培正 喇沙 編程挑戰賽
LA SALLE – PUI CHING
PROGRAMMING CHALLENGE

第七屆 · 2023

2023 年 8 月 19 日（星期六）

香港培正中學

題解

題號		名稱	作者	編製	難易度
A	🟦	Always Right	吳有孚	黃文禮	★★★☆☆
B	🟪	Binary Bracket	謝凌睿	謝凌睿	★★★☆☆
C	🟧	Cross Across the Grid	袁樂勤	梁皓寧	★★☆☆☆
D	🟩	Decisive Duels	謝凌睿	黃正諾	★★★★☆
E	🟦	Enthusiast of Algorithms	龍瑞希	龍瑞希	★★☆☆☆
F	⬛	Far-reaching Citations	葉景輝	葉景輝	★★★★★★
G	🟥	GPT Intrusion	楊汶璁	鄭希哲	★☆☆☆☆
H	🟦	Handful of Balls	招朗軒	招朗軒	★★★★★
I	🟩	Ideal Cutting	衛家熙	衛家熙	★★★★☆
J	🟥	Joining Two Trees	袁樂勤	袁樂勤	★★★★★
K	🟨	Keep Them Stacked	吳有孚	鍾懷哲	★☆☆☆☆
L	🟧	Lift Problem	楊汶璁	黃梓謙	★★★★☆

難易度是指作者團隊認為能在比賽中解決該題的隊伍比例

由 1 星至 6 星為: >75%, 50-75%, 25-50%, 5-25%, 1-5%, <1%

A - Always Right

We may view the maze as a directed graph. Since we have to consider the direction we're facing, the graph will contain $4NM$ nodes with each node representing a cell and one of the four directions.

Let's denote the node representing the cell in row x, column y and direction d (`UDLR`) as $A[x][y][d]$. For each node, construct directed edges according to the two moves: (only construct the edge if both nodes are representing empty cells)

- d is `U`, build edges to $A[x-1][y][\texttt{U}]$ and $A[x-1][y][\texttt{R}]$
- d is `D`, build edges to $A[x+1][y][\texttt{D}]$ and $A[x+1][y][\texttt{L}]$
- d is `L`, build edges to $A[x][y-1][\texttt{L}]$ and $A[x][y-1][\texttt{U}]$
- d is `R`, build edges to $A[x][y+1][\texttt{R}]$ and $A[x][y+1][\texttt{D}]$

Finally, perform BFS on the graph to find the shortest solution, or determine if it is impossible.

Time complexity: $O(NM)$

B - Binary Bracket

It is easy to notice that the answer for any player is bounded by K, since the any match not involving this player can be not an upset, and this player we are solving for only plays K matches.

We try to approach the problem with a greedy intuition: For each opponent in each round faced by our player, what is their minimum possible “power index” given no upsets have happened in their sub-bracket.

We can calculate this using tree DP in O(2^K * K), in each node we store **all** of the possible “power index” reachable to this position without any upsets, transition is linear in terms of subtree size by considering if each of the “power index” reachable in either subtree can win the match without upsetting the weakest possible opponent in the other subtree. The total complexity (time and space) per layer is O(2^K) and with K layers the total complexity is O(2^K * K).

Now with the computed values of each node, we can simply iterate through each player and count the number of nodes on its way to the root (winning) that is greater than their “power index” + upset threshold. This step is also O(2^K * K) which is also out final time complexity.

C - Cross Across the Grid

Introduction:
The task is to rotate layers of a given N*N grid to align diagonal characters with the central character of the grid. The grid size N is always an odd number ensuring a distinct center. Since a solution is guaranteed with the given input, we aim to find the minimum number of rotations needed, considering both clockwise and anti-clockwise rotations for each layer.

Procedure:

1. Identify the center character of the grid using the relation center_position = N/2.
2. Iterate through each layer, starting from the outermost layer, moving towards the center.
3. For each layer, calculate the minimum number of rotations needed to align the 4 corners with the center character:
   For clockwise rotation: Rotate each character position on the layer until the diagonal characters match the center character. Keep a count of the number of rotations needed.
   For anti-clockwise rotation: Similarly, rotate each character position on the layer in an anti-clockwise direction, keeping a count of the number of rotations needed. (Or you can simply rotate the clockwise and calculate the rotation that anti-clockwise needs)

4. Compare the counts of clockwise and anti-clockwise rotations, and choose the direction that requires fewer rotations for each layer.
5. Sum up the minimum rotations from all layers to obtain the total minimum rotations required for the entire grid.

Time Complexity Analysis:
The time complexity is derived from three nested loops:

Looping through layers: O(N)
Looping through positions in a layer for rotation: O(N)
Checking and rotating positions: O(N)
Combining these, the total time complexity is O(N^3).

D - Decisive Duels

First of all, it would be helpful to find a way to determine whether David could win before any modifications to the substring. One way to do that is to save the number of ‘1’s in a substring using partial sum, as well as the difference of score between the players from the start to a particular position. Using those respectively, we can use binary search to find the first position where David gets **k** points, and then starting from that position, use a range maximum data structure and another binary search to find the first position where David scores 2 points more than his opponent. This is the first position where David fulfills the criteria to win the match. Using the same method, we can do the same for his opponent, then all we have to do is to compare who fulfills the criteria to win the match first to see who will win the match. The time complexity for this method would be O(log N).

Putting that aside, one intuitive but useful observation is that when inserting ‘1’s, we should always put it onto the beginning of the substring. (Notice that moving a ‘1’ to the front will neither delay the position at which David gets **k** points, nor affect the difference of score between the players on and after the original position of the moved ‘1’)

Building upon this, we can binary search on the amount of ‘1’s added to the front of the string. (Note that the maximum amount is bounded by **k** + 2) We can check if David will win using a method similar to the one described above by just adding some compensations before doing the binary searches. The final time complexity would be O(N log N + Q log N log K).

Bonus

There also exist solutions which run in linear or linearithmic time, however they aren’t required to pass the time limits for this task.

E - Enthusiast of Algorithms

Let $f(M)$ be the number of days Bob can learn without violating the learning rules. We know that,

$$f(M) = \sum_{i=1}^{N} \left\lfloor \frac{a_i}{M} \right\rfloor$$

When M increases, it is clear that $\frac{1}{M}$ will decrease and lead to the decrease of each $\frac{a_i}{M}$ also. As we round down each $\frac{a_i}{M}$ in the calculation, it is possible that the result of $\left\lfloor \frac{a_i}{M+1} \right\rfloor$ remains the same as $\left\lfloor \frac{a_i}{M} \right\rfloor$. Therefore, we can conclude that,

$$\left\lfloor \frac{a_i}{M} \right\rfloor \geq \left\lfloor \frac{a_i}{M+1} \right\rfloor \; \forall\, i \in [1, N]$$

And thus,

$$\sum_{i=1}^{N} \left\lfloor \frac{a_i}{M} \right\rfloor \geq \sum_{i=1}^{N} \left\lfloor \frac{a_i}{M+1} \right\rfloor$$

Which shows the monotonicity of $f(M)$.

Based on this observation, we can come up with a solution using binary search. Let's binary search on the value M and calculate $f(mid)$ in each iteration.

If this value is larger than or equal to K, it means that $M = mid$ is valid. As there might be a better solution, we will mark M and move the left bound to $mid + 1$.

If this value is smaller than K, it means that $M = mid$ is invalid, we will move the right bound to $mid - 1$.

The latest marked value will be the final answer. Let G be the maximum value of a_i, the time complexity would be $O(NlogG)$.

It is guaranteed that $\sum a_i \geq M$, the solution always exists, as the worst case $M = 1$ must be valid.

F - Far-reaching Citations

July 23, 2024

A Simplified Case

First, let's consider a simplified case where $N = 1$. This means we have two strings, S and T. Our goal is to count the sum of the number of occurrences of each substring of T in S.

Example

Suppose $S = \text{aba}$ and $T = \text{ab}$.

$$\begin{aligned}\text{Answer} &= \text{Occurrences of "a" in S} + \text{Occurrences of "ab" in S} + \text{Occurrences of "b" in S} \\ &= 2 + 1 + 1 = 4\end{aligned}$$

Suffix Automaton (SAM)

This problem is rather standard application of Suffix Automaton (SAM). For a detailed introduction, refer to `https://cp-algorithms.com/string/suffix-automaton.html`.

Here are some key definitions from the tutorial:

- endpos(state) or endpos(str): The end positions associated with a state or a string.
- link(state): The suffix link or suffix link of a state.
- len(state): The length of the longest string associated with a state.

Notable Properties of SAM

We first highlight three notable properties of SAM, we will use: For a given state K:

1. The size of endpos(S) equals the number of occurrences of the string corresponding to K.
2. State S represents strings of length len(link(K)) + 1 to len(K).
3. Considering the path on the suffix link tree from K to the root, the union of the lengths they represent is $[0, \text{len}(K)]$.

Calculating Occurrences

To calculate the total number of occurrences of suffixes (not substrings!) of T:

1. Let O be the state corresponding to string T.
2. Let $O = V[1] \to V[2] \to ... \to V[N] = \text{Root}$ be the path from O to the root using suffix links.

Using the identified properties:

$$\text{Occurrences} = \sum(\text{len}(V[i]) - \text{len}(V[i+1])) \times |\text{endpos}(V[i])|$$

Considering All Substrings

To handle all substrings, a simple algorithm is to invoke the suffix-based algorithms for all prefixes of T. However, this approach is quadratic in time complexity.

To optimize, notice:

1. For an arbitrary state $V[i]$ on the identified path, notice that $V[i+1] = \text{link}(V[i])$.
2. The contribution of $V[i]$ to the final answer is constant, regardless of the path from O to the root.
3. Therefore, we can use dynamic programming to compute the total number of times each node $V[i]$ is used, resulting in a linear time algorithm.

One Caveat

What if T is not the constructed SAM of S? We leave this as an exercise for the reader. (Hint: Use the suffix link.)

General Case

For the general case when $N > 1$, we extend the SAM data structure to handle all suffixes of multiple strings. The dynamic programming algorithm remains the same.

Two Approaches

1. **General SAM**: Extend SAM by constructing it based on the BFS traversal of a Trie. Refer to "General suffix automaton construction" for more details.
2. **Concatenate Strings**: Concatenate all strings $S[1], S[2], ...S[N]$ with a delimiter '#', and construct the SAM on the concatenated string.

G - GPT Intrusion

To determine the answer, we should first read the whole input as a single string, and then check if the string satisfies the conditions given in the problem statement.

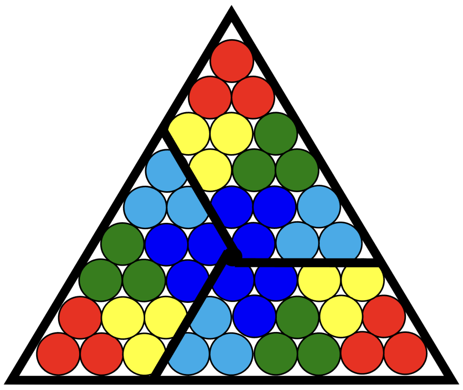
To store the whole input into a single string, we can read the input line-by-line, append an endline character to each of the lines, and concatenate the lines. Note that since the input may contain spaces, we should read the input by line like "std::getline(std::cin, s)" instead of reading the input by word like "std::cin >> s".

To check if the input satisfies the conditions, we can first check if it has more than 500 characters. If so, we can delete the first 500 characters, and check if it is equal to the string "As an AI model, my output is limited to 500 characters.\n", where "\n" represents the end of line character.

H - Handful of Balls

It is easy to see that $N \equiv 0$ or $2 \pmod 3$ is a necessary condition for a desired *3-ball-triangle tiling* to exist, since the total number of balls must be divisible by 3. But it is also obvious that this is not sufficient, as is in the cases of $N = 3$ and 5.

The next possible side-length after $N = 2$ is $N = 9$, as shown in the figure on the right. It is unique under rotation and reflection; the configuration itself also poses rotational symmetry. Constructing it should not be too difficult: start by putting upward triangles at all three corners of the rack, and construct the remaining from the top. All moves are then forced.



Another key observation is that we can use K downward triangles and $(K + 1)$ upward triangles to:

- Extend any length-$(2K - 1)$ rack to a length-$(2K + 2)$ rack; or
- Extend any length-$3K$ rack to a length-$(3K + 2)$ rack

With this observation we can easily construct the next possible configurations by $9 \to 11 \to 14$ and $9 \to 12 \to 14$. The final result is that, it is possible to construct a desired triangle tiling if and only if $N \equiv 0, 2, 9$ or $11 \pmod{12}$ (A072065 - OEIS), through the following recursive structure:

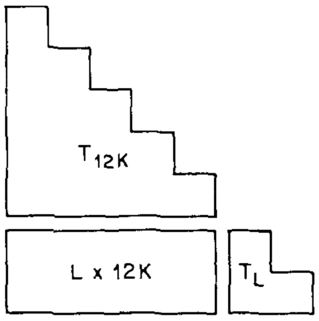


FIG. 3.4. Tiling of T_{12K+L} for $L = 2, 9, 11, 12$.

Aftermath: This is a constructive task, contestants are expected to solve by building on pattern observations. ***Especially in this question***, it is never intended for any contestant to rigorously prove any results within the contest timeframe; in fact, the proof is nowhere near elementary. This task is the result of John H. Conway and Jeffrey C. Lagarias in the 1990 article *Tiling with polyominoes and combinatorial group theory*. Inside the article, they also proved that it is **impossible** to solve the triangle tiling problem by a generalized **coloring argument**.

I - Ideal Cutting

The key observation of this problem is that no matter how you cut the polygon, the mean of the areas of triangles does not change, as the sum of the areas is equal to the area of the polygon and the number of triangles is fixed. Then, we can use dynamic programming to calculate the minimum variance directly.

Let $dp_{i,j}$ be the minimum variance of the sub-polygon containing the vertices i, i+1, …, j. We can enumerate k – the vertex we choose to cut a triangle out. The triangle is formed by the vertices i, j, and k. It breaks the remaining part into two sub-polygons containing vertices i, i+1, …, k and k, k+1, …, j respectively. We can then solve the problem recursively and use the values of $dp_{i,k}$ and $dp_{k,j}$ to calculate the answer. The final answer is $dp_{1,N}$ and the time complexity is $O(N^3)$.

J - Joining Two Trees

Suppose the diameter length of trees are D1, D2, where D1 <= D2 and the diameter length of tree T is D. D is either =D2 or >D2.

For each node x in tree T1, T2, find (d = farthest distance from x, f = the number of occurrences of farthest nodes). This can be done by doing DFS on both trees.

Maintain two sorted lists on (d, f), and perform two pointers on such lists: exhaust the pairs in the list for T1 and maintain a pointer on the list for T2.

Case 1: D = d_T1 + d_T2 + 1 > D2.
The new diameter must pass through the newly added edge. The number of required pairs = f_T1 * f_T2.

Case 2: D = d_T1 + d_T2 + 1 = D2.
The new diameter can pass through the newly added edge or be completely within tree T2 (cannot be within T1, because that only happens when D1 = D2 but it will lead to min(d_T1 + d_T2) + 1 > D2). The number of required pairs = f_T1 * f_T2 + (number of pairs of diameter in T2 originally)

Case 3: d_T1 + d_T2 + 1 < D2 = D
The newly added edge and nodes from T1 have no effect on the answer. The number of required pairs = (number of pairs of diameter in T2 originally)

Fixing a pair (d, f) in T1, you can quickly find the maximum answer that can be attained by choosing a pair (d, f) in T2 for each case. Take the maximum of all these answers you have.

K - Keep them Stacked

We could observe that putting all these papers aligned with their bottom left corners actually gives the optimal answer: If we couldn't cover some of the parts of the papers in this position, then moving some of it will not be able to cover them as well. With this observation, we could use a 2D array to simulate the area and count how many grids are occupied after filling those 3 papers starting with their bottom left corner separately having a time complexity of O(H * W).

But as the length is small enough, we could exhaust every starting position for the 2nd and 3rd paper, in that case the time complexity is $O((H * W)^2)$ by counting with math formulas instead of 2D array simulation.

L — Lift Problem

Solution

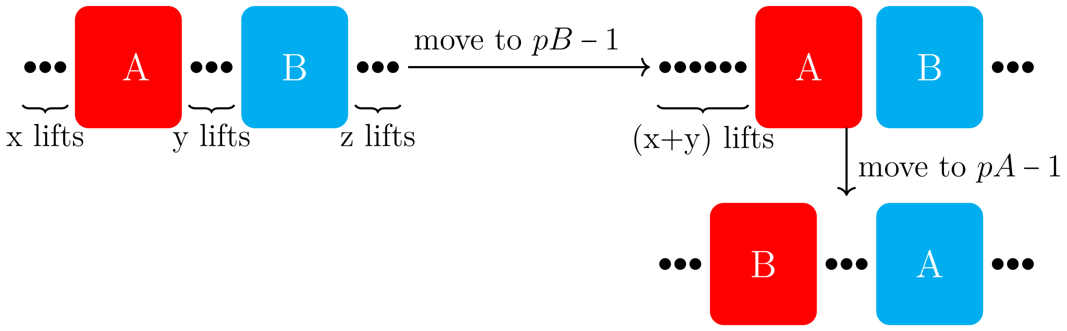
As we can shuffle a permutation into any permutation within N swaps, we can shuffle the lifts into the permutation P if we can swap any two lifts by taking at most 5 rides.

Now, let's consider that you would like to swap lift A and lift B when you are currently in lift C, where pA, pB, and pC are the original waiting floors of lifts A, B, and C respectively. Without loss of generality, we can assume that the original waiting floor of lift A is lower than lift B (i.e. $pA < pB$).

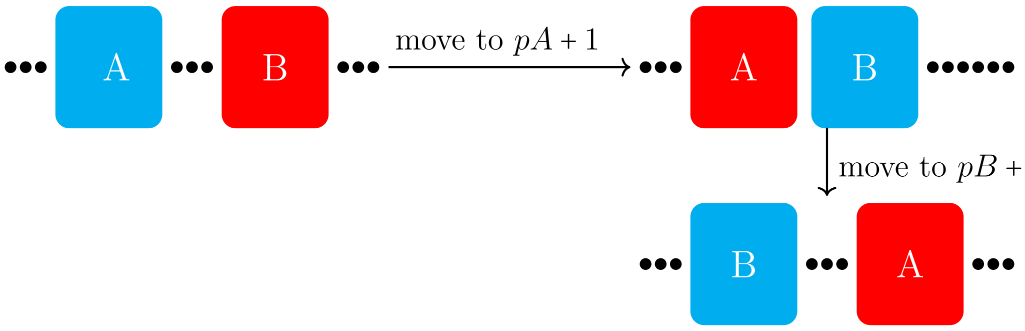
Then, we will consider the four cases with illustrations. In the below illustration, the red rectangle represents the lift that you are currently in after each move.

Case 1: $A = C$ or $B = C$

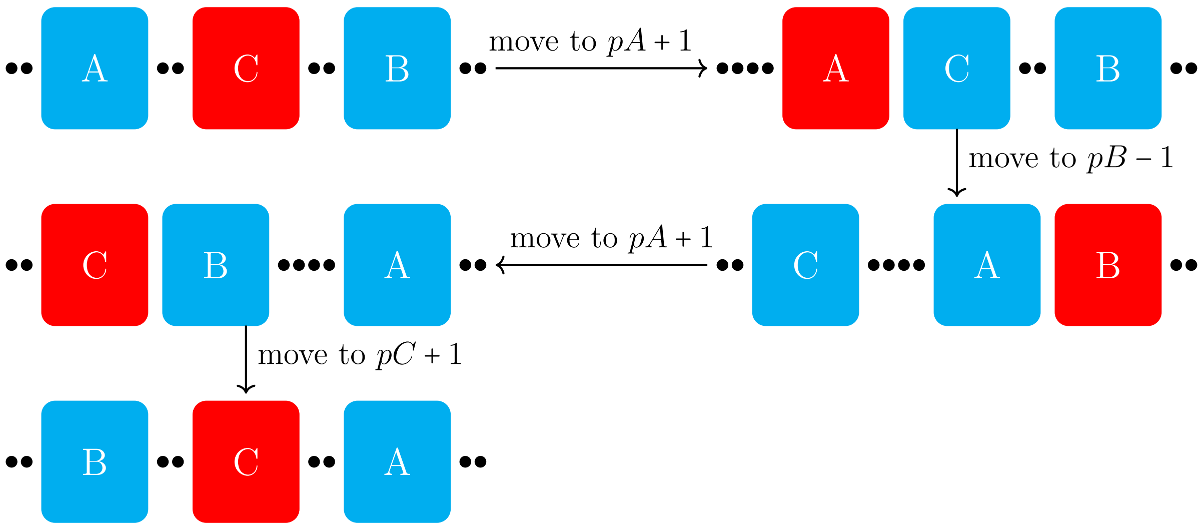
For $A = C$,



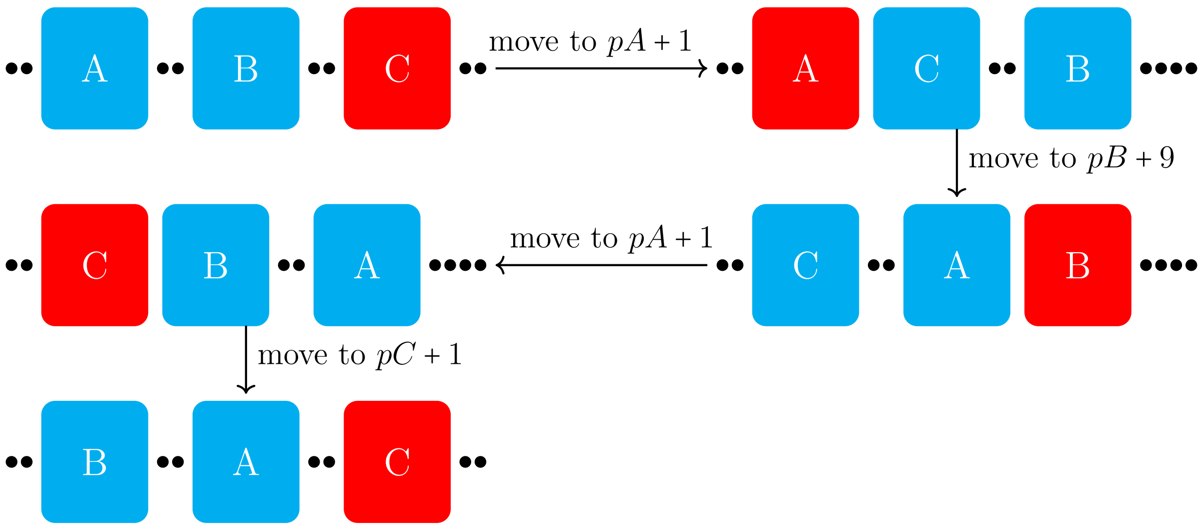
For $B = C$,



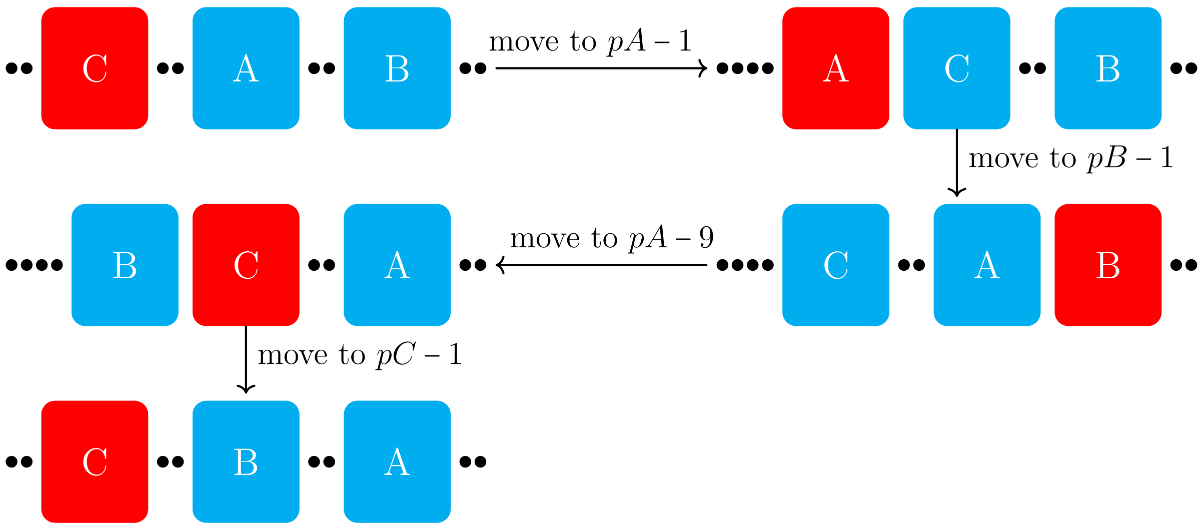
Case 2: Lift C is between lift A and B.



Case 3: Lift C is higher than lift B.



Case 4: Lift C is lower than lift A.



With the above approach, we can swap any two lifts within 4 moves. Therefore, the lifts can be reordered into the permutation P by taking at most $4N$ lift rides.

Bonus: It is possible to shuffle the lifts into the permutation P within $3N$ moves by using an insertion sort approach.