ICPC Asia-Manila 2022 Solution Sketches

Problem Discussion

Cisco Ortega

December 16, 2022

Problem A - An Easy Calculus Question

- Reading comprehension! :)
- Just implement the piecewise function.
- The problem statement actually *gives you the answer*

Problem K - Kapitan Amazing

- No need to bother with the positions in the QWERTY layout. The oily keys are just the letters of the alphabet that *don't* appear in the three lines.
- For each query: check if the characters in s are equal to the oily keys, after removing duplicates and if order does not matter.
- Easily done by just testing for `set` equality.

Problem C - Conform Conforme

Algorithm analysis

- Use a frequency array or unordered map in order to count the frequencies in $\mathcal{O}(n)$
- It's too slow to apply the transformation k times
- Each transformation can be done in $\mathcal{O}(n)$, so that would result in a running time of $\mathcal{O}(nk)$ operations
- With $n \leq 2 \cdot 10^5$ and $k \leq 10^9$, that is way too slow!

Problem C - Conform Conforme

- **Observation:** The array very quickly becomes a fixed point of the transformation.
- So, just naively apply the operation **up to** k times, but if ever the array remains unchanged after the transformation, you can just stop already.
- Running time: $\mathcal{O}(n \times \min(k, \text{iters until fixed point}))$

Problem C - Conform Conforme

- **Claim:** After applying the transformation, for any value v in the array, we have that freq(v) is divisible by v.
- **Definition:** A value v is *refreshed* if there exists an element **not equal to** v that was then mapped to v in the most recent transformation.
- **Claim:** After the ith application of the transformation, all *refreshed* elements are $\geq 2^{i-1}$

Problem C - Conform Conforme

- Proofs left as an exercise :)) Induction is helpful
- This guarantees that there are no more refreshed values after $\mathcal{O}(\lg n)$ iterations.
- Running time: $\mathcal{O}(n \times \min(k, \lg n))$

Problem J - Junior Steiner Three

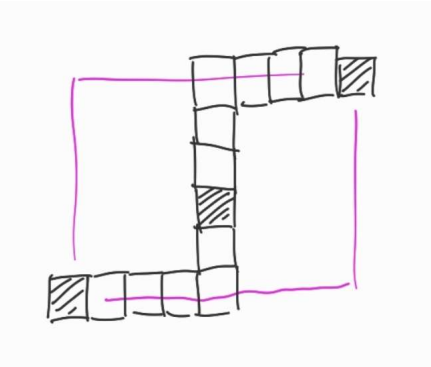
This will be verbose because we're going to prove the optimality of our solution... but the actual construction given at the end is much simpler.

- Suppose that the land cells are at coordinates (x_1, y_1), (x_2, y_2), and (x_3, y_3), **sorted in nondecreasing order of** x
- The answer will have *at least* $|x_3 - x_1| + |y_3 - y_1| + 1$ land cells, in order to connect these two extreme points.

Problem J - Junior Steiner Three

Case I. The middle point lies on a shortest path between these two points.

- Visually: It lies "inside the bounding box" of the other two points.

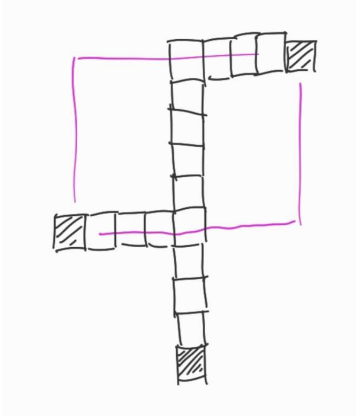


- Then the minimum of $|x_3 - x_1| + |y_3 - y_1| + 1$ can be achieved by just coursing a path through the middle point.

Problem J - Junior Steiner Three

Case II. The middle point *does not* lie on a shortest path between these two points.

- Visually: It lies "outside the bounding box" of the other two points.



Problem J - Junior Steiner Three

- There must be at least $y_2 - \max(y_1, y_3)$ (if above both) or $\min(y_1, y_3) - y_2$ (if below both) land cells used in order to make the middle point connect with the top or bottom row of the bounding box.
- We also must have the $|x_3 - x_1| + |y_3 - y_1| + 1$ land cells for the shortest path between the left and rightmost points.
- These land cells are sufficient, because we can specifically choose a shortest path that allows the middle point to connect with the bounding box.

Problem J - Junior Steiner Three

Simple implementation:

- Extend a line segment at column x_2 that extends from row $\min(y)$ until row $\max(y)$
- Extend a line segment at row y_1 from column x_1 to column x_2, then extend a line segment at row y_3 from column x_2 to column x_3; this connects the left and rightmost points to the central rod.
- Running time: $\mathcal{O}(rc)$

Problem I - Item Crafting

- **Not max flow**

Problem I - Item Crafting

Credits: Kyle See

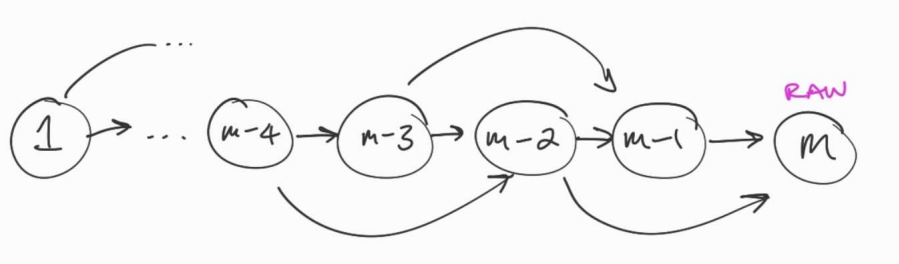
- For each item that is not a raw material, precompute how much of each raw material is needed in order to craft it, by summing up how much of each raw material is needed in order to craft each item in its recipe
- Doable in linear time if you make sure to process a recipe *only after* you've processed all its ingredients
- Because each ingredient in a recipe must have a greater ID than the final product, you can just process the items in decreasing order from m to 1.

Problem I - Item Crafting

- Now, for each legendary final product, suppose we know how much of each raw material is needed in order to make it.
- Because n is small, we can just brute force all possible subsets of legendary final products. Find the maximum-size subset such that we can afford to make all the legendary final products in it.
- Running time: $\mathcal{O}((m + \sum c_i) \times \text{no. of raw} + 2^n \text{poly}(n) \times \text{no. of raw})$

Problem I - Item Crafting

- **Gotcha:** The number of raw materials needed to craft some item can be **exponential** in m.
- For example:



- The amount of raw material needed by item 1 is equal to the mth Fibonacci number, which is $\mathcal{O}(\varphi^m)$

Problem I - Item Crafting

- I hope you handled overflow correctly!
- Easy fix: Introduce a hard cap.
- If you need more of a raw material than you have on hand (only up to 10^8), then immediately declare it as impossible
- Alternatively, you can just suppress any numbers greater than $10^8 + 1$ back down to $10^8 + 1$ at each intermediate step.

Problem G - Gallivanting Merchant

- Note that it is always optimal to choose an s such that $0 \leq s < k$.
- Here, choosing $s = 0$ is equivalent to choosing $s = k$.
- Create an array `buyable` of length k, initialized to all 0.
- We want `buyable[s]` to count the number of purchasable items if we ask the merchant to first appear on this day s.

Problem G - Gallivanting Merchant

- Process each item. This item is purchasable after choosing some starting s if and only if there exists a value $v \in [L, R]$ such that $v \bmod k = s$.
- If $R - L + 1 \geq k$, then this item is actually always purchasable, so we apply `buyable[s] += 1` for all s in $[0, k-1]$
- If $R - L + 1 < k$, then we apply `buyable[s] += 1` for each s in $[L \bmod k, R \bmod k]$.
  - Note that in the case that $L \bmod k > R \bmod k$, this should be broken down into the *two* ranges $[0, R \bmod k]$ and $[L \bmod k, k-1]$.

Problem G - Gallivanting Merchant

- The answer is `max(buyable)`
- This could be solved with a difference array, but that takes $\mathcal{O}(k)$ time and memory, which is too much
- Instead, you can *simulate* a difference array using a `map` or `dict`, keeping track of only the "interesting" points.
- Note that this is also basically just doing a line sweep.
- Running time: $\mathcal{O}(n \lg n)$

Problem F - Factions vs The Hegemon

- Just implement the process :)
- Helpful observation: Once the civilization is in hegemony, it never escapes it.
- We can implement *killing max* and *killing min* separately and just switch over when hegemony is reached.

Problem F - Factions vs The Hegemon

- Find the wealthiest nation using a balanced binary search tree or priority queue, which stores (wealth, index) pairs
- Update elements: delete and re-insert, or delete and let the old values be "staled"
- Do something similar to find the least wealth nation, once hegemony is reached
- Not a hassle at all if you know how to pass a custom comparator into your priority queue :))

Problem F - Factions vs The Hegemon

- How to find the nearest neighbors to the west and east, while accounting for deletions?
- One easy way: Just use pointers!
- Have each faction maintain a pointer to its left and right neighbors, and update these accordingly whenever a faction collapses
- Running time: $\mathcal{O}(n \lg n)$

Problem L - LCG Manipulation

- Use induction to show that $x_n \equiv a^n s + b(1 + a + a^2 + ... + a^{n-1}) \pmod{p}$
- The general strategy: Let v equal the right-hand side, and then isolate n.

Problem L - LCG Manipulation

Case I. $a = 1$

- This reduces to $x_n \equiv s + bn \pmod{p}$

$$\begin{aligned} v &\equiv s + bn && \bmod p \\ n &\equiv (v - s)b^{-1} && \bmod p \end{aligned}$$

- The multiplicative inverse $b^{-1} \bmod p$ can be found using e.g. Fermat's Little Theorem
- $b \neq 0$ by the given constraints

Case II. $a \neq 1$

- Using the geometric series formula, $x_n \equiv a^n s + b(a^n - 1)(a - 1)^{-1} \pmod{p}$

$$\begin{aligned} v &\equiv a^n s + b(a^n - 1)(a - 1)^{-1} && \mod p \\ v(a - 1) &\equiv a^n s(a - 1) + b(a^n - 1) && \mod p \\ v(a - 1) + b &\equiv a^n (s(a - 1) + b) && \mod p \end{aligned}$$

Problem L - LCG Manipulation

Case II.A. $s(a-1)+b \equiv 0 \pmod{p}$.

- Actually, note that this is equivalent to saying $s \equiv as+b \pmod{p}$
- In other words, in this case, we have a *constant* sequence
- The answer is $n=0$ if $s=v$, and `IMPOSSIBLE` otherwise.

Problem L - LCG Manipulation

Case II.B. $s(a-1)+b \not\equiv 0 \pmod{p}$.

$$a^n \equiv (v(a-1)+b)(s(a-1)+b)^{-1} \mod p$$

- Recall that n is the only unknown here. The entire RHS is some constant.
- To isolate n, we would set "$n = \log_a(\text{RHS})$", but this logarithm is performed modulo a prime p
- Doable in $\mathcal{O}(\sqrt{p})$ using the baby-step giant-step algorithm (a meet-in-the-middle algorithm)
- Running time: $\mathcal{O}(\sqrt{p})$ per test case

Problem E - Escape from Markov

Credits: Kyle See

- Blow up the graphs so that the vertices are a *tuple* (u, r), meaning you are at city u, and at this current time, the police are blocking the rth roads in their patrols.
- The value of r will cycle modulo l as time passes
- Then, just perform a BFS to find the shortest path to the target
- Unfortunately, this graph would have $n \times l$ vertices, which is too many.

Problem E - Escape from Markov

- Suppose you **first** reach city u at time t, and then consider all possible actions you can take from there. Then, suppose you later find yourself at city u again, at some later time t' such that $t \leq t'$.
- **Observation:** Revisiting city u at a later time **provides us with no new options** that we couldn't have considered the first time we were at u.
- When we first arrived at city u at time t, it was already an option to just wait there until time t'.

Problem E - Escape from Markov

- Therefore, it is never suboptimal to try to reach each city at the **earliest time possible**. Furthermore, after we first reach city u and explore our options from there, **we never have to revisit it again**.
- So, *don't* blow up the graph, and just do Dijkstra's!

Problem E - Escape from Markov

What are the edge weights?

- Suppose you are at vertex u at time t. The weight of some $u \to v$ edge is equal to 1 plus the minimum amount of time we need to wait until there are no patrol cars on the (u, v) edge.
- For each edge, store a set of all the times $(\mathrm{mod} k)$ that it is impassable due to a patrol car passing through it (there are only pl such bad times, in total, across all edges).
- We can precalculate all the nonzero minimum waiting times in $\mathcal{O}(pl)$ and store them in a map to be later accessed in $\mathcal{O}(\lg l)$ time.
- Running time: $\mathcal{O}(pl + (m + n) \lg n \lg l)$

Problem H - HIIT

Solution 1

- To find the minimal number of skips, sort the exercises in nondecreasing order of a.
- Suppose that in the optimal solution, we can do up to k exercises. Then, greedily, we can always just take the k exercises with the cheapest a values.
- Let these cheapest k be our **chosen** exercises
- We now need to decide which to "upgrade" into their intense forms.

Problem H - HIIT

Solution 1

- **Option 1:** Upgrade a chosen exercise to its intense form
- **Option 2:** Replace a chosen non-intense exercise with an intense non-chosen exercise
- Example where Option 2 is needed:

```
3 6
2 2 3
9 9 4
```

Problem H - HIIT

Solution 1. The greedy options.

- **Option 1:** Upgrade the chosen exercise whose $b - a$ value is minimal into its intense form
- **Option 2:** Replace the chosen exercise whose a is **maximal** with the non-chosen exercise whose b is **minimal**
- Of course, when an exercise is replaced or taken, it is removed from the pool for future consideration
- Running time: $\mathcal{O}(n \lg n)$ using a priority queue

Problem H - HIIT

Solution 2

- Sort in nondecreasing order of a so that we can find k, the maximum number of exercises we can do.
- For the rest of this solution, **we sort in nondecreasing order of b.**
- **Observation:** If we take exercise j intensely, then all exercises i such that $i < j$ must all be taken (either easily *or* intensely)
- **Proof:** If an $i < j$ exists that wasn't taken at all, then we can always get as good of an answer, but cheaper, by taking i intensely instead of j.

Problem H - HIIT

Solution 2

- Therefore, there exists some magical j such that:
  - All intense exercises have index $\leq j$
  - All exercises $\leq j$ are taken (though not necessarily all intensely)
- Note that we *can* take exercises with index $> j$, it's just that they won't be done intensely.

Problem H - HIIT

Solution 2

- We can just try all possible j.
  - Take the first j exercises, not-intensely.
  - Of the remaining $n - j$ exercises, take the $k - j$ cheapest of them not-intensely
  - Using our remaining "budget", try to upgrade these as many of the first j exercises as possible into their intense forms.
- We may only upgrade the exercises with index $\leq j$ by our definition of the magical j.
- Greedily, it is never sub-optimal to always upgrade, at each step, the exercise whose $b - a$ is minimal.

Problem H - HIIT

Solution 2

- In order to test all possible j, we need to be able to perform this simulation quickly for each j
- A modified segment tree can accommodate all the needed queries in logarithmic time each
- Running time: $\mathcal{O}(n \lg n)$

Problem B - Better than Bitcoin

- You can compute the first n primes naively. You don't even need a sieve or the $\mathcal{O}(\sqrt{p})$ primality testing algorithm.
- Let $S(n)$ be the sum of the first n primes.
- By Prime Number Theorem, $S(n)$ is $\mathcal{O}(n^2 \lg n)$
- But also... you can just compute this sum directly

Problem B - Better than Bitcoin

- Straightforward $\mathcal{O}(n\ S(n))$ knapsack DP
- Let $\text{dp}(i, A)$ count the number of ways to give Alice a total of A when considering only the first i primes.
- Note that we don't need to track B, the total value given to Bob, because it can be recovered as $S(i) - A$.
- Transition: $\text{dp}(i, A) = \text{dp}(i-1, A) + \text{dp}(i-1, A\ -\ i\text{th prime})$

Problem B - Better than Bitcoin

- Suppose p and q are in lowest terms.
- If $S(n)$ is not divisible by $p+q$, then the answer is 0.
- Otherwise, the answer is dp $\left(n, \frac{S(n)}{p+q}\ p\right)$

Problem B - Better than Bitcoin

- How to do better? ...You probably can't!
- **Intuition:** Primes usually behave sort of like random numbers, especially when we're adding them.
- **Intuition:** If the input is random, then there's *literally nothing else we can do* that is smart

Problem B - Better than Bitcoin

- However, note that there aren't *that* many possible test cases
- Let's precompute the answers locally and hard-code them into our source code :DDD

Problem B - Better than Bitcoin

- Even with this method, your DP cannot afford to use $\mathcal{O}(n\ S(n))$ memory
- But note that $\text{dp}(i, A)$ only depends on the *previous* layer, i.e. the values of $\text{dp}(i-1, A)$ and $\text{dp}(i-1, A\ -\ i\text{th prime})$
- You can write a solution that uses only $\mathcal{O}(S(n))$ memory by only maintaining two layers at a time: the previous and current layers.
- Or, if you order the loops of the knapsack DP properly, you can get away with using just a single array!

Problem B - Better than Bitcoin

- The file size limit for this contest was 256 KB, which is plenty lenient.
- You can lower the file size by assuming $p \leq q$, and only encoding the cases where the answer is nonzero (and assuming the answer is 0 if it isn't found)
- The judge's brute-forcer directly outputs a copy-pasteable Python `dict`, and even without nontrivial compression algorithms, the model solution is only ≈ 100 KB large
- You can get it to under 64 KB by compressing the info and then parsing it at runtime, but this is not necessary for ICPC Manila 2022.

Problem B - Better than Bitcoin

- The judge's most optimized brute-forcer runs in ≈ 4 seconds when compiled using the `-O3` flag
- But it's okay even if your solution takes a long time—you can let it run in the background while you do other stuff
- Running time of submission: $\mathcal{O}(1)$ per test case (unordered map access).

Problem D - Domination Devil

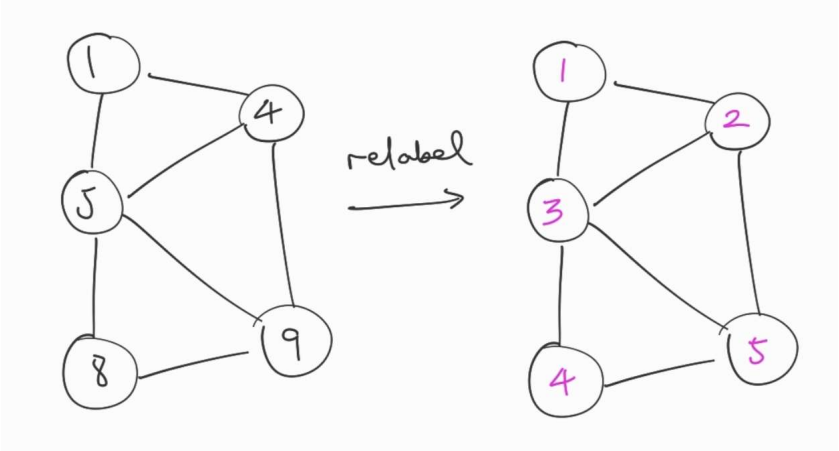
- Consider a graph with its vertices labeled 1 to n.
- If each vertex is connected to at most k other vertices with a greater label, let's say that the graph satisfies the k-*power* condition
- Note that a graph can be decomposed into its connected components, and that a graph satisfies the k-power condition if and only if each of its connected components does too.

Problem D - Domination Devil

- Suppose a connected component has i vertices. If we replace the label of each vertex with the integers from 1 to i such that the *relative ordering* of the labels is preserved (i.e. replace each value with its "sorted order" in this conneted component), then the relabeled graph *still* satisfies the k-power condition.

Problem D - Domination Devil

- Consider this graph with satisfies the 2-power condition before and after relabeling.



Problem D - Domination Devil

- Let all_n count the number of labeled graphs on n vertices that satisfy the k-power condition.
- Let conn_n count the number of **connected** labeled graphs on n vertices that satisfy the k-power condition.
- It seems that all and conn are related somehow, because the objects counted by the former are built up out of objects counted by the latter.

Problem D - Domination Devil

How many graphs with n vertices satisfy the k-power condition?

- Suppose the connected component containing vertex n has i vertices.
- We also need to choose $i-1$ out of the remaining $n-1$ vertices, to decide who is with vertex n in its connected component.
- We arrange these i vertices into a connected component, and then we need to figure out how to arrange the remaining $n-i$ vertices into *the rest of the graph*.

$$\mathsf{all}_n = \sum_{i=0}^{n-1} \binom{n-1}{i-1} \mathsf{conn}_i \ \mathsf{all}_{n-i}$$

Problem D - Domination Devil

- We need to compute all_i for i from 0 to n.
- Recursively,

$$\text{all}_{i+1} = \text{binomSum}_i \times \text{all}_i$$

- where binomSum_i is defined as follows,

$$\text{binomSum}_i = \binom{i}{0} + \binom{i}{1} + \binom{i}{2} + \cdots + \binom{i}{k}$$

- You can use the fact that $\text{binomSum}_i = 2 \times \text{binomSum}_{i-1} - \binom{i-1}{k}$.
- **Proof:** Pascal's Triangle

Problem D - Domination Devil

- This recurrence relation leads to an $\mathcal{O}(n^2)$ algorithm using DP, but that's too slow.
- ...doesn't that recurrence relation sort of smell like polynomial multiplication?
- Let's use **generating functions**!

Problem D - Domination Devil

- Define the following exponential generating functions,

$$A(x) = \sum_{n \geq 0} \mathsf{all}_n \frac{x^n}{n!}$$

$$C(x) = \sum_{n \geq 0} \mathsf{conn}_n \frac{x^n}{n!}$$

Problem D - Domination Devil

- By expanding out the previous recurrence relation and doing some algebra,

$$A(x) = \exp(C(x)),$$

but we can invert this to get a formula for $C(x)$ in terms of $A(x)$:

$$C(x) = \ln(A(x)).$$

- Actually, slogging through the algebra is not very enlightening. I suggest reading Ch. 3 of generatingfunctionology by Wilf or Ch. 2 of Analytic Combinatorics by Flajolet and Sedgwick.
- The theory gives an easy and insightful way to directly derive this equation from the fact that graphs are built up out of *connected* graphs.

Problem D - Domination Devil

- In order to compute the first n terms of the power series of $\ln(A(x))$,

$$\begin{aligned} C(x) &= \ln(A(x)) \\ \frac{\mathsf{d}}{\mathsf{d}x} C(x) &= \frac{A'(x)}{A(x)} \\ C(x) &= \int \frac{A'(x)}{A(x)} \mathsf{d}x + \text{constant} \end{aligned}$$

Problem D - Domination Devil

- Differentiation and antidifferentiation are easy to do, using the power rule
- Computing the first n terms of $1/A(x)$ can be done in $\mathcal{O}(n \lg n)$ using a technique similar to Hensel lifting.
- Where to learn the lifting technique:
  - Tutorial of Codeforces' "The Child and Binary Tree", problem 438E
  - Tutorial of CodeChef's "Chef and Rainbow Road", code `RNBWROAD`
  - Section "Hensel's lemma" in the article "Operations on polynomials and series" on cp-algorithms

Problem D - Domination Devil

- In order to do all this, we need to be able to perform fast polynomial multiplication
- You can use FFT (specifically, an NTT) because $1699741697 = 2^{20} \times 1621 + 1$ has a 2^{20}th primitive root of unity.
- Running time: $\mathcal{O}(n \lg n)$