Will Code for Drinks 2023

Solutions

November 10, 2023

A: Alcohol

Problem Author: Thore Husfeldt

- **Problem:** Given a chemical compound in SMILES notation, determine the type of alcohol it is.
- **Solution:** Build the (undirected multi-)graph from the specification. This is the difficult part; it requires a nontrivial parser.
  - Use a stack to keep track of parenthetic structure of open branches.
  - Use a symbol table (also known as dictionary or map) to keep track of open cycle bonds.
  - Remember if you've just seen a double bond symbol (=).
- The rest is quite easy: locate the hydroxy groups by searching for single-bonded oxygen with one unspecified neighbour (which must be H). Then scan over all degree-4 carbon atoms adjacent to one or more hydroxy groups and count their neighbouring carbons.
- **Simplifications:** Valences can be ignored. Bond strengths can be ignored except for oxygen (because `CO` has a hydroxy group but `C=O` does not) and carbon (because `C(=O)O` has a hydroxy group but is not an alcohol).

B: Bitte ein Bit

Problem Author: Thore Husfeldt

- **Problem:** Print any element of a given 01-sequence $b = b_1 \cdots b_n$.
- **Solution:** There are many ways of solving this problem, including printing
  - the first bit b_1,
  - the largest bit $\max b$,
  - the smallest bit $\min b$,
  - the disjunction ('or') $b_1 \vee \cdots \vee b_n$,
  - the conjunction ('and') $b_1 \wedge \cdots \wedge b_n$,
  - b_r for random index $r \in \{1, \dots, n\}$.
- **Wrong answers:** Here are some ways of getting it wrong: Print
  - always 0,
  - always 1,
  - the sum $b_1 + \cdots + b_n$,
  - a random sample from $\{0, 1\}$,
  - "Prost!"
- **Runtime expection:** Print
  - the second bit b_2 (which may not exist.)

C: Chugging

Problem Author: Thore Husfeldt and Oskar Haarklou Veileborg

- **Problem:** It takes t seconds to empty the first bottle, $t+d$ seconds for the second, $t+d+d$ seconds for the third, etc. Given such values for Alice and Bob, respectively, who finishes N bottles first?
- **Intended solution (by simulation):** Simulate the process to find the total time T for a participant:

```
    $T = 0$
    while $N > 0$:
        $T = T + t$
        $t = t + d$
        $N = N - 1$
```

  Then compare the resulting totals. *Hint for debugging:* Print the total for every round.
- **Solution by algebra:** A closed formula for the total time of a particcpant is

$$N \cdot (t + \tfrac{1}{2}d \cdot (N-1))\,,$$

  which follows from $1 + \cdots + n = \frac{1}{2}n \cdot (n-1)\,.$

D: Drunk Texting

Problem Author: Bergur Snorrason and Thore Husfeldt

- **Problem:** Given strings s and t, find string m of minimal length such that s is a subsequence of m and t is a subsequence of m.
- The string m can be much shorter than $|s| + |t|$ because of overlaps, for instance if $s = t$.
- We can't determine locally which letters of s to match with letters of t; for instance, for $s =$ oxx and $t =$ xxoxx, the "greedy" matching $\begin{array}{cccccc} \text{o} & \text{x} & \text{x} & & & \\ & \text{x} & \text{x} & \text{o} & \text{x} & \text{x} \end{array}$ is inferior to $\begin{array}{ccccc} & & \text{o} & \text{x} & \text{x} \\ \text{x} & \text{x} & \text{o} & \text{x} & \text{x} \end{array}$.
- Exhaustive search is too slow: there are at least $2^{|s|+|t|}$ different ways of aligning s and t.
- **Solution:** Instead, this is a prototypical problem for the technique called *dynamic programming*: Define, for each $0 \leq i \leq |s|$ and $0 \leq j \leq |t|$ the length $\mathsf{L}(i,j)$ of an optimal alignment of the prefixes $s_1 \cdots s_i$ and $t_1 \cdots t_j$. Then we have the recurrence

$$\mathsf{L}(i,j) = 1 + \min \begin{cases} \mathsf{L}(i-1, j-1)\,, & \text{if } s_i = s_j\,; \\ \mathsf{L}(i, j-1)\,, \\ \mathsf{L}(i-1, j)\,. \end{cases}$$

- Recovering a solution string from the values $L(i,j)$ is, again, a standard dynamic progamming method; typically solved by traversing a table of values "backwards." Total time and space is $O(|s| \cdot |t|)$.

E: Shandy

Problem Author: Thore Husfeldt

- **Problem:** How many whole bottles of shandy correspond to B bottles of beer and L bottles of lemonade?
- **Intended solution:** The answer is $2 \cdot \min\{B, L\}$.
- **Simulation:** Instead, you can pour pairs of shandies:

```
$S$ = 0
while $B > 0$ and $L > 0$:
    $S$ += 2
    $B$ -= 1
    $L$ -= 1
print($S$)
```

F: Fair brewing

Problem Author: Oskar Haarklou Veileborg and Thore Husfeldt

- **Problem:** Given a description of a network of beer tanks, bottling lines and joints connected by pipes with capacities, determine the maximum amount of beer that can be produced in a valid configuration, or determine that no valid configuration exists.
- **When does a valid configuration exist?** In a valid configuration there is a path in the network from each tank to distinct bottling lines. Notice that joints can only connect disjoint pairs of incoming pipes, which means the paths from the tanks to bottling lines must be edge-disjoint.
- We can add two artificial vertices to the network, S and T, add edges between S and the K beer tanks, and add edges between T and the K bottling lines. Now a valid configuration exists iff. there are K edge-disjoint paths from S to T.
- This check reduces to a max-flow problem. Namely, if the maximum S-T flow in the new network (where all edges have capacity 1) is K, there exists a valid configuration.
  Use your favorite flow algorithm to solve this problem.
  Ford-Fulkerson algorithms run in $\mathcal{O}(K \cdot (M + N + K))$ time, which is fast enough.

F: Fair brewing

Problem Author: Oskar Haarklou Veileborg and Thore Husfeldt

- **Now that we know when a valid configuration exists, how do we find the best one?**
- When we find K edge-disjoint paths from S to T, it corresponds to a valid configuration in the original problem where the amount of beer produced is equal to the minimum capacity of a pipe (from the original network) used in one of the K paths.
- There are multiple possible solutions at this point. Assume that we do not want to change how we find the K paths (i.e. we want to use a max-flow algorithm as a black box).
- **Insight:** We can check whether we can find K edge-disjoint paths after removing all pipes (A_i, B_i, C_i) from the network with $C_i < \mathbf{C}$. If the K paths exist, they must necessarily use pipes with $C_i \geq \mathbf{C}$. If we cannot find K paths, we know that we need to use some pipes with $C_i < \mathbf{C}$ to get a valid configuration.
- **Solution 1:** Binary search to find the largest $\mathbf{C}$ where a valid configuration exists. There are at most M distinct capacities, so this can be done in $\mathcal{O}(\log M \cdot K \cdot (M + N + K))$ time in total.

F: Fair brewing

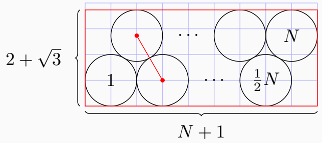
Problem Author: Oskar Haarklou Veileborg and Thore Husfeldt

- If you are comfortable with the implementation of max-flow algorithms, there are more options:
- **Solution 2:** Process pipes in decreasing order of C_i with an incremental flow algorithm until K edge-disjoint S-T paths have been found. The answer is equal to the capacity of the last added pipe. This can be done in a total time of $\mathcal{O}(M \log M + K \cdot (M + N + K))$.
  Slower algorithms may also work.
- **Solution 3:** Reduce the problem to a maximum cost max-flow problem, where the cost of a path is equal to the minimum capacity (from the original network) of a pipe used in the path. Then solve the problem with the successive shortest path algorithm.
  The implementation is a lot simpler than the (standard) case where the cost of a path is the sum of weights of edges on the path, as we do not add negative weight edges to the residual network.
  The paths can therefore be found with a standard implementation of Dijkstra's algorithm.
  This solution has a time complexity of $\mathcal{O}(K \cdot (M + N + K) \log(N + K))$.

G: Greener Tuborg

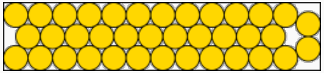
Problem Author: Thore Husfeldt

- **Problem:** Pack N unit circles into a rectangle of area at most $4N - \frac{1}{100}$. For many small N, this is impossible; for instance the standard 2×3 sixpack is optimal. But for $N \geq 14$, this can always be done, often in many ways.
- **Solution:** It is good enough to pack the beers in just two rows:



The height is $h = 2 + \sqrt{3}$. For even N, the width is $N + 1$. (For odd N, the width is N.) The resulting area is $(2 + \sqrt{3})(N + 1)$, which is smaller than $4N$ for $N \geq 14$.

- **Other packings:** Can one do better? Yes. For most N, a hexagonal packing with more than 2 rows is the best known. But weird things happen, such as for $N = 37$:



Actual *optimality* is an open problem for most N(!)

H: Handkäs and Mispelchen

Problem Author: Lars Huth and Thore Husfeldt

- **Problem:** The ith out of k pubs offers h_i many handkäs and m_i many mispelchen. Find the shortest sequence of visits such that you can consume n handkäs and n mispelchen. Common courtesy dictates that you cannot consume any mispelchen before you have consumed n handkäs.
- **Observations:**
  1. You may have to visit the same pub twice.
  2. Drinking at a pub after finishing their handkäs costs nothing.
  3. Checking all sequences requires checking at least $(k!)^2$ cases, which is infeasible.
  4. If there was only one course, this is a prototypical greedy problem: sort pubs by availability, largest first, then visit pubs in that order until you're full. (This is optimal by an exchange argument.)
  5. Greedily sorting by h_i and then worrying about the mispelchen once you're full makes you miss optimal solutions by 1 switch; sample 1 shows this.
- **Solution:**
  1. If you knew the "state-changing" pub i (where you consume the nth handkäs and then start drinking) then it would be two easy problems: Greedily visit pubs (except i) to find $n - h_i$ handkäs, and greedily visit pubs (except i) and $n - m_i$ mispelchen.
  2. We don't know i, but we can just iterate over $i \in \{1, \ldots, n\}$ and report the best outcome of those subproblems. Time $O(k^2 \log k)$
  3. Also doable in time $O(k \log k)$ if you're careful about choosing the "last handkäs place". Very easy to mess up.

I: Uno Reverse

Problem Author: Johan Sannemo

- **Problem:** Players $2, \dots, K$ have in total played in N Reverse cards in *Uno*, starting with player 2. Can it be player 1's turn?
- **Intended solution:** There are a number of easy cases:
  - $N = 0$: it is player 2's turn, so "NO".
  - $N = 1$: the card was played by player 2, so "YES".
  - $K = 2$ (but $N > 0$): player 1 is the only other player, so "YES"
  - N odd (but $N \geq 3, K > 2$). Player 2 can have played all the cards; if so, it's now 1's turn. Or 2 played two cards, after which 3 and 2 alternate; if so, it's now 2's turn. So "MAYBE".

  In the remaining cases, N is even. Then possibly K was last to act, after play passed through all other players, each playing at least 2 cards. So if $N \geq 2(K-1)$ the answer is "MAYBE", otherwise "NO".
- **Simulation:** There are only $N-1$ moments where the turn can have passed on to another player, so there are at most $2^{N-1} \leq 2^{15} = 32\,768$ different ways the game can have happened while player 1 was away. Simulate all of them; remember that player 1 did not act.

J: Juggling

Problem Author: Johan Sannemo

- **Problem:** A number of bottles are thrown, one at a time. The sequence of digits $t = t_1 \dots t_n$ describes that the bottle thrown at time i is also thrown at time $i + t_i$. Given a juggling sequence, is it valid and minimal in the sense of $t \neq ss \cdots s$ for any other sequence?
- **Solution:** Need to check two things:
  - t contains no repetitions.
  - t is a "juggleable" sequence in the sense that no two balls ever are thrown at the same time.
- **Repetitions.** Need to check if $t = s^d$ for some other prefix s of t and some integer r.
  - Simplest way is to check this for *each* prefix s of t; this requires $O(|t|^{3/2})$ in the worst case, if we restrict to d that divide $|t|$.
  - Observation: The concatenation tt contains the substring t more than twice exactly if t is periodic. Thus, we only need to find the 2nd occurrence of t in tt. Substring search is in linear time by nontrivial algorithms, which are built into some programming languages.
  - Can probably do something clever with tries.
- **Juggleability.** Just simulate beat by beat, keeping track of when the current bottle will be re-thrown. Watch out for inputs that are shorter than the sequence lengh, like $(1, 9)$. Fix this either by computing modulo $|t|$ or by repeating t a few times. Need to check for collision at beats $2, \dots, n + 9$; e.g., in input $1 \cdots 12$ the collision only occurs at beat $n + 1$.

Will Code for Drinks 2023

Jury work

- Problem authors: Bergur Snorrason, Christian Janos Lebeda, Holger Dell, Johan Sannemo, Lars Huth, Oskar Haarklou Veileborg, Thore Husfeldt (chair.)
- Special thanks to Ragnar Groot Koerkamp and Sebastian Mateos Nicolajsen.
- Stated goal this year: even better problem quality, eventually WCFD should stand next to NCPC, BAPC, NWERC wrt. problem quality. (But remain much easier.)
- Task development: 300+ commits, 200+ programs (generators, validators, model submissions), 5000+ lines of code, 600 testcases. Shared Github repo, communication via Slack.
- New: problem statements in Danish and German.
- Solution slides.
- New: Used BAPCtools (originally developed for the BAPC and NWERC contests.) WCFD led to lots of development on BAPC-tools, in particular for internationalisation. (Statements are in three languages now.) WCFD was also an early use case for BAPCtools' improved generators framework; development began before Summer 2023.