IV Кубок України з програмування 2025

Contest 2, div1

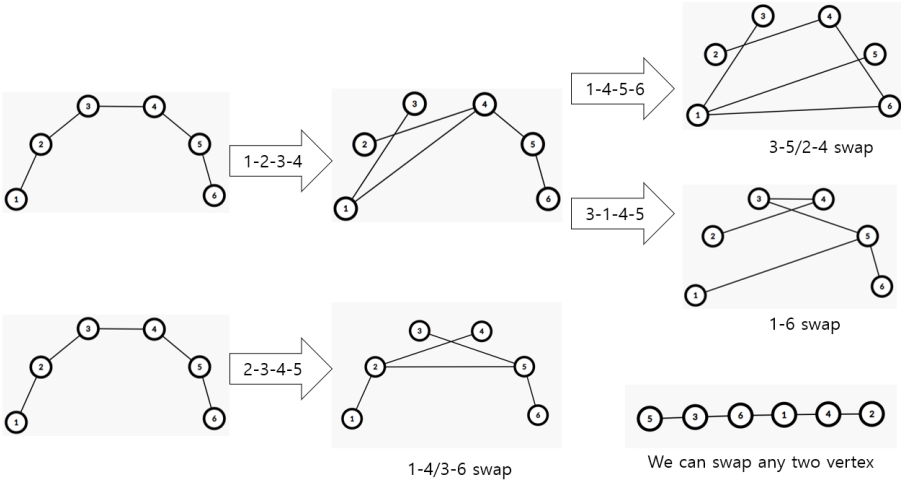
Official Solutions

A. Add, Remove, Transform

`ad_hoc`, `math`

Observation: For a subgraph isomorphic to a path of length 6, you can reorder the vertices in any order.

A. Add, Remove, Transform



A. Add, Remove, Transform

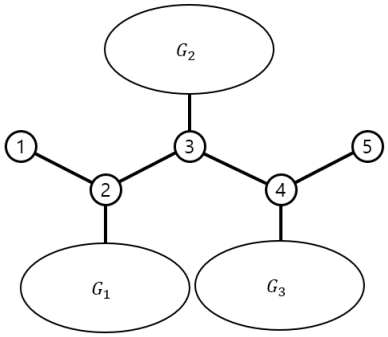
- For tree of 6 vertices, there are 6 non-isomorphic trees.
- Five are shown in the picture, and one is a star graph.
- We have shown that a tree of 6 vertices which is not a star graph could be a line graph by some sequence of operations, and any line graph could be reordered.
- Therefore any tree of 6 vertices could be any tree of 6 vertices by a sequence of operations when both trees are not a star graph.

A. Add, Remove, Transform

- First, let's make the diameter of tree less than or equal to 4.
- If there is a path of length 3 starting from vertex 1, remove edges at the path and add edges by operation. By this operation, degree of 1 increases by 1.
- Therefore after some operations there is no path of length 3 starting from vertex 1, resulting in a tree with diameter less than or equal to 4.

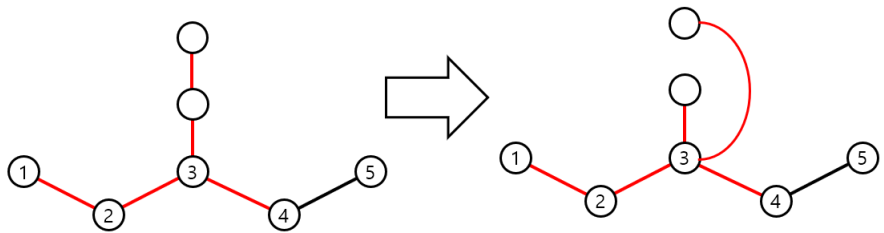
- If diameter is less than or equal to 3, we are done. Let's assume that diameter of a tree is 4.
- Let $(1, 2, 3, 4, 5)$ be a path of length 4.
- Let G_1, G_2, G_3 each be a partition of components of $G \setminus \{1, 2, 3, 4, 5\}$ where components of G_i is connected with $i + 1$ in G.

A. Add, Remove, Transform



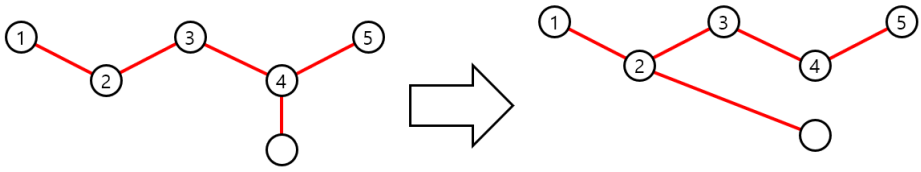
– Since diameter of tree is less than or equal to 4, there is no edge in G_1 and G_3.

A. Add, Remove, Transform



– We can remove edges in G_2.

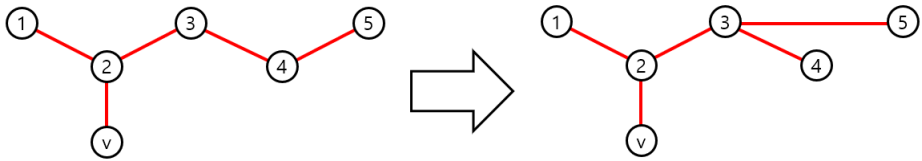
A. Add, Remove, Transform



– We can move vertices in G_3 to G_1.

- If $|G| = 5$, by removing edges $(2, 3), (3, 4), (4, 5)$ and adding $(2, 4), (3, 5), (2, 5)$, we get a tree of diameter 3.
- Otherwise, $G \setminus \{1, 2, 3, 4, 5\}$ is not empty. Let v be any vertex in $G \setminus \{1, 2, 3, 4, 5\}$.

A. Add, Remove, Transform



- We could move 5 into G_2.
- Now we have a tree where G_2 has no edges and G_3 has no vertices. This is a tree of diameter 3.

B. Big Sieve Game

`greedy`, `number_theory`

Definition

– Let's define an operation as an i–operation that increases or decreases p_j by 1 for all j divisible by i.

Observations

- The value of p_1 is determined solely by 1–operation. There is no need for 1–operation thereafter.
- For a prime number q, the value of p_q is determined by 1–operation and q–operation. Similarly, there is no need for 1–operation or q-operations thereafter.
- In general, for k, p_k is determined once all d–operation are completed for all divisors d of k.

Greedy Strategy

- Based on the observations, the optimal strategy is to prioritize determining p_a before p_b when $a|b$ and $a < b$.
- Therefore, simply iterating through $i = 1 \dots n$ in order and using i–operation to immediately set p_i to 0 repeatedly achieves the minimum value required by the problem.

C. Catch The Flea

graph

C. Catch The Flea

- Let's solve the problem in reverse.
- For any given cell, if it is possible to escape outside the trap by jumping K cells in its weak direction, it is considered an escapable cell.
- The cells reachable from escapable cells, i.e., R within K cells to the left, L within K cells to the right, D within K cells upwards, and U within K cells downwards from the escapable cell are escapable cells.
- Therefore, we can recursively find escapable cells.

C. Catch The Flea

- To optimize the runtime, we pre-process the nearest R on the left, L on the right, D upwards, and U downwards for each cell.
- Due to the tight time limit, managing connection information with vectors might lead to a time limit exceeded.
- Since each cell requires at most 4 connection information, it can be solved using arrays.

D. Decorative Birds

`dp`, `sweeping`, `segment_tree`

D. Decorative Birds

First, we can plot the situation on a 2D plane.

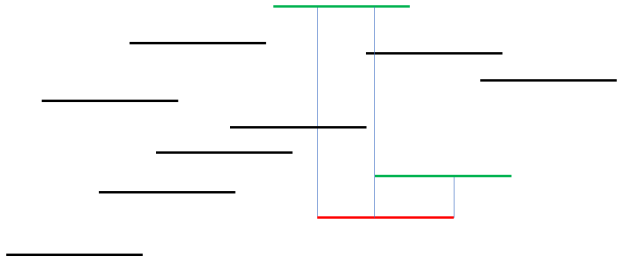
Consider a plane with x-axis denoting time and y-axis denoting speed of geese. Then a line segment of length L parallel to x-axis represents a goose.

D. Decorative Birds

Consider a set of segments that can be selected.
(= a set of geese that can be feeded)

Let’s observe properties of sets which can be selected.

D. Decorative Birds



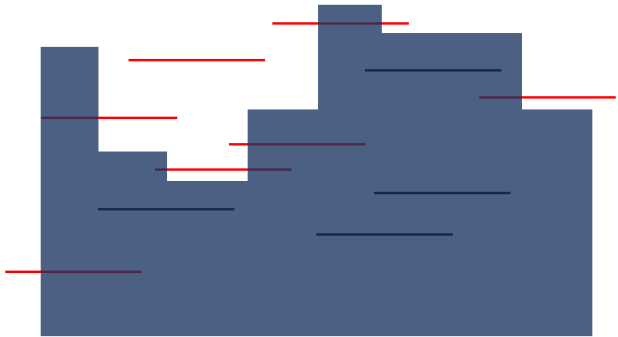
In this picture, green segments cover the red segment.

Every selected segment shouldn't be covered by segments which are not selected. Satisfying this condition for all segments is equivalent to saying that the set is valid.

Idea. Consider a histogram.

If we choose every segment which is not fully contained in a histogram, they form a valid set. (The formal proof is simple, so it is omitted.)

D. Decorative Birds



An example of a set generated by a histogram

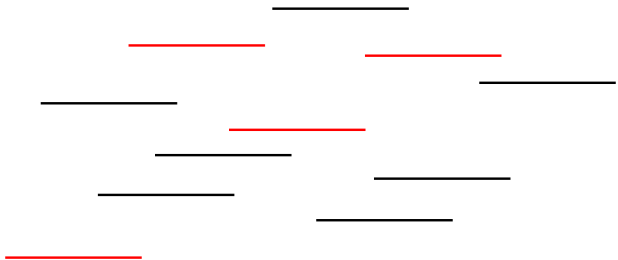
Idea. Consider a histogram.

Every histogram generates a valid set of segment.

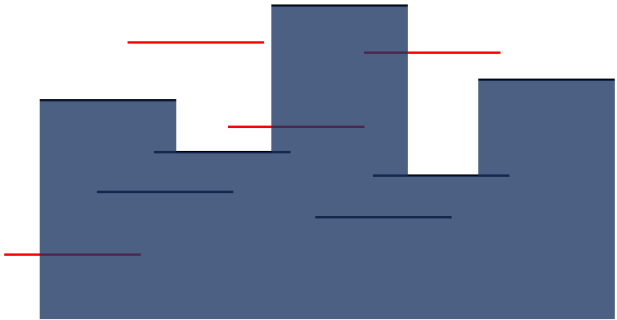
Conversely, every valid set of segments has a histogram generating it.

The area of histogram would be the union of areas under a segment not in the set.

D. Decorative Birds



D. Decorative Birds



Idea. Consider a histogram.

The conclusion of this idea is that we can consider every valid set by considering histograms that can be drawn.
Let's think about a dynamic programming algorithm on constructing histograms.

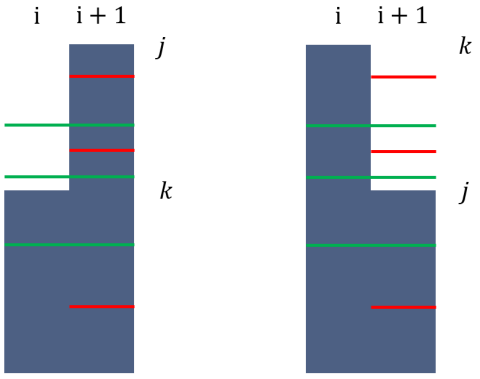
D. Decorative Birds

By compressing coordinates, we only have $O(N)$ points, both x-value and y-value.

$D_{i,j} :=$ maximum sum of weight of segments in a histogram which is drawn for x value $[0, i]$ and the height of the i-th unit segment is j.

Here, a segment is in a histogram drawn for x value $[0, i]$ if every point on a segment having x-value less than i is in the histogram.

D. Decorative Birds



Here green segments are related to $F_{i,j}$ and red segments are related to $A_{i,j}$.

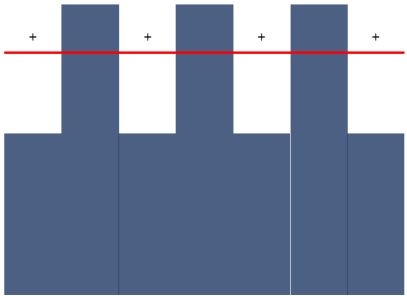
D. Decorative Birds

$$D_{i+1,j} = \max_{1 \le k \le N} (D_{i,k} + \max(F_{i+1,k} - F_{i+1,j}, 0)) + A_{i+1,j}$$

- $F_{i,j} :=$ sum of weight of segments which is under j and covers both $i-1$-th and i-th unit segment.
- $A_{i,j} :=$ sum of weight of segments which appears at i-th unit segment and is over j.

...But this DP is wrong!

D. Decorative Birds



D. Decorative Birds

One segment can be calculated multiple times!
Now we should use the property that **every segment has same length,** L.

Let's limit the length of a 'roof' at least L.
The 'roof' of a histogram means an interval having local maximum.
Since every segment has same length, any segment is only calculated at most once.

By 'sweeping down', we can show that every valid set can be generated by only using such histograms.

D. Decorative Birds

To limit the length of roofs, we can divide the DP into two tables, Uphill part / Downhill part.

Uphill part has a restriction that $k \leq j$ must be held.
Downhill part has a restriction that $j \leq k$ must be held.

Propagation from uphill to downhill must be delayed for length L.
Propagation from downhill to uphill can be done directly.

Note that at propagation from uphill to downhill, delayed length means x-value, not the number of unit segments.
The coordinates are compressed.

Now we got an $O(N^3)$ solution. We need to improve this method.
Consider histograms made by sweeping down a set.
We can cover all sets by only such histograms.

Idea. Every height changing only occurs on start/end of segments.
More precisely, the histogram jumps up to a starting point of a segment and falls down from a ending point of a segment.
The number of such points is $O(N)$.

D. Decorative Birds

Use segment trees to store DP.

At a certain x-value, a segment tree holds a row of a DP table.

Uphill part can implemented by a simple lazy max segment tree.

$$U_{i,j} = \max_{k \le j}(U_{i-1,k}) + A_{i,j}$$

By the previous idea, $k < j$ case only occur on a starting point and this can be done by a range max query. Most of cells maintain its value.

For A, total segment appearing events occurs N times and each appearing can be done by a lazy addition operation.

D. Decorative Birds

Downhill part is very complicated and requires sophisticated data structure technique.

$$D_{i,j} = \max_{j \le k}(D_{i-1,k} + F_{i-1,k}) - F_{i-1,j} + A_{i,j}$$

By the previous idea, $j < k$ case only occur on an ending point.
An ending point spreads the propagation by doing a range max update query.
But the F term is very annoying...

If $k = j$, F term has no effect.

Maintained values without changing height can be tackled same as uphill.

Idea. Range updates of F occur $O(N)$ times.

Appearing and disappearing of segments changes F for some range, $[y, N]$.

So we can manage a row of F by a lazy segment tree.

D. Decorative Birds

The spreading is done by applying $D_j = max(D_j, x + F_j)$ and $D_j = D_j + a$ queries. To handle propagation from downhill to uphill, we need to do some range max query for D.

So, we have to do...

- Given x, y, $D_j = max(D_j, x + F_j)$ for $j \in [1, y]$
- Given f, y, $F_j = F_j + f$ for $j \in [y, N]$
- Given a, y, $D_j = D_j + a$ for $j \in [1, y]$
- Given y, get $\max_{j \le y}(D_j)$

D. Decorative Birds

Store 5 values for each node: D, F, x, f, a

They mean lazy queries, $D = max(D, x + F) + a$ and $F += f$.

D is range max of D_j and F is range max of F_j.

Two lazy values can be added:

Applying x_2, f_2, a_2 to D, F, x_1, f_1, a_1

$=> \max(D, F + x_2) + a_2, F + f_2, \max(x_1 - f_2, x_2 - a_1), f_1 + f_2, a_1 + a_2$

OK, it works for every queries we want!

D. Decorative Birds

Our last problem is the propagation from uphill to downhill.
However, it's easy since the height never be changed!
Just delay the update and calculate sum of A for length L by using a fenwick tree or structure you want. It's just 2D sum query.

Finally we simulated $O(N^3)$ DP solution in $O(NlogN)$ with a segment tree.
You might need to consider some details to implement. Good Luck!

E. Excellent HLD

heavy_light_decomposition

E. Excellent HLD

- The problem can be viewed as increasing the weight of edges in a path for each query and solving the sum of weight of heavy edge for every vertex.
- **Observation**: For Q queries, number of changes of heavy edge is bounded to $4QlogN$.

E. Excellent HLD

- Define **hld-heavy edge** as a heavy edge in heavy light decomposition, where the child incident to heavy edge has the biggest subtree size. Define **hld-light edge** as edges that are not hld-heavy edge.
- When we add weight to a path, we can observe that heavy edge either changes to the edge in the path or it does not change.
- Let's assume that heavy edge changes to the edge on a path for every vertex on a path.

E. Excellent HLD

- When we follow the hld-light edge in upward direction, size of subtree doubles. Therefore there are at most $logN$ hld-light edge in a path from a leaf vertex to root vertex.
- There are at most $2logN$ hld-light edges on a path. Therefore there are at most $2QlogN$ changes where heavy edge changes into hld-light edge.
- For each vertex, there are more changes where heavy edge changes into hld-light edge than heavy edge changes into a hld-heavy edge. So the number of heavy edge change is bounded to $4QlogN$.

E. Excellent HLD

- Since the number of heavy edge change is bounded, let's try to find the change of heavy edge fast enough.
- For each query, by using heavy light decomposition, we can divide the path on a tree into $logN$ chains.
- For each chain, use segment tree to store (weight of hld-heavy edge) - (maximum weight of hld-light edge) for every vertex. By maintaining such value for vertices that have heavy edge on hld-light edge, we can efficiently find the change of heavy edge from hld-light edge to hld-heavy edge in $O(logN)$ time for each change.
- Total complexity is $O((N+Q)log^2N)$.

F. Full Irreducibility

ad_hoc

- Denote $D_i = \{P_1, P_2, \cdots, P_i\}$. For P to be an irreducible permutation, $D_i \neq \{1, 2, \cdots, i\}$ should satisfy for $1 \leq i \leq N-1$.
- When ith element and $i+1$th element is swapped, D_i changes and D_j remains the same for every $j \neq i$.
- For every indices i where $D_i = \{1, 2, \cdots, i\}$, swap ith element and $i+1$th element. The order does not matter.

G. Good Triangle

`ad_hoc`, `segment_tree`

G. Good Triangle

- Let's try to find a simpler way to define whether there is a point having same distance from given three points.
- A circle in manhattan distance is 45 degree rotated square. Let's rotate the grid 45 degrees.
- If coordinates are monotone, there is no square that passes through three dots. Otherwise there always exists such square.
- By using segment tree, we can find the number of three dots that have monotone coordinate.

H. Holes in Queue

`ad_hoc`, `segtree`, `two_pointer`

H. Holes in Queue

- Let's define $f^{(d)}(x)$ as the number on the xth index after d pop operations.
- Then the following properties hold:
  - $f^{(0)}(x) = x$
  - $f^{(d)}(x) = f^{(d-1)}(f(x))$
  - $f(x) = x + (\#\, of\ deletion\ points \leq x)$
- Naively implementing this function will lead to more than $O(d)$ time complexity per query.

H. Holes in Queue

- Let's sort $(A[1], A[2], ..., A[N])$, and assume $A[0] = 0$, $A[N+1] = \infty$.
- It is clear that for all $x < A[1]$, $f^{(d)}(x) = x$.
- So we will consider only xs greater than or equal to $A[1]$ further on.
- Then we can modify the property as:
  - $f(x) = x + i$, where $A[i] \leq x < A[i+1]$
- And it can be further improved to:
  - $f^{(d)}(x) = x + d \times i$, where $A[i] \leq x < x + (d-1) \times i < A[i+1]$
- Therefore the critical part of this problem is to figure out a clever way to manage cases where both $A[i] \leq x < A[i+1]$, and $A[i+1] \leq x + i$.

H. Holes in Queue

- The main idea is to partition the whole queue into 'blocks', such that for all the elements from a left block, their $f(x)$ will be placed in an adjacent right block.
- We can do this with the following simple algorithm:
  - Start from $A[1]$. The initial block length is 1.
  - Repeat partitioning with length 1 until we meet $A[2]$.
  - Increment the block length by 1.
  - Repeat partitioning with length 2 until we meet $A[3]$.
  - Increment the block length by 1.
  - .....

H. Holes in Queue

- This is an example where the deletion points are 1, 3, 8, 10.
  - (1) (2) 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
  - (1) (2) (3 4) (5 6) 7 8 9 10 11 12 13 14 15 16 17
  - (1) (2) (3 4) (5 6) (7 8 9) 10 11 12 13 14 15 16 17
  - (1) (2) (3 4) (5 6) (7 8 9) (10 11 12 13) (14 15 16 17)

- You can notice that the problem is now much easier.
- Let's define:
  - $len(b) =$(length of the bth block)
  - $g(b, i) =$(ith element of the bth block)
- Then we can rewrite f to:
  - $f(g(b,i)) = \begin{cases} g(b+1,i), & \text{if } len(b) == len(b+1) \\ g(b+1,j), & \text{otherwise} \end{cases}$
  - where the above j is the ith non-deletion index of the $b+1$th block.
  - Note that at least 1 deletion index exists in the 2nd case.

H. Holes in Queue

- Now the whole problem comes down to finding the appropriate j, where
  - $f^D(g(b,i)) = g(b+D,j)$
- Let's further define:
  - $R[i] = j$, where $f^d(A[i]) = g(\_, j)$, when d is sufficiently large
- We can calculate all $R[i]$s with proper segment tree operations.

H. Holes in Queue

- We can calculate the appropriate j using sequence $R[1]$, $R[2]$,..., $R[N]$.
- Let's try to calculate $f^D(g(b,i))$.
- First find $R[k]$, which is the ith smallest element among $R[1]$, $R[2]$, ..., $R[len(b)]$.
- Then j is the # of elements smaller or equal to $R[k]$ among $R[1]$, $R[2]$, ..., $R[len(b+D)]$.
- One can prove it directly easilly, so we skip the proof in this section.

H. Holes in Queue

- Since D is constant, we can conduct a two pointer of the index of blocks b and $b + D$ and answer the queries offline.
- Keep in mind that the number of blocks may be large as 10^{18}, but the number of blocks with distinct lengths don't exceed N, and the block length monotonically increases. One needs careful implementation to bound the time complexity.
- With segment tree similar to the one we used to find $R[i]$s, we can do both operations in $O(logN)$.
- The total time complexity is $O((N + Q)logN + QlogQ)$, due to sorting and segtree operations.

H. Holes in Queue

- Instead of using two pointers, we can also use 2D data structures such as persistent segment trees.
- With this implementation, we can also process queries with different Ds with the same time complexity, but much slower due to high constants.

I. Isn’t It Beautiful?

`ad_hoc`

I. Isn't It Beautiful?

- Let the ith bit of x be 0 and $0, 1, \cdots, i-1$th bit of x be 1. Then $x \& A_j$ could not be 2^i.
- So it is optimal for bits higher than ith bit to be 0. Therefore $x = 2^i - 1$ for some i.
- By bruteforcing through $0 \leq i \leq 20$, we can get the answer.

J. Joy Of Sleep

`graph`

- First, the time it takes for the i-th chameleon to wake the j-th chameleon is $max(|X_i - X_j|, |Y_i - Y_j|)$ if the colors of the two chameleons are the same, and $min(|X_i - X_j|, |Y_i - Y_j|)$ if the colors are different.
- Let's define the event of the i-th chameleon waking the j-th chameleon satisfying $C_i = C_j$ as 'using a max edge,' which takes max_edge(i, j) seconds.
- Similarly, let's define the event of the i-th chameleon waking the j-th chameleon satisfying $C_i \neq C_j$ as 'using a min edge,' which takes min_edge(i, j) seconds.

Observation

- There is no need to use the max edge more than twice.
- If max edge is used consecutively to reach from one vertex to another, using max edge only once from start vertex to end vertex is more efficient.
- Is it necessary to use it once?
- Let's denote the currently awake chameleon as the i-th chameleon, and the chameleon to be awakened at the end as the j-th chameleon.
- If $C_i \neq C_j$, using min_edge(i, j) is faster than using the max edge at least once. Therefore, there is no need to use the max edge.

Observation

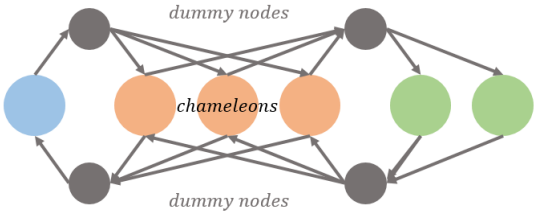
- Assume $C_i = C_j$, and both the max edge and min edge are used at least once. Since the min edge is used, the chameleon must consist of at least two different colors. For the k-th chameleon, where $C_i \neq C_k$, using only the min edge(min(i, k) + min(k, j)) is faster or takes the same time as using both the max and min edges. Hence, the case of using both the max and min edges does not exist.
- Therefore, the max edge is not used except once when the 1st chameleon wakes up the Nth chameleon.

Optimization

- Min edge connecting (X_i, Y_i) and (X_j, Y_j) could be separated into two edges with weight of $|X_i - X_j|$ and $|Y_i - Y_j|$. Let an edge with weight $|X_i - X_j|$ be an x-edge and edge with $|Y_i - Y_j|$ be y-edge.
- A graph where every two chameleons with different color are connected with a x-edge has too many edges. Let's construct an equivalent graph which has less edges.
- Consider i-th, j-th, and k-th chameleons where $C_i \neq C_j$, $C_j \neq C_k$ and $X_i \leq X_j \leq X_k$. Since $|X_i - X_k| = |X_i - X_j| + |X_j - X_k|$ we don't require a x-edge connecting i and k.
- Sort chameleons with it's x value and cluster adjacent chameleons with it's color. Making a x-edge with chameleons in two adjacent cluster would be enough.

Optimization

- For two cluster of size p and q, naively adding x-edges would need pq edges. By adding a dummy node between two cluster and connecting chameleons in clusters with the dummy node with a x-edge, we only need $p + q$ edges.
- This only requires at most $4N$ x-edges.



Optimization

- Similarly doing for y edge, we can construct a graph with at most $5N$ nodes and $8N$ edges which is equivalent to a graph with min edges between chameleons with different colors.
- Let's add a max edge between chameleon 1 and chameleon N if two chameleons have the same color.
- We can easily solve the problem using dijkstra algorithm.

L. Lottery

`dynamic_programming`, `math`

L. Lottery

- If there are several albums that have same cost, we can purchase the album that gives the most lottery tickets.
- We can consider only $max(A_i)$ albums.

L. Lottery

- Let T be a list of album you bought.
- The expected cost can be calculated as $\sum_{i \in T} A_i + R\frac{S + \sum_{i \in T} B_i}{\sum_{i \in T} B_i} = \sum_{i \in T} A_i + \frac{RS}{\sum_{i \in T} B_i} + R.$
- Denote $f(t)$ as $\max(\sum_{i \in T} B_i)$ when $\sum_{i \in T} A_i \leq t$.
- Our goal is to solve minimum value of $t + \frac{RS}{f(t)}$.

L. Lottery

- Let pth album have the maximum value of $\frac{B_i}{A_i}$. Let $K = \frac{B_p}{A_p}$.
- We can observe that $f(t) \leq Kt$.
- Also we can purchase $\lfloor \frac{t}{A_p} \rfloor$ copies of pth album. Therefore

$$f(t) \geq \lfloor \frac{t}{A_p} \rfloor B_p \geq (\frac{t}{A_p} - 1)B_p = Kt - B_p$$

- $t + \frac{RS}{f(t)} \leq t + \frac{RS}{Kt - B_p} = \frac{B_p}{K} + t - \frac{B_p}{K} + \frac{\frac{RS}{K}}{t - \frac{B_p}{K}} = A_p + t - A_p + \frac{\frac{RS}{K}}{t - A_p}$
- Let $t = \lfloor \sqrt{\frac{RS}{K}} + A_p \rfloor$.
- $t + \frac{RS}{Kt - B_p}$ is approximately $A_p + 2\sqrt{\frac{RS}{K}}$
- Now we have bounded t to $A_p + 2\sqrt{\frac{RS}{K}} + M$, where M is a constant made by floor operation on t.

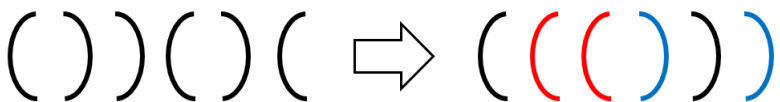
- Let's use dynamic programming for solving $f(t)$. This can be solved in a knapsack dynamic programming fashion.
- Time complexity would be $O(max(t) \times N)$.
- We have observed that $N \leq max(A_i)$ and $max(t) \leq 2\sqrt{\frac{RS}{K}} + A_p + M$. This would be enough to pass through time limit.

M. Make It Regular

`dynamic_programming`

M. Make It Regular

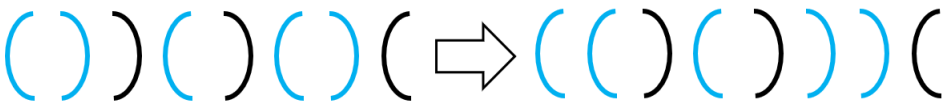
- When we fix S, it is optimal to let open brackets be on the left side and closed brackets to be on the right side.
- Let's observe which brackets are flipped. On the left side, some closed brackets are flipped into open brackets. On the right side, some open brackets are flipped into closed brackets.



M. Make It Regular

- Define a proper bracket prefix as a bracket sequence where for every prefix, there are an equal or greater number of open brackets than closed brackets.
- Define $dp1_{i,j}$ as the number of possible choices of j closed brackets and some open brackets from 1th bracket to ith bracket where by flipping chosen closed brackets into open brackets, prefix $[1, i]$ becomes a proper bracket prefix.
- Define $dp2_{i,j}$ similarly for suffixes. It will be the number of possible choices of j open brackets and some closed brackets from ith bracket to $2N$th bracket where by flipping open brackets, suffix $[i, 2N]$ becomes a proper bracket suffix.
- Both $dp1$ and $dp2$ can be easily solved in $O(N^2)$.

- For each S, let i_S be the number of flipped open/closed brackets, and x_S be the index of first closed bracket in S after swapping is done.
- For every (x, i), let's count the number of S where $i_S = i$ and $x_S = x$.



$i_S = 1, x_S = 6$

- If xth bracket is an closed bracket, possible choices of S would be $dp1_{x-1,i} \times dp2_{x+1,i}$.
- Otherwise, we should flip the xth bracket. If flipping i closed brackets in $[x, 2N]$ does not result in a proper bracket suffix, the result would be 0. Otherwise possible choices of S would be $dp1_{x-1,i} \times dp2_{x+1,i-1}$.

– Note that we did not count cases where no brackets are flipped. If the sequence itself is a proper bracket sequence, the answer would be $2^{2N} - 1$. Otherwise, we do not need to count such case.