APIO 2021 Problems Discussion

Gunawan, Jonathan Irvin
Jump Trading Pacific Pte. Ltd.
jonathanirvingunawan@gmail.com

Djonatan, Prabowo
Garena Online Pte. Ltd.
prabowo1048576@gmail.com

Rifa'i, Wiwit
Google Taiwan Engineering Limited
wiwitrifai@gmail.com

Mualim, Sebastian
Shopee Singapore Pte. Ltd.
sebastianmualim@gmail.com

Nurrokhman, Abdul Malik
IA TOKI
abdul1024malik@gmail.com

Yudhiono, Hocky
IA TOKI
hocky.yudhiono@gmail.com

May 26, 2021

Below are the official solutions used by APIO 2021 Scientific Committee. Note that there might be more than one solutions to some subtasks. Also, note that the order of the subtasks in the discussion might not be ordered for the ease of discussion.

Since the purpose of this editorial is mainly to give the general idea to solve each subtask, we left several (implementation) details in the discussion for reader's exercise.

1 A. Hexagonal Territory

Problem Author
Written by: Wiwit Rifa'i
Prepared by: Wiwit Rifa'i
Solutions, review, and other problem preparations by: Prabowo Djonatan, Jonathan Irvin Gunawan, Abdul Malik Nurrokhman
Analysis author: Wiwit Rifa'i

1.1 Subtask 1

If $N = 3$, the territory will form an equilateral triangle ($L[0] = L[1] = L[2]$) with each side has a length that will cover $L[0] + 1$ cells. Since the initial cell is located at the corner cell of the triangle, there are exactly $d+1$ cells that has distance d in the territory for all $0 \leq d \leq L[0]$. Since $B = 0$, all cells have the same score (i.e. A). So, we can use the arithmetic summation formula to calculate the total score as the following.

$$Total\ Score = \sum_{d=0}^{L[0]} (d+1) \times A = \frac{(L[0]+1) \times (L[0]+2)}{2} \times A$$

1.2 Subtask 2

This subtask is still similar to subtask 1, i.e. the territory forms an equilateral triangle. The only difference is the score of each cell could be different depending on the distance. Therefore, the total score can be calculated as the following formula.

$$Total\ Score = \sum_{d=0}^{L[0]} (d+1) \times (A + d \times B) = \sum_{d=0}^{L[0]} A + \sum_{d=0}^{L[0]} d \times (A+B) + \sum_{d=0}^{L[0]} d^2 \times B$$

Using $\sum_{d=0}^{S} d = \frac{1}{2} \times S \times (S+1)$ and $\sum_{d=0}^{S} d^2 = \frac{1}{6} \times S \times (S+1) \times (2 \times S + 1)$, we can get the final form of the formula. When calculating the total score, we should be careful of integer overflow and make sure that the division is done correctly before the modulo or we can simply use modular inverse for the division.

1.3 Subtask 3

Observation 1.1. *We can assign each cell to an index (X, Y) so that every cell (X, Y) will be neighbouring with cell $(X, Y+1)$, $(X+1, Y+1)$, $(X+1, Y)$, $(X, Y-1)$, $(X-1, Y-1)$, and $(X-1, Y)$ in each direction from 1 to 6 respectively. If we map the cells into lattice points in 2D Cartesian Coordinates, it means we can move between lattice points horizontally, vertically, or diagonally (only one diagonal: $y = x + c$).*

Let $S = \sum_{i=0}^{N-1} L[i]$. Since $S \leq 2000$ in this subtask, we can put all cells in the territory into a grid with size $S \times S$. First, we can use flood-fill starting from the outermost cells of the grid to mark all cells in the grid that are outside of the territory. Then, we can use BFS from the initial cell to calculate the distance of each cell in the territory and calculate the total score.

The running time of this solution is $O(S^2)$ where $S = \sum_{i=0}^{N-1} L[i]$.

1.4 Subtask 4

Since $B = 0$, we just need to count the number of cells in the territory and then multiply it with A to get the total score. To count the number of cells in the territory, we can use something like prefix-sum. First, we need to find all cells that are on the top boundaries of the territory and all cells that are on the bottom boundaries of the territory. A cell is said to be on the top boundaries if this cell is part of the territory but the upper neighbour is outside of the territory. Likewise, a cell is said to be on the bottom boundaries if this cell is part of the territory but the lower neighbour is outside of the territory. For each cell (X, Y) on the top boundaries of the territory, we add the value of Y to the count. For each cell (X, Y) on the bottom boundaries of the territory, we add the value of $-Y + 1$ to the count. Adding all those values will give us the number of cells in the territory.

To find the top boundary cells and the bottom boundary cells, we can use flood-fill similar to subtask 3. First, we collect all cells in the path and their direct neighbours, and put them into a set. Then, we can do flood-fill in the set starting from the lowest cell in the set to mark all cells in the set that are outside of the territory. And for each cell in the path, we can know whether this cell is on the top boundaries and/or on the bottom boundaries by checking its upper and lower neighbours are outside of the territory or not. The running time of this solution is $O(S \log S)$ where $S = \sum_{i=0}^{N-1} L[i]$.

1.5 Subtask 5

To solve this subtask, we can use Pick's theorem and Shoelace formula to count the number of cells in the territory. By using Pick's theorem, we can count the number of points inside the territory using formula $i = a - \frac{b}{2} + 1$ where i is the number of points inside the territory, a is the area of the polygon representing the territory, and b is the number of points on the boundary which equals to $\sum_{i=0}^{N-1} L[i]$. To calculate the area a, we can use Shoelace formula. And so, the number of cells in the territory is $i + b$.

There is another solution by optimizing the solution from subtask 4 using the following observation.

Observation 1.2. *WLOG, suppose the path circulates the territory in counter-clockwise direction (i.e. if we use the Shoelace formula then the result will be positive, otherwise we can reverse the visiting order to make it positive). We can observe that if we are moving to the left (horizontally or diagonally) then most of the upper neighbours are outside of the territory (the exception can only happen when the direction is changing), meanwhile if we are moving to the right (horizontally or diagonally) then most of the lower neighbours are outside of the territory (the exception can only happen when the direction is changing).*

Using this observation, we can find some line segments that make the top boundaries and some line segments that make the bottom boundaries. And then, we can use arithmetic formula to do prefix-sum similar to subtask 4 for counting the number of cells in the territory based on those line segments.

The running time of both solutions is $O(N)$.

1.6 Subtask 6

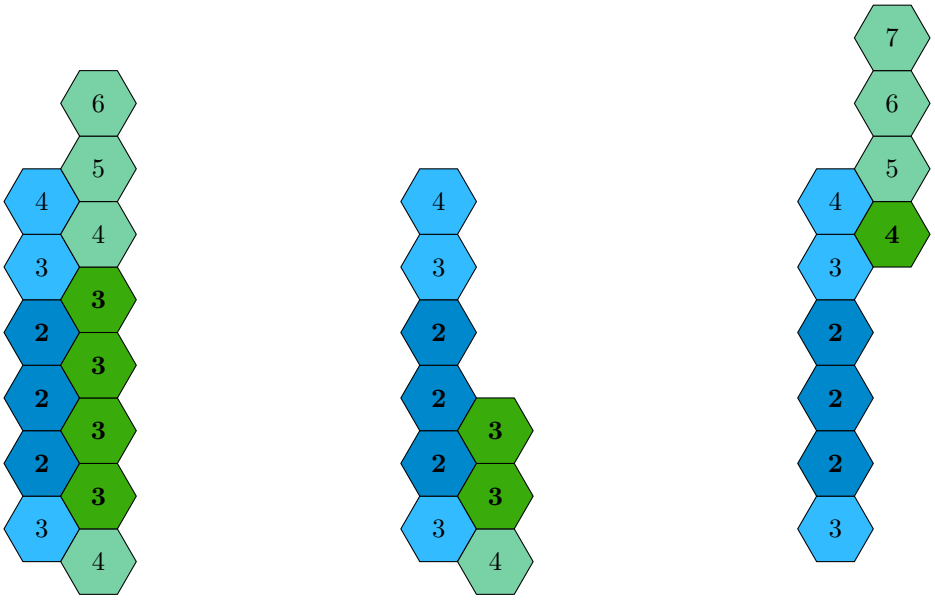
To solve this subtask, we need several observations.

Observation 1.3. *Define a column as a group of contiguous cells that are neighbouring vertically in the territory such that two cells are at the same column if and only if both cells have the same X-coordinate and all cells between them are also part of the territory. If we represent each column as a node in a graph and we add an edge for every two touching columns, the resulting graph will be a tree.*

Observation 1.3 is correct because the territory forms a simple polygon without holes. So for every two cells, any shortest path between them will always have the same set of visited columns.

Observation 1.4. *Following observation 1.3, if we make the column containing the initial cell as the root node of the tree, then the distances of cells in the column of the child node can be calculated only based on the distances of cells in the column of the parent node. And, if we observe the distances of cells in each column, then there's a pattern. From the top-most cell to the bottom-most cell in a column, the value of the distances will constantly decrease by 1 until it reaches the lowest distance in the column, then the lowest distance can be repeated multiple times, and then the distance will constantly increase by 1.*

Observation 1.4 can be proved using mathematical induction. First, it's obvious that the column containing the initial cell follows the pattern. Then, we can traverse the other columns based on the tree using DFS/BFS. When we visit a column, we can fill the distances in this column based on the distances in the parent column. The lowest distance in the child column is usually increased by 1 from the parent column and the occurrence of the lowest distance is also increased by 1, except if the positions of the lowest distance in the child column are truncated or shifted because of the difference in the position of top-most and bottom-most cells between both columns. The following is the illustration how we update the distances in the right column based on the distances in the left column for some various cases.



To construct the columns, we can first find the top boundary cells and the bottom boundary cells similar to subtask 4 or subtask 5. Then, we can construct the columns by matching the top boundaries and the bottom boundaries. Then, we construct the tree, and traverse the tree using DFS/BFS. For each column, we just need to save the lowest distance and its position range. We can calculate the total score for each column using arithmetic formula and sum them as the answer. The running time of this solution is $O(S \log S)$ where $S = \sum_{i=0}^{N-1} L[i]$.

1.7 Subtask 7

If we use the solution from the previous subtask, then the number of columns will be too large. So, we need to merge several columns into a block. A block should consist of several contiguous columns such that the

top-most cells form a straight line segment (horizontally or diagonally) and the bottom-most cells also form a straight line segment (horizontally or diagonally). For each block, we still can calculate the total score in $O(1)$ by combining some formulas similar to formulas in subtask 1 and subtask 2.

Let the initial cell is located at cell $(0, 0)$. Since the value of $L[i]$ is constant, we can construct the blocks as the following.

- If a column has X-coordinate that is divisible by $L[0]$, then this column can be an independent block (not merged to other columns).
- The other columns will be merged if they are contiguous or touching each other since their top-most cells and bottom-most cells will be guaranteed to form straight line segments. Each of these blocks will consist of $L[0] - 1$ columns.

The rest of the solution is almost similar to the previous subtask with the columns are replaced by blocks. The running time of this solution is $O(N \log N)$.

1.8 Subtask 8

The full solution is almost similar to the previous subtask, but we cannot construct the blocks using the same method. To construct the blocks, we need to find the top and bottom boundary line segments similar to subtask 5. Then, we can use sweep line algorithm from the left to the right while maintaining a set containing those boundary line segments. The set will contain all boundary line segments that are being touched by the sweeping line. The bottom boundary line segments can be lowered by 1 so that there's no intersection between any two line segments, and we can use custom comparator such that the line segments are always ordered correctly inside the set from the lower to higher line segments. When we insert or remove a line segment from the set, we can check whether there's a block that can be constructed using the upper and/or lower line segments inside the set.

After the block construction, the rest of the solution is still the same as the previous subtask. The running time of this solution is $O(N \log N)$.

2 B. Rainforest Jumps

Problem Author
Written by: Benson Lin
Prepared by: Jonathan Irvin Gunawan
Solutions, review, and other problem preparations by: Prabowo Djonatan, Sebastian Mualim
Analysis author: Jonathan Irvin Gunawan, adapted from the analysis written by Benson Lin

2.1 Subtask 1

The orangutan can only jump one tree to the right. Therefore, the answer for all plans is $C - B$.

2.2 Subtask 2

We can construct a directed graph where each node corresponds to each tree and a directed edge is added between two nodes if the orangutan can jump from one tree to the other. We can then compute the all-pair-shortest-path of this graph in $O(N^3)$ using Floyd-Warshall.

Each plan can be answered by iterating all possible starting point and all possible ending point. The running time of this solution is $O(N^3 + QN^2)$.

2.3 Subtask 3

To solve this subtask, we need to optimize the computation of all-pair-shortest-path to $O(N^2)$ using BFS from each starting point. We also need to be able to find the ending point between C and D with the minimum distance from each possible starting point in $O(\log N)$ using standard range-min-query data structure. The running time of this solution is $O(QN \log N)$.

2.4 Subtask 4

To solve this subtask, we need to construct the graph in $O(N)$ using monotonic stack. Since there is a small number of plans, we can answer each plan in $O(N)$ time. To do this, we can use BFS where we put all nodes between tree A and B in the initial queue and finish the search when we visit any nodes between tree C and D. The running time of this solution is $O(QN)$.

2.5 Subtask 5

To solve this subtask, we need several observations.

Observation 2.1. *Let $t = C = D$. Suppose the orangutan is currently at tree x and can jump to tree y and z and $H[y] > H[z]$. If $H[y] \leq H[t]$, it is not worse to jump to tree y instead of tree z.*

For each node with two outgoing edges, we call the edge that goes towards the higher vertex the **high edge** and call the other the **low edge**. If a vertex only has one outgoing edge, that edge is a **low edge**.

Observation 2.2. *Following observation 2.1, an optimal path from tree s to t is obtained by always taking the high edge as long as the height of the destination tree is not more than $H[t]$, and then always taking the low edge to tree t. If the low edge goes to a tree taller than tree t, then the plan is impossible.*

Using this observation, we can build two sparse tables, one containing the destination after always taking the high edges 2^z times from each tree and another one containing the destination after always taking the low edges 2^z times. We can build this sparse table in $\mathrm{O}(N \log N)$ and use them to answer each plan in $O(\log N)$. The running time of this solution is $O((N + Q) \log N)$.

2.6 Subtask 6

Let $t = C = D$ and the **reverse-GIS (greedy-increasing subsequence)** of a plan be the sequence $s_1, s_2, \ldots, s_k$ such that:

- $s_1 = B$
- s_{i+1} is the largest index $A \leq x < s_i$ such that $h[x] > h[s_i]$ for each $i \geq 1$.

If the orangutan starts on a tree x not in reverse-GIS, then any sequence of jumps to tree t must visit either one tree in reverse-GIS, or the tree that is reached by jumping from s_k to the left. Therefore, if the orangutan can go to tree t from some tree in $[A, B]$, then the orangutan can go to tree t from some tree x in reverse-GIS with less or equal number of jumps.

Let l be the largest index such that $H[s_l] < H[t]$. If there is no such l, then the plan is impossible. If the orangutan starts from tree $s_{l'}$:

- If $l' > l$, then it is impossible to reach tree t since $H[s_{l'}] > H[t]$.
- If $l' < l$, then any sequence of jumps to tree t must visit either tree s_l or the tree that is reached by jumping from s_l to the right.

Therefore, it is not worse to start from tree s_l. We can also construct a sparse table in $\mathrm{O}(N \log N)$ to find the value of l for each plan in $O(\log N)$. The running time of this solution is $O((N + Q) \log N)$.

2.7 Subtask 7

We will say tree x is *reachable* on this plan if the orangutan can jump to the right from tree x and jumping to the right will not go to the right of tree D. Otherwise, that means tree x is higher than all trees y ($C \leq y \leq D$) and the orangutan will never reach any of the tree destination from tree x.

Continuing from subtask 6, let l be the largest index such that tree s_l is reachable. It is not worse to start from s_l. If there is no such l, then the plan is impossible.

Observation 2.3. *Extending from observation 2.2, starting from tree s_l, if both the high edge and low edge go to a tree that is reachable and are not any of the destination trees, then always use the high edge. After that, always jump to the right. If at any point the orangutan reach a tree that is not reachable, then the plan is impossible.*

We can build all the necessary sparse tables in $O(N \log N)$ which are similar to those from subtask 5 and subtask 6. The running time of this solution is $O((N + Q) \log N)$.

3 C. Road Closures

Problem Author
Written by: Jatin Yadav
Prepared by: Hocky Yudhiono
Solutions, review, and other problem preparations by: Prabowo Djonatan, Jonathan Irvin Gunawan, Wiwit Rifa'i, Jatin Yadav
Analysis author: Hocky Yudhiono, adapted from the analysis written by Jatin Yadav

3.1 Subtask 1

The tree given in this subtask forms a star. For each k, the answer can be obtained by closing $n-k-1$ roads with the least cost. Therefore, a sorting algorithm can be used here. Then, by iterating and keeping the sum of cost, the answer can be obtained. The running time of this solution is $O(N \log N)$.

3.2 Subtask 2

The tree given in this subtask forms a line. For case $k = 0$, the answer is the sum of all edges as no vertices are allowed to have degree > 0.

For case $k \geq 2$, the tree has no vertices with degree > 2, so the answer will always be 0.

For case $k = 1$, the problem is reduced to an array. We can use the array W given in the input. We must find the minimum cost such that no two consecutive costs are not taken. This is because for each vertex, it can only connect to one other vertex.

One of the common ideas to solve this is to use dynamic programming. Define $dp[i]$ as the minimum cost for a valid closure of road from 0 to i inclusively with road connecting the junction i and $i-1$ closed.

We can define the transition as $dp[i] = \min(dp[i-2], dp[i-1]) + W[i-1]; (2 \leq i \leq n-2)$, with base case $dp[0] = 0$ and $dp[1] = W[0]$ The answer can be obtained in $\min(dp[n-2], dp[n-1])$. The running time of this solution is $O(N)$.

3.3 Subtask 3

This problem can be further reduced into dynamic programming on tree problem. We will compute the answer for each k separately. We can consider the tree as a rooted tree.

For each vertex i starting from the root, define $dp[i][c]$, where c is a boolean value on whether the road connecting this vertex and its parent's has been closed. We want to minimize the cost to close some roads so that i has no more than k degree. Define E as the set of all children of i, and P as the set of picked children, where the road connecting i to $p \in P$, with cost $W[p]$ will be closed. Optimally, $|P| = \max(0, \text{degree}(i) - K)$.

The dynamic programming will be used to pick the minimum cost by minimizing $\sum_{j \in \mathrm{P}} dp[j][1] + \sum_{j \notin \mathrm{P}} dp[j][0]$. We can compute this like a knapsack problem. Iterate for each $e \in E$ and determine whether to close the road connecting the vertex i and e or not. The dp for all vertices can be computed in $O(N^2)$ because each vertices i will be iterated in $O(\text{degree}(i))$ and $\sum_i \text{degree}(i) = 2(N-1)$.

The running time of this solution for all k is $O(N^3)$.

3.4 Subtask 4

We will optimize the computation of dp from subtask 3. The initial equation can be rewritten as $\sum_{j \in \mathrm{E}} dp[j][0] + \sum_{j \in \mathrm{P}} (dp[j][1] - dp[j][0])$. To find the best picked children, one can greedily sort them by comparing $dp[j][1] - dp[j][0]$ for each of its children, and take the cheapest $\max(0, \text{degree}(i) - K)$ roads to close.

This sorting optimization will reduce the time complexity on computing the dp for each k to $O(N \log N)$. The running time of this solution is $O(N^2 \log N)$.

3.5 Subtask 5

Define H_k as a subgraph induced by vertices with degree $> k$. Each vertices i will appear exactly degree(i) times in this subgraph. Meaning any $O(N)$ or $O(N \log N)$ algorithms would pass for each subgraph H_k.

In the case of an unweighted tree, recursively take any leaf in H_k, and remove the edge from it to its parent. In the end the forest H_k will consist of singletons and we can just add degree(i) $- k$ for all such singletons i. This can be implemented using a depth-first search algorithm. The running time of this solution is $O(N \log N)$.

3.6 Subtask 6

We will apply the idea of subtask 4 to compute the cost in each connected trees in the forest H_k. Now, we must remove extra edges from this subgraph. Any edges that do not appear in this graph must be removed as well.

Define m as the number of extra edges, meaning the edges that don't exist in H_k but exist in the initial given tree. We need to add the cost of $(m - (k - c - |P|))$ cheapest edges for $dp[i][c]$. Overall, for each given k and a data structure to retrieve those edges we need $O(\alpha|H_k|)$ time. A simple bucket array can achieve $O(W|H_k|)$ and sufficient to solve this subtask. The running time of this solution is $O(NW \log N)$

3.7 Subtask 7

Following from the previous subtask, we need to further optimize and implement a good data structure. This data structure will be used to find the cheapest extra edges when computing $dp[i][c]$.

To be precise, we need to handle the sum query of least q values, for an arbitrary integer q, and to insert also delete values inside the container. One can use a balanced binary search tree like Treap, dynamic segment tree, or an offline fenwick tree. The running time of this solution is $O(N \log N)$.