NAPC 2021 Week 1

Authors: Patrick Li, Allen Pei, and Ava Pun
Edited by: Andrew He and Richard Peng
Incorporated comments from: Jingbang Chen, Yufan Huang, Zack Lee, Charlie Liu, Rungzhe Wang

Combo is hard. There are lots of tough combinatorial bijections and counting techniques.

- map onto 2D plane and establish a bijection
https://atcoder.jp/contests/abc205/tasks/abc205_e

These are catalan type things? https://yufeizhao.com/olympiad/bijections.pdf

- **E**cnerwala, aka. Andrew He: “All problems I can think of of this type are super hard, one problem from OpenCup is finding number of times paths from (0,0) to (x, mx) cross the diagonal with m large and x like 1e5 or something”

On the other hand, for CS, you can just... doit...

So Instead the NAPC Trainers Recommend: Brute Force & Look for Pattern

Sometimes, you will encounter problems which seem unapproachable or impossible to solve. However, it is often possible to guess the answers to these problems based on viewing some small test cases, especially if math is involved. Done quickly and intelligently, guessing can be a penalty-free approach to some problems.

https://atcoder.jp/contests/abc205/tasks/abc205_e is an example of this approach. The intended solution uses multiple clever insights to generate a mathematical formula that can also be guessed.

The key to guessing is to use a brute force program to accurately print out the results of small test cases. The first step is looping through all possible small inputs. What exactly constitutes a small input depends on the time complexity of the brute force algorithm that will be used for each input. Thus, it is beneficial to estimate this time complexity beforehand. Next, a simple and correct but slow brute force algorithm should be used on each generated input. Any bugs here will block any chance of seeing the correct pattern later, so be sure to double check this part. Finally, print the results in an easy to understand format. Brute force code for the example is as follows: https://ideone.com/BI4W2n

At this point, look over the brute force output in order to check for patterns. If you are lucky, something will be immediately obvious and you can code it up and try to submit it. Otherwise, some further manipulation may be needed. In the case of the example, the upper bound for the solution can be easily obtained using trivial math, and the difference between the upper bound and the actual solution is the value that follows an obvious pattern. If anything looks promising, be sure to edit your brute force code to print these manipulated values to help see patterns.

Here is a brief summary of the pattern in the example output. All the nonzero values in the DIFF column seem to be the result of a mathematical combination. Furthermore, it seems that the total number for each combination is n + m. This is especially obvious when n = 4 and m = 3, as DIFF values of 35, 21, and 7 show up, so 7 choose 3, 7 choose 2, and 7 choose 1 respectively. The number being chosen for each combination decreases as k increases. We know that the DIFF of 7 choose 3 was when n = 4, m = 3, and k = 0. Trying the formula n + m choose m - k doesn't work. However, the formula n + m choose n - k - 1 does. Thus, the answer is (n + m choose n) - (n + m choose n - k - 1) along with some simple border cases that are obviously 0.

More problem where print table helps:

https://open.kattis.com/contests/nac20open/problems/allkill (this looks quite scary, but the answer is some simple 1-line formula of the input numbers, so a brute force + some small numbers should make this apparent)
https://atcoder.jp/contests/agc021/tasks/agc021_e (answer is something involving a choose)
https://atcoder.jp/contests/agc054/tasks/agc054_e (more shenanigan w. chooses)

Harder ones:
https://acm.dingbacode.com/showproblem.php?pid=4794, quoting xyz2606: “we found a pattern and then found some special cases , and we printed a table for the special cases, then special cases of pattern of special cases of pattern, after 3~4 iterations of the process above we solved the problem....”

https://dmoj.ca/problem/ioi20p3 (the n^2 is not apparent for this, but can be guessed from looking at n<=10 examples, then some ~~work~~ segtreeing is still needed, but hey, at least things got started...)

Binary Search / Lagrangian Multipliers

Why should we use binary search? It helps us simplify the question of “what is the optimal value of X?” to simply “is X possible or impossible?” Sometimes thinking this way actually makes our problem more solvable; at other times it helps reduce an extra runtime factor of n to just a factor of log n.

Binary search on a variable x is applicable when a certain property is only true when x is above some cutoff (or below some cutoff) and false otherwise. Whenever this is the case, think of using binary search. But be careful not to apply binary search when this isn’t actually the case. Example: “given an n x m rectangle full of 0’s and 1’s, find the largest square within the rectangle filled entirely with 1’s” can be solved with binary search. This is because an 8x8 square filled entirely with 1’s means there exists a 7x7 filled square as well. But if we change the problem to “find the largest square whose border is all 1’s,” this property no longer holds.

Some concrete ways by which binary search get used:

Intermediate value theorem:
If I have a[0..n] where a[0] = 0 and a[n] = 1, by querying O(logn) values of a[i], I can find a location where a[i] = 0 & a[i + 1] = 1.

This is used in CEPC01 problem G: https://dmoj.ca/problem/tle17c5p6
Also IOI07 problem 1: https://oj.uz/problem/view/IOI07_aliens

Geometric problems
Such `intermediate value detection’ is very useful for geometric problems because many such problems are about detecting the boundary point of some region:

NWERC07 flight safety https://open.kattis.com/problems/flightsafety
(I think the Delaunay on WF18 https://icpc.kattis.com/problems/pandapreserve, can be done like this too, but haven’t coded this yet, can someone verify?)

Binary search for expected value when there is cyclic dependence
https://atcoder.jp/contests/abc189/tasks/abc189_f (see https://atcoder.jp/contests/abc189/editorial/589)
NWERC 2020 G

Use Lagrange multipliers to solve 'find best k things' in O(logn) calls. Set a cost for each item, then bin search on the cost so that exactly k items are picked in the optimum. Note that you can more or less guess a function is convex, after which you can do this.

https://dmoj.ca/problem/ioi16p6

CF Article on this
https://codeforces.com/blog/entry/49691

This idea was used, among other things, in
https://nadc21.kattis.com/problems/trainline
And also here
https://open.kattis.com/problems/blazingnewtrails

Money for Nothing – Kattis, Kattis
In some dp problems, the structure of applying monotonicity is like binary search stuffs (this sentence may need to rewrite)

https://dmoj.ca/problem/noi12p2 - Another problem using Lagrange multiplier and binary search but with a very different flavor