APIO 2020 Problems Discussion

Gunawan, Jonathan Irvin
Google Asia Pacific Pte. Ltd.
`jonathanirvingunawan@gmail.com`

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

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

Nurrokhman, Abdul Malik
IA TOKI
`abdul1024malik@gmail.com`

Rifa'i, Wiwit
`wiwitrifai@gmail.com`

August 21, 2020

Below are the official solutions used by APIO 2020 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. Painting Walls

Problem Author
Written by: Wing-Kin Sung, modified by APIO 2020 Scientific Committee
Prepared by: Jonathan Irvin Gunawan
Solutions, review, and other problem preparations by: Abdul Malik Nurrokhman, Prabowo Djonatan

1.1 Subtask 2

We can check the validity of an instruction with a specific parameter x and y in $O(\sum_{k=0}^{K-1} f(k))$. This can be done by iterating the content of $B[(y+j) \bmod M]$ and check whether $C[x+j] \in B[(y+j) \bmod M]$, for $0 \le j < M$. The running time of this check is $O(\sum_{j=0}^{M-1} A[j])$, which is equal to $O(\sum_{k=0}^{K-1} f(k))$.

For each $0 \le i < N$, we can try all possible $0 \le j < M$ to check whether a valid instruction with $x = i$ exists. Therefore, we can check whether Pak Dengklek can paint all segments from i to $i+M-1$ (inclusive) with a single instruction in $O(M \times \sum_{k=0}^{K-1} f(k))$. Let $P[0..N-1]$ be an array of boolean such that $P[i]$ is whether Pak Dengklek can paint all segments from i to $i+M-1$ (inclusive) with a single instruction. We can fill the whole array in $O(N \times M \times \sum_{k=0}^{K-1} f(k))$.

Once we have the P array, we can solve the original problem using dynamic programming. Let $L[i]$ be the maximum value i' that satisfies $i' \le i$ and $P[i'] = \text{true}$, or $L[i] = -\infty$ if no such value exists. We can precompute this array in $O(N)$. Let $g(i)$ be the minimum number of instructions Pak Dengklek has to give to paint all segments from i to $N-1$ (inclusive). We can define $g(i)$ recursively as follows:

$$g(i) = \begin{cases} 0 & \text{if } i \ge N \\ \infty & \text{if } i - L[i] \ge M \\ 1 + g(L[i] + M) & \text{otherwise} \end{cases}$$

The answer to the problem is $g(0)$ and can be computed in $O(N)$. This solution runs in $O(N \times M \times \sum_{k=0}^{K-1} f(k))$ to fill the whole array P and is fast enough to solve subtask 2.

1.2 Subtask 3

To solve this subtask, we need to fill the whole array P faster. We need to precompute an additional 2D array of boolean $E[0..K-1][0..M-1]$ such that $E[k][j]$ is whether $k \in B[j]$. We can do so by iterating the values in $B[0..M-1]$ and updating the corresponding field in E. This takes $O(K \times M + \sum_{k=0}^{K-1} f(k))$.

Once we have E, we can check the validity of an instruction with a specific parameter x and y in $O(M)$, by checking the value of $\bigwedge_{j=0}^{M-1} E[C[x+j]][(y+j) \bmod M]$. Therefore, the value of $P[i]$ can now be computed in $O(M^2)$.

Therefore, this solution runs in $O(K \times M + \sum_{k=0}^{K-1} f(k) + N \times M^2)$, which is fast enough to solve subtask 3.

1.3 Subtask 1

To solve this subtask, we need to fill the whole array P even faster. If there exists a wall segment i where $f(C[i]) = 0$, then we will not be able to paint that wall segment. Therefore, the answer for this case is obviously -1. The rest of the discussion for this subtask assumes $f(C[i]) = 1$, for $0 \leq i < N$.

Since $f(C[i]) = 1$, there is exactly one j satisfying $C[i] \in B[j]$. Let $\overline{C}[i]$ be such value. Therefore, the value of $P[i]$ now becomes the value of $\bigwedge_{j=0}^{M-1} \overline{C}[i+j] = (\overline{C}[i] + j) \bmod M$. This solution runs in $O(N \times M)$.

We can optimize the computation of $P[i]$ by computing the rightmost index j such that $\overline{C}[i+j] = (\overline{C}[i] + j) \bmod M$ is still true. We can compute an array R in $O(N)$ as follows:

$$R[i] = \begin{cases} i & \text{if } i = N-1 \vee \overline{C}[i+1] \neq (\overline{C}[i]+1) \bmod M \\ R[i+1] & \text{otherwise} \end{cases}$$

Computing $P[i]$ can now be done as simple as checking whether $R[i] - i \geq M - 1$. This solution runs in $O(N)$.

1.4 Subtask 4

Since $f(C[i]) = 1$ is not guaranteed in this subtask, the value of $\overline{C}[i]$ is not defined anymore. Therefore, we will need to re-adjust our solution.

Let $Q[i][j]$ be the rightmost index k such that $C[i+k] \in B[(j+k) \bmod M]$ is still true. If $C[i] \notin B[j]$, we define $Q[i][j]$ to be $-\infty$. We can compute the array Q in $O(N \times M)$ as follows:

$$Q[i][j] = \begin{cases} -\infty & \text{if } C[i] \notin B[j] \\ i & \text{if } C[i] \in B[j] \wedge (i = N-1 \vee C[i+1] \notin B[(j+1) \bmod M]) \\ Q[i+1][(j+1) \bmod M] & \text{otherwise} \end{cases}$$

Similar to previous subtask, computing $P[i]$ can now be done by checking whether there exists $0 \leq j < M$ such that $Q[i][j] - i \geq M - 1$. This solution runs in $O(N \times M)$.

1.5 Subtask 5

Since $f(k) \geq 0$, we can show that $f(k) \leq \sqrt{\sum_{k=0}^{K-1} f(k)^2}$, which is bounded by 400 000. We can see that there are only $f(C[i])$ values of j such that $Q[i][j] \geq 0$. Therefore, we can optimize the computation of Q by not computing the value of $Q[i][j]$ where $C[i] \notin B[j]$.

Let $F[k]$ be the set of contractors j such that the j-th contractor likes the color k. Therefore, $|F[k]| = f(k)$. We can populate these sets by iterating through the whole B array once. After we have these sets, we can populate the values of $Q[N-1], Q[N-2], \ldots, Q[0]$. For each i, we can only iterate the values of $j \in F[C[i]]$, and then compute the value of $Q[i][j]$. To optimize space usage, we will need to discard all values of $Q[i+1]$ once we have computed all values of $Q[i]$ since they will not be needed anymore.

This solution runs in $O\left(N \times \sqrt{\sum_{k=0}^{K-1} f(k)^2}\right)$.

2 B. Swapping Cities

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

For the ease of discussion, we use graph terminology. Each city can be represented as a node, and each road can be represented as an edge. The i-th edge connects node $U[i]$ and node $V[i]$ with weight $= W[i]$. Our goal is to swap the positions of cars from node $X[i]$ and node $Y[i]$ without meeting at any point in time, and we want to minimize the maximum weight of the used edges.

2.1 Subtask 1

Each node has degree no more than 2, so there are only 2 possibilities, i.e. either the edges form a cycle or a linear path. This is because if we follow the edges in one direction, then it is either we will come back to the original node or arrive at a node whose degree is 1.

If the edges form a cycle, then we can swap both cars by using all edges. This is because there are 2 different paths connecting node $X[i]$ and $Y[i]$, so each car can use a different path. If the edges form a linear path, then we cannot swap both cars, since both cars will block each other from arriving at their destinations at the same time.

So, if the edges form a cycle ($M = N$) then the answer is $\max_{i=0}^{M-1} W[i]$, otherwise the answer is -1.

2.2 Subtask 2

The graph of this subtask is a star. If $M \leq 2$, then the edges also form a linear path, so it is impossible to swap both cars without meeting at any point. But if $M \geq 3$, then we can swap both cars.

We will show that we can do the swapping using only 3 edges which meet at one node. Let's say we will use 3 edges that meet at node 0, and the other nodes connected to the edges are node A, node B, and node C. WLOG, we want to swap cars from node A and B. First, we can move car from A to C. Then, we move car from B to A. Finally, we move car from C to B. If we want to swap cars from node 0 and A, then we can move car from 0 to B first and then we can do the same thing as before.

So, we need to use only 3 edges with weight as minimum as possible. But, we also need to use the edges on the path from node $X[i]$ to $Y[i]$. So, the answer is the maximum between the third lowest weight and the maximum weight of edges on the path from node $X[i]$ to $Y[i]$.

2.3 Subtask 3

To solve this subtask, we can use an algorithm similar to the Dijkstra's algorithm. Let $cost[A, B]$ be the minimum of the maximum weight of used edges to move car from node $X[i]$ to A and move car from node $Y[i]$ to B without meeting at any point. So, the answer will be $cost[Y[i], X[i]]$.

Initially, we can assign $cost[X[i], Y[i]] = 0$ and $cost[A, B] = \infty$ if $A \neq X[i]$ or $B \neq Y[i]$. Then, we can use algorithm similar to Dijkstra's algorithm to update $cost[A, B]$ for all $0 \leq A, B < N$. If we have already got the value of $cost[A, B]$ then we can try to move car from node A to every adjacent node which is not the same as node B, i.e. if there is an edge connecting A and C with weight $= D$ and $C \neq B$, then we can update the value of $cost[C, B] = min(cost[C, B], max(cost[A, B], D))$. We also can try to move car from node B to every adjacent node which is not the same as node A using the similar method. After we have computed $cost[A, B]$ for all $0 \leq A, B < N$, then the answer is $cost[Y[i], X[i]]$ if $cost[Y[i], X[i]] \neq \infty$, otherwise the answer is -1.

This solution runs in $O(Q \times N \times (N + M) \times \log(N \times N))$.

2.4 Subtask 4

Based on the observations from subtask 1 and subtask 2, we can say that the only condition so that we cannot swap both cars is when the used edges form a linear path. In other words, we can swap both cars from node $X[i]$ and $Y[i]$ if the used edges connecting both nodes, and at least one of the below conditions is satisfied:

- The used edges form a cycle.
- There are at least 3 used edges that meet at one node.

To solve this subtask, we can use algorithm similar to Kruskal's algorithm for Minimum Spanning Tree. First, we need to sort the edges based on the weight. Then, we can insert the edge one by one from the lowest weight to the highest weight. After we insert an edge, we should check whether the conditions of the swapping have been satisfied or not. If the conditions have been satisfied, then we stop the process and the answer is the weight of the last inserted edge. If all edges have been inserted but the conditions are not satisfied, then the answer is -1.

To make the conditions checking fast, we can use data structure such as Disjoint Set Union (DSU). The DSU will maintain the connected components and boolean flags $swappable[c]$ representing whether component c contains a cycle or a node with degree ≥ 3. Initially, each node has its own component and there is no edge. Then, we insert the edges one by one. Let's say we insert an edge connecting node A and node B to the DSU. If A and B are on the same component, then that component will contain a cycle and we set $swappable[c]$ as true. Otherwise, we need to merge component containing node A (component a) and component containing node B (component b) as a component c, and set $swappable[c]$ as true if and only if $swappable[a]$ is true, or $swappable[b]$ is true, or the new degree of node $A \geq 3$, or the new degree of node $B \geq 3$. If after inserting an edge, node $X[i]$ and $Y[i]$ are on the same component and that component has $swappable[c]$ as true, then we stop the process and the answer is the weight of the last inserted edge.

This solution runs in $O(M \times \log(M) + Q \times M \times \alpha(N))$ where $\alpha(N)$ is the Inverse-Ackermann function from the DSU.

2.5 Subtask 5

The graph of this subtask is a tree. Since the graph has no cycle, then we have to use all edges on the path from $X[i]$ to $Y[i]$ and use some additional edges so that there is at least one node with degree ≥ 3. If there is no node with degree ≥ 3, then the answer is -1.

Let node 0 be the root of the tree. To calculate the maximum weight on the path from node $X[i]$ to node $Y[i]$, we can precompute $ancestor[i][j]$ and $max\_weight[i][j]$ for $0 \leq i < N$ and $0 \leq j \leq \log_2(N)$. The value of $ancestor[i][j]$ is the 2^j-th ancestor of the node i and $max\_weight[i][j]$ is the maximum weight of the edges from node i to the 2^j-th ancestor of the node i. We can calculate those precomputed values in

$O(N \times \log(N))$. Then with those precomputed values, we can find the Lowest Common Ancestor (LCA) of node $X[i]$ and node $Y[i]$, and also calculate the maximum weight on the path from node $X[i]$ and node $Y[i]$ to their LCA in $O(\log(N))$.

Since node $X[i]$ and $Y[i]$ are already connected, then we just need to add edges so that there is a node with degree ≥ 3. To calculate the additional edges, we can use dynamic programming. Let $cost[x]$ be the minimum of the maximum weight of the used edges so that node x is connected to a node with degree ≥ 3. Initially, $cost[x]$ is the third lowest weight of edges connected directly to node x if the degree of node x is ≥ 3, otherwise $cost[x] = \infty$. Then, if there is an edge connecting node A and node B with weight C, we can update $cost[A] = min(cost[A], max(cost[B], C))$ and $cost[B] = min(cost[B], max(cost[A], C))$. We can calculate $cost[x]$ for all $0 \leq x < N$ in $O(N)$. To do that, we can update the value in the parent node from each child starting from the deepest leaves to the root. And then, we can spread the value in the parent node to update each child starting from the root to the deepest leaves.

Finally, the answer is maximum between $cost[X[i]]$ and the maximum weight of edges on the path from node $X[i]$ to node $Y[i]$. Each query can be answered in $O(\log(N))$. So, this solution runs in $O(N \times log(N) + Q \times \log(N))$.

2.6 Subtask 6

To get a full score, we can develop a solution based on the solution of subtask 4. In subtask 4, we basically do the same thing for each query, i.e. we insert the edges one by one from the lowest weight to the highest weight while maintaining the connected components and the properties of each component. We can improve this by building a tree where each node represents a component in the DSU. Initially, there are only N nodes in the tree representing the initial components in the DSU. And then, we will create a new node representing the updated component when we insert an edge to the DSU. If we merge 2 components in the DSU, then the new node in tree will become the parent of the nodes representing both components. But if we insert an edge connecting nodes from the same component, then the new node will only have one child that is the node representing the component before inserting the new edge. So in the end, there will be $N + M$ nodes in the tree. While building the tree, we also need to save the properties of the components from the DSU to each node, i.e. we need to save the boolean flags $swappable[c]$ and the weight of the last inserted edge to each component.

Each query can be answered by finding the LCA of node $X[i]$ and node $Y[i]$ in the tree. And then, we can find the lowest ancestor of LCA($X[i]$, $Y[i]$) having $swappable[c]$ as true. Finally, the answer is the weight of the last inserted edge to the component represented by that node. We can use the same method as mentioned in subtask 5 to find the LCA and the lowest ancestor that $swappable[c]$ is true. So overall, this solution runs in $O((N + M) \times \log(N + M) + Q \times log(N + M))$

3 C. Fun Tour

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

3.1 Subtask 1

For the ease of discussion, define $dist(i, j)$ as the time required to go from the i-th attraction to the j-th attraction.

This subtask can be solved using dynamic programming with bitmask. Let $dp[mask][i]$ be the maximum last travel time for a fun tour that ends with the i-th attraction and visits j-th attraction if and only if j-th bit in $mask$ is on. In order to get the value of $dp[mask][i]$, we can iterate all active bits in $mask$. The value of $dp[mask][i]$ is the maximum of $dist(i, j)$ for all $0 \leq j < N$ such that j-th bit in $mask$ is active and $dist(i, j) \leq dp[mask - 2^i][j]$. If there is no j that meets the conditions, then $dp[mask][i] = 0$. We can set $dp[2^i][i] = \infty$ for $0 \leq i < N$ as our base case.

Once we have all the value of $dp[mask][i]$, we can create a fun tour by backtracking from any S such that $dp[2^N - 1][S] \neq 0$. In other words, there is a complete fun tour that ends in the S-th attraction. We also need to precompute all $dist(i, j)$ so that we only need to ask $\frac{N \times (N-1)}{2}$ queries. This solution runs in $O(2^N \times N^2)$.

3.2 Subtask 2

In this subtask, we also need to ask all $dist(i, j)$ for $0 \leq i < j < N$. We can start a fun tour from the S-th attraction such that $dist(0, S) \geq dist(0, i)$ for $0 < i < N$. After that, we can visit the farthest unvisited attraction from the attraction we are currently in. Formally, when we are currently in i-th attraction, then visit the j-th attraction that has the maximum $dist(i, j)$ among all attractions that are not visited yet. This solution runs in $O(N^2)$.

3.3 Subtask 3

For the ease of discussion, we will use graph terminology. Let's represent the theme park as a tree where its nodes represent the attractions and its edges represent the roads.

For this subtask, the tree structure can be derived from the constraint. The tree is a full binary tree rooted at node 0. We can create a fun tour using a similar way as subtask 2, but to find the farthest node, we should use another approach. For each node, we need to store all nodes in its subtree sorted by distance to root. It can be done in $O(N \times \log(N))$ time and $O(N \times \log(N))$ memory. After that, we can get the farthest node from some node by iterating all ancestors of it. Once we visited the farthest node, update all of its ancestors by removing the visited node. Since the depth of the tree is $O(\log(N))$, the total complexity of this solution is $O(N \times \log(N))$.

3.4 Subtask 4

In this subtask, it can be shown that the tree is a binary search tree. In order to construct the binary search tree, we have to know all $dist(0, i)$ and $dist(i, N-1)$, for $0 \leq i < N$. Let node R be the root of the tree. We say node i is a special node if and only if $dist(0, i) + dist(i, N-1) = dist(0, N-1)$. In other words, the node is located on a path between node 0 and node $N-1$. Root node R can be chosen from any special node that fulfill the following requirements. For each special node i, if there exist another node j such that $dist(0, i) + 1 = dist(0, j)$ and $dist(i, N-1) + 1 = dist(j, N-1)$, then $i < R$ if $i < j$, otherwise $i > R$ if $i > j$.

Once we have the root of the tree, we can construct binary search tree by inserting nodes one by one starting from the node i that has the smallest $dist(0, i) + dist(i, N-1)$. In case of a tie, start from the smallest $|R - i|$. After that, we can create a fun tour using the same way as subtask 3 solution. This solution runs in $O(N \times \log(N))$ time and needs $2N - 3$ queries.

3.5 Subtask 5

For the ease of discussion, define $subtree(i)$ as the subtree size below node i with a fixed root. If the fixed root is node 0, then $subtree(i) = attractionBehind(0, i)$.

To get a full score, we do not need the complete map of the attractions. We just need to know the centroid node so that we can separate the tree into two or three parts with size not larger than $\frac{N}{2}$. To get that information we can ask $subtree(i)$ for $0 < i < N$. The centroid of the tree will be a node i that has the minimum $subtree(i)$ but $subtree(i) \geq \frac{N}{2}$.

Once we know the centroid node, we can separate the tree into two or three parts by deleting that centroid node. Let node R be the centroid node of the tree. Then, we can ask $dist(R, i)$ for $0 \leq i < N, i \neq R$. Let some node P is any node that satisfy $dist(R, P) = 1$. Node i will be in the same part as node P if and only if $dist(R, i) = dist(P, i) + 1$. So, we need to check $dist(P, i)$ for some valid node P. Since the number of valid P is no more than three, if a node i is not in the same part as two valid nodes P, then it must belong to the remaining one (no need to check the last valid node P). This observation makes the number of queries needed in the worst case is no more than $4N$.

Now, we have already separated the tree into some parts, and know for each node which part it belongs to. If there are three parts, create a fun tour starting from the farthest node from the root. After that, add the farthest node that is not in the same part as the last node in the current fun tour. Repeat this step until there is a part such that the number of nodes left in it is equal to the total number of nodes left in the other two parts. Let's call such part as a big part and the others are small parts. Combine two small parts into one combined part. Now, we have two parts with equal size. If we have only two parts to begin with, we can skip this step and proceed from here.

Since the number of nodes in both parts are equal, we can finish the fun tour by alternating farthest nodes in both parts. To determine from which part we should start to alternate, if there are only two parts since we separate the tree then start from the bigger one. Otherwise, it is obvious that the last node in the current fun tour is from one of the small parts. If there is a node in the other small part which is farther than the last node, then we should start from the combined part, otherwise start from the big part. Lastly, include node R as our last node for the fun tour.

This full solution runs in $O(N \times \log(N))$ and needs $4N - 10$ queries.