NOI 2023 Solution Outlines

NOI jury

March 2023

1 Ice Cream Machines

The task is to minimize the number of times ice cream machines must be cleaned before switching to new flavors. There are n customers, m different flavours and k ice cream machines. Each customer i orders ice cream with a single flavour, represented by c_i, and the customers are served in the order they appear in the input. The ice creams are empty initially and must be cleaned before loading the first flavour.

1.1 Subtask 1

There is only one ice cream machine, meaning we must use it to serve each customer. At first the machine is empty, so it will have to be cleaned for the first customer's flavour. If the next customer has the same flavour then the machine does not need cleaning. Otherwise, we must clean the machine and load it with the customer's flavour. This can be implemented by simply keeping track of the previous customer's flavour as the input is read and then checking if the current customer's flavour is the same.

Time complexity is $\mathcal{O}(n)$, but the bounds are generous, allowing for slower implementations to pass.

1.2 Subtask 2

There are at most two ice cream machines. For each customer we are presented with the choice to use machine 1 or machine 2. We simply try all possibilities and keep track of the currently loaded flavours at each step. Note that if a customer's flavour is already loaded in a machine then it is optimal to use that machine. Let m_i represent the flavour in machine i. Define the recurrence

$$\mathrm{OPT}(i, m_1, m_2) = \begin{cases} 0, & \text{if } i = n+1 \\ \mathrm{OPT}(i+1, m_1, m_2), & \text{if } c_i = m_1 \text{ or } c_i = m_2 \\ \min\left(\mathrm{OPT}(i+1, c_i, m_2), \mathrm{OPT}(i+1, m_1, c_i)\right) + 1, & \text{otherwise} \end{cases}$$

and compute $\mathrm{OPT}(1, -1, -1)$ for the optimal answer. Implementing this recurrence in code results in a time complexity of $\mathcal{O}(2^n)$, since we have two options in each state. We can turn this into a dynamic programming solution.

Time complexity is $\mathcal{O}(nm^2)$.

1.3 Subtask 3

Now we can have up to five machines. As in 1.2 we will use dynamic programming to solve this subtask. Doing so by tracking each machine could be too slow as the time complexity would be $\mathcal{O}(nm^5)$. Note that we only need to know whether a flavour is loaded, not in which machine it is. Therefore we can change the state of the machines into a bitmask, which represents the set of loaded flavours. Depending on implementation, the number of states will be either $n \cdot 2^m$ or $n \cdot \binom{m}{k}$ and each state is computed in constant time.

Time complexity is $\mathcal{O}(n \cdot 2^m)$ or $\mathcal{O}\left(n \cdot \binom{m}{k}\right)$.

1.4 Subtask 4

Suppose we are processing customer i and currently have the choice between using two machines a and b, which have already loaded the flavours m_a and m_b, respectively. Consider the next time m_a and m_b appears in the input after index i. If m_a appears earlier in the rest of the customers' choices than m_b, then it is at least as good to use machine b instead of machine a. A greedy solution that replaces the flavour which comes farthest in the future stays ahead. Since the bounds are low we can implement this naively. Each time we process a customer's order we loop through all the machines. For each machine we loop through the rest of the customers' orders to find the next time the machine's flavour is requested. We then switch out the flavour of the machine whose flavour appears farthest in the future.

Time complexity seems to be $\mathcal{O}(n^2k)$ at a glance, but due to the pigeonhole principle it will be $\mathcal{O}(nmk)$.

1.5 Subtask 5

We will build on the solution from 1.4.

We can pre-compute a table NEXT, where NEXT$[i][j]$ represents the smallest index $i' > i$ such that $c_{i'} = j$. This can be achieved by iterating through the customers' orders in reverse order, maintaining an array LAST, such that LAST$[j]$ represents the last index we saw flavour j. For each order i we then set the value of NEXT$[i][j]$ to LAST$[j]$. Then update $LAST[c_i]$ to i before moving on to the previous order. Building the table has time and memory complexity $\mathcal{O}(nm)$.

The table allows us to cut out the inner most loop from the solution in 1.4 and simply query the table instead for each machine. That removes a factor of m from the time complexity. Querying for the optimal machine is therefore $\mathcal{O}(k)$.

Time complexity is $\mathcal{O}(nm + nk)$.

1.6 Subtask 6

We will build on the solution from 1.5.

We will optimize our data structure for querying the next index. For each flavour maintain a queue (or linked list) of the indices at which that flavour appears, in ascending order. This reduces the pre-computation time to $\mathcal{O}(m)$. When we process an order we pop the front of the queue for the customer's requested flavor.

Time complexity is $\mathcal{O}(nk)$.

1.7 Subtask 7

We will build on the solution from 1.6.

By using a priority queue to maintain the queues, sorting by the front of the queue, we can improve the query time for the optimal machine to $\mathcal{O}(logk)$.

Time complexity is $\mathcal{O}(nlogk)$.

2 Island Alliances

There are n islands, and m bad relationships, each between two islands. The islands intend to unite into alliances, forming states. There are q proposals for alliances, which are processed in order. If any islands involved in the alliance have a bad relationship then the proposal is refused. Otherwise it is approved, merging the two states as well as any bad relationships they carry with them.

2.1 Subtask 1

Here we have $N \leq 500$. One solution is to store an edge for each pair of islands belonging to states that cannot be merged. When two states are merged, the edge sets (which can simply be represented using an $N \times N$ adjacency matrix) are updated by iterating over all islands in the states to be merged and taking the union of the islands that they distrust. Then merging two states takes $\mathcal{O}(N^2)$ time, so the total time complexity is $\mathcal{O}(N^3 + M + Q)$.

2.2 Subtask 2

Here we have $M \leq 250$. Use the union-find data structure to keep track of which islands belong to the same state. Recall that this data structure is usually implemented using a disjoint-set forest, so at all times we can identify each state by the island at the root of the state's tree.

This subtask can be solved by using union find to keep track of state membership, and for each query we iterate over all pairs (a_i, b_i) of islands that distrust each other and check if a_i belongs to the same state as the first island in the query, and b_i belongs to the same state as the second island. The time complexity is $\mathcal{O}(MQ + (N + Q)\alpha(N))$, where α is the inverse Ackermann function.

2.3 Subtask 3

We would like to keep track of which pairs of states contain islands whose inhabitants distrust each other, and for this purpose we keep an (unordered) edge set for each state A's root island containing edges to all the root islands of states B such that some pair of islands in A and B distrust each other. For example, if A is a state that currently contains islands a_1 and a_2, where a_1 is at the root of the state's tree in the disjoint set forest, and B is a state currently only containing the island b_1, then if the inhabitants of the islands a_2 and b_1 distrust each other then a_1 should have an edge to b_1, and vice versa.

In the beginning, each state only contains a single island and so each island is at the root of its state's tree. This means that the edge sets should simply contain all the M pairs of islands given in the input.

However, when two states with root islands a and b are merged, where a is chosen as the root of the new merged state, then all edges (b, c) should be replaced by edges (a, c). This is done by simply iterating over all of b's incident edges, removing them from the sets of both endpoints, and finally adding the updated edges.

Time complexity is $\mathcal{O}(NM + (N + Q)\alpha(N))$

2.4 Subtask 4

Here it is guaranteed that at most one of the proposals should be rejected. We can do a binary search over how many proposals to approve before we have to reject a proposal. To check if the first q proposals can all be approved, we construct a graph G consisting of all the edges (a_i, b_i) for proposals 1 through q. Then we identify the connected components in G and check if any component contains a pair of islands that distrust each other, in which case we can conclude that the first q proposals can not all be approved. Note that there is no need for the union find data structure in this solution. The time complexity is $\mathcal{O}((N + M) \log Q)$.

2.5 Subtask 5

Same as 2.3 but use union-by-rank or union-by-size. To see why this is efficient enough, note that as long as our union find implementation uses either union-by-rank or union-by-size when merging two sets, the number of times any particular island will get a new root island representing its state is bounded by $\log N$. This implies that the number of times any particular edge needs to be updated, because either of its endpoints gets a new root island, is bounded by $2 \log N$. Hence, the time complexity of updating the edges is $\mathcal{O}(M \log N)$.

Finally we note that with the help of these edge sets we can easily check if two states contain islands that distrust each other, by using the union find data structure to find the root island of each state and then checking if there is an edge between the roots. These checks can be done in $\mathcal{O}(Q\alpha(N))$.

The total time complexity of the algorithm is

$$\mathcal{O}(M \log N + (N + Q)\alpha(N))$$

3 ChatNOI

You are given an ordered sequence of n words and an integer k. The task implicitly defines a directed graph where:

- the set of k-grams (i.e. k consecutive words) that appear in the sequence form the vertices,
- the set of $(k+1)$-grams that appear in the sequence form edges from the k-gram formed by the first k words in the $(k+1)$-gram to the k-gram formed by the last k words in the $(k+1)$-gram, and
- the weight of an edge defined as the number of times the corresponding $(k+1)$-gram appears in the original sequence.

You are asked to answer q queries of the form: given an initial k-gram in the graph and an integer m, output a path of length m that starts at the initial k-gram and maximizes the minimum weight of an edge that it traverses.

Fun sidenote: Although the recent hype around ChatGPT is what prompted us to write this problem, it's actually based on a much older and more primitive "chat bot" called Mark V. Shaney. See http://glenda.cat-v.org/friends/mark-v-shaney/grain-of-salt for an interesting piece of computer history written by a now very famous magician.

3.1 Subtask 1

This subtask has a lot of constraints:

- $k = 1$ means that the graph is pretty simple to understand; it's just a graph over the words in the input sequence.
- $m = 1$ means that we only have to produce a path of length 1. In this case the greedy solution of taking the edge from the initial vertex that has the maximum weight works.
- Small n and q means that we don't have to worry much about time complexity.

Summarizing, any solution that outputs the word that appears most frequently following the word in each query should work here. The time complexity is $O(qn)$.

3.2 Subtask 2

The same greedy approach as in 3.1 works since $m = 1$. However, the higher k means that k-grams have to be considered instead of single words, and the higher n and q means that some care has to be taken with regards to time complexity.

A simple algorithm runs over the input sequence and stores a frequency table for each k-gram in a hashmap. The frequency table can then be replaced with the word that has the highest frequency. Each query can then be answered in the time it takes to look up the given k-gram in the hashmap, for a time complexity of $O(nk + qk)$.

3.3 Subtask 3

Now that $m > 1$ the greedy approach of (repeatedly) picking the most frequent word following a k-gram no longer works, and instead we need an approach that finds a path in the graph that maximizes the minimum edge weight traversed.

This subtask has very small values of n and m, meaning that any type of complete search solution should be fast enough. For example, backtracking can be used to test each possible value of the n words in each of the m possible positions, for a total time complexity of $O(nk + qn^m)$.

3.4 Subtask 4

Now that n and m have gotten quite a bit bigger complete search will no longer work. Instead this subtask can be solved using dynamic programming. Specifically, the following recurrence will return the weight of a path of length m starting from vertex u that has the highest possible minimum edge weight:

$$\mathrm{dp}(u, m) = \begin{cases} \infty & \text{if } m = 0 \\ \max\limits_{(u,v)\in E}\Big(\min(w(u, v), \mathrm{dp}(v, m - 1))\Big) & \text{otherwise} \end{cases} \quad (1)$$

where $(u, v) \in E$ iterates over all neighbors v of the vertex u, and $w(u, v)$ denotes the weight of the edge from u to v.

Note that, while it takes $O(nk)$ time to construct the graph induced by the input sequence, it only has n vertices and n edges. If M is the maximum over all query lengths, calculating the recurrence using dynamic programming will visit a total of nM states, and the loop computing the value for each state will visit each edge at most M times over the entire course of the algorithm. Keeping an

auxiliary table of which word gave the optimal answer for each state allows efficient construction of the optimal sentence. This gives an algorithm with total time complexity $O(nk + nM + M')$, where M' is the sum over all query lengths.

3.5 Subtask 5

This subtask has a large n and large queries, but only a small number of them.

Consider a particular query with a given initial vertex u and a desired path length m. Assume we try to find such a path of quality at least Q. If such a path exists, then the optimal path must have quality at least Q. If no such path exists, then the optimal path must have quality less than Q. As such we can use binary search to find the quality of the optimal path.

So how can we discover if there exists a path satisfying the given query that has quality at least Q? First, we can discard all edges that have weight less than Q from the graph, as traversing over any of them would lower the quality of the path below Q. Now we are looking for any path of length m starting at vertex u in the resulting graph.

We could use a dynamic programming algorithm similar to the one in 3.1, but it would be too slow. Dropping the m parameter from the recurrence is also not feasible as the graph has cycles. But this prompts looking at the strongly connected components (SCCs) of the graph. If the initial vertex is inside a SCC of size at least 2, or can reach a SCC of size at least 2, then it is always possible to find a path of length m (or more) by traveling to any such SCC and then looping around the vertices in that component indefinitely.

This can be incorporated into a dynamic programming algorithm. Specifically, the following recurrence will return the length of the longest path starting from vertex u:

$$\mathrm{dp}(u) = \begin{cases} \infty & \text{if } u \text{ is in an SCC of size } > 2 \\ \max\limits_{(u,v)\in E}\Big(1 + \mathrm{dp}(v)\Big) & \text{otherwise} \end{cases} \tag{2}$$

This recurrence can be computed using dynamic programming as the set of vertices in SCCs of size 1 are not part of any cycle, and the recurrence terminates as soon as it encounters a vertex in an SCC of size 2 or more.

Summarizing, the algorithm works as follows for each query. We binary search Q, the quality of the optimal sentence that satisfies the query. For a particular Q, we construct a new graph that is equivalent to the original graph, but with all edges of weight less than Q removed. We use an algorithm such as Tarjan's SCC algorithm to decompose the graph into SCCs and determine the size of the SCC that each vertex belongs to. We calculate the above recurrence using dynamic programming, visiting each vertex and each edge at most once. If the longest path starting at u is at least m then we search for a higher Q; otherwise we search for a lower Q. Finally, when the optimal value of Q has been discovered we can repeat the same procedure to discover a path of length at least m, using an auxiliary table to keep track of the optimal word for each state in the recurrence and then randomly traveling around the final SCC if an SCC of size at least 2 is encountered.

The total time complexity of this is $O(nk + qn\log(n) + M')$, where M' is the sum over all query lengths.

3.6 Subtask 6

In this case the value of n is slightly smaller, but we have a lot more queries.

Consider the algorithm implemented in 3.5. We can reuse some of the computation carried out by the dynamic programming algorithm across different queries. In particular, if we ever test the

same value of Q across two different queries, we will be using dynamic programming to calculate the same recurrence over identical graphs. Thus we only need to compute the recurrence for each Q we encounter once, and store the results for use across different queries.

So which values of Q do we need to test? Starting with $Q = 0$ as a special case, we can notice that the graph only changes when Q is increased to the weight of an edge that exists in the graph; all the value of Q that are strictly between two consecutive edge weights that appear in the graph will all give identical graphs and identical results for the recurrence. As such, we only need to perform binary search over the values of Q that correspond to edge weights that exist in the graph.

It is clear that the edge weights can be any integer between 1 and n. However, a key observation is that the sum of all edge weights in the graph is exactly n. If one wanted to maximize the number of distinct edge weights in the graph, one would label one edge with 1, one edge with 2, one edge with 3, and so on. However, the sum of these edge weights grows quadratically, and already exceeds n after about $\sqrt{n}$ edges have been added. As such, the maximum number of distinct weights is asymptotically at most $\sqrt{n}$. This means that the dynamic programming algorithm only needs to be run $\sqrt{n}$ times, giving the total time complexity of $O(nk + n\sqrt{n} + M')$, where M' is the sum over all query lengths.

3.7 Subtask 7

Continuing the analysis from 3.6, we can notice that the time complexity of running the dynamic programming algorithm for a given value of Q is highly dependent on the size of the graph after all edges with weight smaller than Q have been removed.

When $Q = 1$, all edges in the original graph are included, leaving at most n edges and n vertices for an asymptotic time complexity of the dynamic programming algorithm at $\sim n$. When $Q = 2$, we remove all edges of weight 1 leaving only edges of weight 2 or higher. Recalling that the sum of edge weights is n, we can notice that there are at most $\lfloor n/2 \rfloor$ such edges. Removing the vertices that have no adjacent edges (and thus have a trivial recurrence result), there will be at most $\lfloor n/2 \rfloor$ left. This gives an asymptotic time complexity of $\sim \lfloor n/2 \rfloor$. Similarly for $Q = 3$ we get an asymptotic time complexity of $\sim \lfloor n/3 \rfloor$, and so on.

In total, the asymptotic time complexity of running the dynamic programming algorithm over all values of Q is $n + \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + \lfloor n/n \rfloor$, which is a Harmonic sum bounded by $\sim n \log n$.

By carefully implementing the algorithm described in 3.6 and aggressively removing (i.e. making sure you never reiterate over) edges and vertices that will no longer contribute to the recurrence relation as Q is increased will result in a total time complexity of $O(nk + n \log n + M' \log n)$, where M' is the sum over all query lengths.