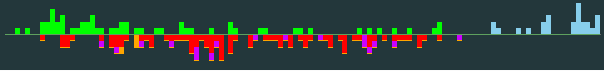
Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

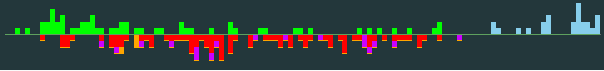
Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

Observation 2: We can rewrite this to $(y-1)N = (xy-1)A$, or $pN = qA$.

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

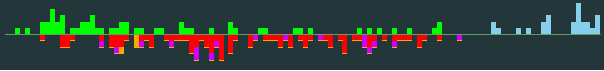
Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

Observation 2: We can rewrite this to $(y-1)N = (xy-1)A$, or $pN = qA$.

Observation 3: We are now looking for the smallest N for which q divides pN. This is the case when $pN = lcm(p, q)$.

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

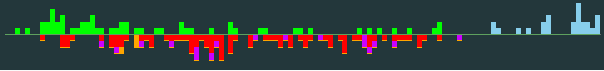
Observation 2: We can rewrite this to $(y-1)N = (xy-1)A$, or $pN = qA$.

Observation 3: We are now looking for the smallest N for which q divides pN. This is the case when $pN = lcm(p, q)$.

Solution: $N = \dfrac{\mathrm{lcm}(xy-1, y-1)}{y-1}$.

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

Observation 2: We can rewrite this to $(y-1)N = (xy-1)A$, or $pN = qA$.

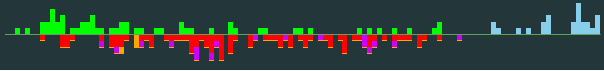
Observation 3: We are now looking for the smallest N for which q divides pN. This is the case when $pN = lcm(p, q)$.

Solution: $$N = \frac{\mathrm{lcm}(xy-1, y-1)}{y-1}.$$

Alternative: $$N = \frac{xy-1}{\gcd(x-1, y-1)}.$$

M: Manic Ceramic

Problem author: Arnoud van der Leer, Maarten Sijm



Problem: Given two types of tiles with x and $\frac{1}{y}$ instances of a pattern, how many should you at least buy to get equally as many patterns as tiles?

Equivalently: Minimize $N = A + B = Ax + \frac{B}{y}$.

Observation 1: We can get rid of B since we only really care about N: $N = Ax + \frac{N-A}{y}$.

Observation 2: We can rewrite this to $(y-1)N = (xy-1)A$, or $pN = qA$.

Observation 3: We are now looking for the smallest N for which q divides pN. This is the case when $pN = lcm(p, q)$.

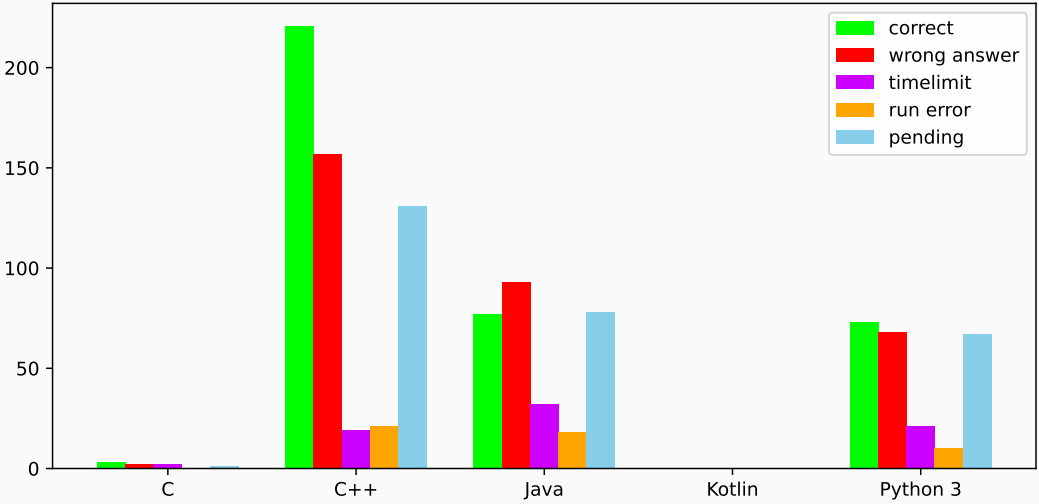
Solution: $N = \dfrac{\mathrm{lcm}(xy-1, y-1)}{y-1}$.

Alternative: $N = \dfrac{xy-1}{\gcd(x-1, y-1)}$.

Statistics: 180 submissions, 57 accepted, 25 unknown

ETV also did their best!

Language stats



Random facts

Jury work

- 382 commits (last year: 440)

Random facts

Jury work

- 382 commits (last year: 440)
  - About half of those were in the final two weeks before the first preliminary

Random facts

Jury work

- 382 commits (last year: 440)
  - About half of those were in the final two weeks before the first preliminary
- 839 secret test cases (last year: 802) ($\approx$ 64.5 per problem!)

Random facts

Jury work

- 382 commits (last year: 440)
  - About half of those were in the final two weeks before the first preliminary
- 839 secret test cases (last year: 802) ($\approx$ 64.5 per problem!)
- 237 jury + proofreader solutions (last year: 221)

Random facts

Jury work

- 382 commits (last year: 440)
  - About half of those were in the final two weeks before the first preliminary
- 839 secret test cases (last year: 802) (≈ 64.5 per problem!)
- 237 jury + proofreader solutions (last year: 221)
- The minimum[1] number of lines the jury needed to solve all problems is

$$8 + 2 + 15 + 14 + 1 + 1 + 20 + 12 + 18 + 1 + 11 + 9 + 1 = 113$$

  On average 8.7 lines per problem, up from 4 in last year's preliminaries

[1] With mostly™ PEP 8-compliant code golfing

Thanks to:

The proofreaders

Arnoud van der Leer
Bartjan Henkemans
Davina van Meer
Jaap Eldering
Kevin Verbeek
Pavel Kunyavskiy Kotlin Hero 🎈
Ragnar Groot Koerkamp
Thore Husfeldt
Thomas Verwoerd Kotlin Hero 🎈

The jury

Jeroen Op de Beek
Jippe Hoogeveen
Jonas van der Schaaf
Lammert Westerdijk
Leon van der Waal
Maarten Sijm
Marijn Adriaanse
Mike de Vries
Reinier Schmiermann
Tobias Röhr
Wietze Koops
Yanxu Chen

Want to join the jury? Submit to the Call for Problems of BAPC 2027 at:

`https://jury.bapc.eu/`