# Day 5, SPb SU LOUD Enough Contest 2

February 4, 2023

## Greedy Bipartite Matching

- We want to find a greedy matching in a bipartite graph.
- Greedy matching is a matching that maximizes number of edges of weight 1, then number edges of weight 2, etc.
- We get edges in order of increasing of their weights.
- Now, how to solve the problem? Our plan is to solve our problem for edges of weight 1 first and then modify the graph in a way such that solving it for heavier edges won't break anything.

## Greedy Bipartite Matching

- Consider a vertex cover on our light edges (initially, edges of weight 1).
- Let $L_1, R_1$ be the vertices from $VC$ in left and right part correspondingly, and $L_2, R_2$ be other vertices.
- It is easy to see that light edges of the type $L_1$–$R_1$ never belong to the answer. So, we remove them.
- Similarly, heavy edges can only be of the type $L_2$–$R_2$. So, we remove others.
- Now, we can also see that any maximal matching in the resulting graph could be transformed into a matching having maximal number of light edges.
- Repeat this process for all $q$ weights.

## Greedy Bipartite Matching

- For each weight we need to find a maximal matching and then to remove some edges which can be done in $O(mq)$ total time.
- The naive upper bound is $O(q \cdot \text{Matching}(n, m) + nq)$, which seem to be too much, however, running the optimized version of Kuhn's algorithm was enough to get Accepted.
- Running optimized version of Kuhn at this point of time is a kind of magic that throws against wisdom rather than intelligence, but I'll try to convince you that it clearly fits in TL.

## Greedy Bipartite Matching

- We can use online version of slow Kuhn instead. It will work in time $O(nm + nq)$. Note that there is no $q$ in the first term anymore.
- It still a bit too much, but if you look closely, fast Kuhn could be considered as an optimized version of online Kuhn (up to some additional $O(nq)$ overhead, which is small enough), thus, it should be fast enough already.
- Intuitively, the running time of the described algorithm should be $O(\text{Matching}(n, m) + nq)$, but it is hard to prove, as well as it is hard to prove that Kuhn is indeed fast enough for $m = 2 \cdot 10^5$.
- If you are still in doubt, you can implement Dinic' (or Hopcroft–Karp) algorithm with an easy upper bound of $O\left(mq\sqrt{n}\right)$ and a bound of $O\left(m\sqrt{qn}\right)$ that is harder to prove.

## Banshee

- We have an army of StarCraft banshees, we want to eliminate all opponent's buildings as quickly as possible.
- All our units are at the origin, all buildings have non-negative coordinates.
- Let's forget about shields for a second. How to solve the problem then?
- First of all, we can see that we can limit our banshees to choose their target in the order of increasing coordinate.
- Now, we can do a binary search on the answer and solve the problem of minimizing the amount of banshees required to destroy all the buildings.
- How do we do that?

## Banshee

- Greedy!
- Given the time bound we can see units as buying some number of shots we can distribute on a prefix of buildings.
- Thus, we can consider buildings from right to left and each time choose the smallest number of units we should add in order to have enough unassigned shots to destroy the current building.
- So, we solved a problem without shields in time $O\left(n \log \varepsilon^{-1}\right)$. In this problem $\varepsilon \approx 10^{-15}$.

## Banshee

- But what to do with shields?
- Nothing! We can prove that we can construct an optimal (in the sense of the previous slide) solution that does not allow recharging.
- Consider optimal solutions that minimize the number of the furthest building that recharges its shields. Among those, consider a solution that minimizes the number of different units that are hitting it, but then flies further.
- Consider the banshee that kills the building. If the building is its first target or was targeted only by it, then the shields never recharge.
- Otherwise, we can exchange its shots for the previous builds with the shots of another banshee that flies further. It contradicts minimality.

## Elimination Race

- We have a results of races. We can reorder them, the last participant eliminated.
- For each participant find whether he can win and how (if he can).
- Let's solve a problem for the first participant. Consider a bipartite graph between races and players other than first where the edge is present iff the first participant beats the player in the race.
- The answer exists iff such graph contains a perfect matching.
- Proof in one direction is obvious: we can match races to the eliminated players.

## Elimination Race

- In order to prove into another let's consider a smallest counterexample.
- If there is a race matched with the last player, the counterexample is not smallest.
- Otherwise draw a directed edge from the matched player to the last player in each race.
- We obtained a graph on players with out-degree equal to 1 for each vertex, thus, it contains a cycle. Therefore, we can replace matches according to this cycle and obtain a contradiction.
- Note that this proof also provides us an algorithm to construct a permutation from the matching. It is doable even in linear time, if implemented carefully enough.

## Elimination Race

- The overall complexity is $O(n \cdot \text{Matching}(n, n^2))$.
- Again, known upper bounds for Kuhn are not tight and the best known lower bound for $|E| \approx |V|^2$ is $\Omega\left(\frac{|V|^3}{\log |V|}\right)$, so here we appeal to the knowledge how fast it practically works.
- However, there are two key optimizations:
  1. Using Kuhn's algorithm with bitsets. It leads to the complexity of $O\left(\frac{n^4}{w}\right)$.
  2. Reusing a partial matching from one participant to compute answers for another.

  Using any of them could be sufficient to get Accepted, using both of them makes your code fit into a third of TL. Again, you can also use Dinic for better theoretical bounds.

## Shared Memory Switch

- We need to construct an optimal algorithm for the shared memory switch.
- Firstly, run the algorithm with infinite $B$, compute the time each packet was sent.
- Now, we run second time and each time we have to push out a packet, we choose the packet with the largest time of sending. It is always the last element in the queue, so we can store them in a heap. The complexity is $O(n \log n)$.
- Such algorithm is called LateQD, the proof of its optimality is relatively long (but not necessary too hard), it is written in detail in the paper New Competitiveness Bounds for the Shared Memory Switch.
- This algorithm allowed to obtain new lower bounds for competitiveness for the online policy LQD (Longest Queue Drop) which chooses a longest queue whenever is has to push something out.

## Text Editor

- The task is to make a simple text editor with operations load, write, print, copy, cut, paste, insert, undo and redo.
- A naive solution is to store strings in a persistent treap and answer each query in logarithmic time.
- It would definitely work, but it is not expected to fit into ML and the bounds for serialization.
- The next step is to make each node "wide" and store a small string there, similarly to structures like B-trees. It may get Accepted, however, there is a bit more elegant solution.
- We can store a concatenation of all input queries in a string separately and make nodes of our treap hold views into this string. Now, we don't have to worry about small nodes eating all our memory.
- This is the authors' solution!

## LCSLCSLCS

- The task is to compute LCS for two long strings with short periods.
- A little spoiler: this problem is solvable in time $O(n \log n \log m + n^2)$ where $n$ is the length of our strings and $m$ is the number of repetitions.
- In order to solve this problem, we need a notion of sticky braids that also called seaweed in two (part one, part two) recently published articles on this topic by ko_osaga.
- There is a dynamic programming solution without it, it works in time $O(n^3 \log m)$.
- Now, we will briefly describe some key moments of the solution with sticky braids.

## LCSLCSLCS

- First of all, we need to solve a simpler problem of computing LCS for the strings $A$ and $B^\infty$ using sticky braids.
- The result would be a periodic sticky braid (with period $B$) that we can later use to compute the answer for $A$ and $B^k$ for any $k$ in linear (in $|B|$) time.
- In order to do that, we need to remove all unique letters from both strings and then start computing each new row from the cell with a wall. The complexity of this part is $O(|A||B|)$.
- Now, we only need to learn how to concatenate such periodic braids. If we can do that quickly enough, then the problem could be solved using ordinary binary exponentiation.

## LCSLCSLCS

- Our main obstacle is the fact that the braids are infinite. However, if we take three continuous periods and concatenate them like regular braids, the middle period would be valid as a period for the result.
- The main idea of the proof why is to note that if our resulting braid is invalid (thus, some two lines intersect twice), then it should happen for lines from two adjacent periods.
- So, we reduced our problem to the known problem of multiplication of sticky braids (sticky multiplication).

## LCSLCSLCS

- Sticky multiplication can be done via tropical multiplication of two unit-Monge matrices in cubic time naively or in quadratic time by using Knuth's optimization.
- The total complexities in those cases are $O(n^3 \log m)$ and $O(n^2 \log m)$, respectively. Practically, the first version is about five times faster than the dp solution and the second solution was used to choose the TL for this problem.
- However, there is an algorithm Steady Ant that does sticky multiplication in time $O(n \log n)$ giving the total complexity for this problem of $O(n \log n \log m + n^2)$.
- The authors are very grateful to Alexander Tiskin for his tremendous help in theoretical side of this problem and also for his great course on string algorithms that was held in SPb SU.