Kick Start 2022 - Round H

Analysis: Level Design

We have been given a permutation $\mathbf{P}$ of length $\mathbf{N}$ and are asked to find minimum swaps required to form at least one permutation cycle of length K, for all $1 \le K \le \mathbf{N}$. Here are a few steps required to solve this problem.

- Step 1: Find the existing permutation cycles.

  Considering permutation cycle as a graph, perform DFS from each unvisited node and keep traversing till we get a visited node. Maintain the cycle length for each DFS. At the end of this we have an array representing cycle lengths of the permutation cycles. Alternative approach is to use DSU to find all cycles and respective lengths.
  Let us name the cycle size array as cycle_sizes.

- Step 2: Analyze the effect of a swap in the permutation.

  Two types of swaps are possible:

  - Type 1: Swap within the same cycle, the cycle will be broken into two smaller cycles.
  - Type 2: Swap two elements from different cycles, both the cycles will end up merging. The length of the new cycle will be the sum of lengths of two cycles merged.

- Step 3: Optimal merging for each cycle length.

  Now we know the existing cycle lengths and the effect of a swap, we need to find the optimal strategy to form a cycle of length K in the minimum number of swaps.
  Now the problem essentially boils down to optimally selecting the cycles to merge/break to form a cycle of length K. There are two potential strategies:
  1) We can sort the cycle lengths in descending order by length and then merge the cycles starting from the maximum one till we get a final cycle of length $> K$ and then use one more swap to turn that into exactly K. The number of steps required here is an upper bound on the optimal number of swaps and the lower bound will be at least one less than the upper bound (forming cycle of exact length K). Time complexity of greedy approach is $O(\mathbf{N})$ since sorting the permutation cycles itself will take $O(\mathbf{N})$ using Counting sort.
  2) We can apply dynamic programming to optimally find minimum number of cycles required to form a cycle of size exactly K.
  The optimal strategy is to choose the minimum of the two strategies.

Test Set 1

For each of the cycles we do have the option either to include it in the optimal set of cycles to be merged or leave it. This problem is now very similar to the knapsack problem.
Let $dp(i, j)$ be the minimum number of swaps required to form a cycle of length j using cycles from indices $(1, i)$.
$dp(i, j) = \min(dp(i-1, j), 1 + dp(i-1, j - \text{cycle\_sizes(i)}))$
$S = \text{cycle\_sizes.size()}$.
We need to populate the dp table with the above mentioned recurrence relation. $dp(S, K)$ is the answer from the knapsack approach and $\min(1 + dp(S, t))$ for all $K + 1 \le t \le \mathbf{N}$ is the answer from greedy approach. Final answer will be minimum of both the values.
Time complexity will be $O(\mathbf{N}^2)$ to populate the dp table.

Test Set 2

Let X be the number of length wise distinct cycles, minimum number of nodes required to form all the X cycles is $\sum_{i=1}^{X} i$. As per the constraints, $\sum_{i=1}^{X} i \le \mathbf{N}$.
Hence there will be $O(\sqrt{\mathbf{N}})$ number of length wise distinct cycles. Using this fact we can speed up our knapsack-based solution from test set 1.
Let us name the distinct cycle size array as $\text{distinct\_cycle\_sizes}$ and corresponding occurrences as cnt. Let $dp(i, j)$ be the minimum number of swaps required to form a cycle of length j using cycles from indices $(1, i)$ where $1 \le i \le \text{distinct\_cycle\_sizes.size()}$.

Hence, in order to calculate $dp(i, j)$ (where i is the index of the current value x being processed and j is the cost), we only need these states $dp(i-1, j-x), dp(i-1, j-2 \times x), \ldots, dp(i-1, j - cnt(i) \times x)$. For each distinct cycle size, store its number of occurrences and then the idea to process v(occurrences of each cycle) items of value x is to treat the problem as a sliding window minimum problem.

Hence, the recurrence becomes $dp(i, j) = \min(v + dp(i-1, j - v \times \text{distinct\_cycle\_sizes}[i]), v \in [0, cnt(i)]$

Then, if we consider the values $j \mod k$, you will notice that for a fixed remainder it just becomes a range min query on a 'sliding window' interval (namely, the left bound of the interval may only move to the right each query), which can be computed in amortized constant time using a monotonic deque.

So, now for each $i \in [1, \mathbf{N}]$, the optimal answer $= \min(dp(\text{items\_count}, i), dp(\text{items\_count}, j) + 1)$ where $j > i$.

Complexity will be $O(\mathbf{N}\sqrt{\mathbf{N}})$ to populate the dp table.

Sample Code (C++)

```
const int inf = 1e9;
void knapsack(vector<int> distinct_cycle_sizes, vector<vector<int>> dp, vector<int> cnt, int n) {
  dp[0][0] = 0;
  for(int i = 1; i <= (int)distinct_cycle_sizes.size(); ++i) {
    int cs = distinct_cycle_sizes[i - 1];
    dp[i][0] = 0;
    // Using standard sliding window approach.
    for(int j = 0; j < cs; ++j) {
      deque<int> dq;
      for(int l = j; l <= n; l += cs) {
        dp[i][l] = min(dp[i][l], dp[i - 1][l]);
        int opt = inf, x = -1;
        if(l and !dq.empty()) {
          opt = dp[i - 1][dq.front()] + (l - dq.front()) / cs;
          x = dq.front();
        } else if(!l) {
          opt = 0;
        }
        dp[i][l] = min(dp[i][l], opt);
        while(!dq.empty() and dq.front() <= l - cnt[cs] * cs) {
          dq.pop_front();
        }
        while(!dq.empty() and dp[i - 1][l] <= dp[i - 1][dq.back()]) {
          dq.pop_back();
        }
        dq.push_back(l);
      }
    }
  }
  vector<int> fans(n + 1);
  int best_right = inf, items_count = distinct_cycle_sizes.size();
  for(int i = n; i >= 1; --i) {
    // best_right + 1 accounts for greedy.
    fans[i] = min(dp[items_count][i], best_right + 1);
    best_right = min(best_right, fans[i]);
  }
  for(int i = 1; i <= n; ++i) {
    cout << fans[i] - 1 << " ";
  }
  return;
}
```