3. There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.

Suppose we have two television networks, whom we'll call $\mathcal{A}$ and $\mathcal{B}$. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a *schedule*—an assignment of each show to a distinct slot—so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed *rating*, which is based on the number of people who watched it last year; we'll assume that no two shows have exactly the same rating. A network *wins* a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network $\mathcal{A}$ reveals a schedule S and Network $\mathcal{B}$ reveals a schedule T. On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We'll say that the pair of schedules (S, T) is *stable* if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' such that Network $\mathcal{A}$ wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' such that Network $\mathcal{B}$ wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapley's question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

(a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

(b). As a counterexample:

case when ($S = [1, 3]$, $T = [4, 2]$):

- If Network B switches 4 and 2, we get $T' [2, 4]$.
- In this case, Network B under schedule of $T' [2, 4]$ can win more time slots than it

originally wins under the schedule T=[4,2].
. Therefore, this is a contradiction and proves that there is no stable pair of schedules.

4. Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is *stable* if neither of the following situations arises.

- First type of instability: There are students s and s', and a hospital h, so that
  - s is assigned to h, and
  - s' is assigned to no hospital, and
  - h prefers s' to s.
- Second type of instability: There are students s and s', and hospitals h and h', so that
  - s is assigned to h, and
  - s' is assigned to h', and
  - h prefers s' to s, and
  - s' prefers h to h'.

So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

the Algorithm would be:

- Both the student and the hospital would have their lists of preferences.
- the student S will ask the hospitals according to it's preference list.
- If the next hospital in S's list is not full, the hospital will match with S. If the hospital is full, it needs to look at its preference list.

If S is ranked higher than the lowest in the hospital's current student list, S will be accepted with another student being removed. The kicked out student will be back to the unmatched student list.

- Repeat the same loops until all students are matched to a hospital.

— the first type of instability is not possible, because by contradiction:

- We know from the problem that s is assigned to h, and student s' is assigned to no hospital, and h prefers s' to s.
- If s' is left unmatch, s' must have asked all hospitals including h.
- If s' has asked h and h prefers s', there are two possibilities:
  ① h has matched with s
  = in this case h should unmatch with the current student and match with s'
  ② h matches with s'.
  = in this case, s shouldn't have unmatched s' and match with s since s' is its perference.

— the second type of instability is not valid, because by contradiction:

• We know that s is assigned to h, and s' is assigned to h', and h prefers s to s', and s' prefers h to h'.

• s' would have asked h first before asking h' since h is the preference!

• When s' asks h, there are two possible cases:

① h has matched with s.
- But since s' is higher than s in h's list, so h should have unmatch with s and match with s'.

② h is not matched with any student, then it should be matched with s'.

• But in our case, h is matched with s, so we reach a contradiction.

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a *schedule* that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) *No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an algorithm to find them.

Example. Suppose we have two ships and two ports, and the "month" has four days. Suppose the first ship's schedule is

port P_1; at sea; port P_2; at sea

and the second ship's schedule is

at sea; port P_1; at sea; port P_2

Then the (only) way to choose truncations would be to have the first ship remain in port P_2 starting on day 3, and have the second ship remain in port P_1 starting on day 2.

The algorithm would be like:

1. Construct the prefix list of

a ship by timestamp.

(2). For port, the ship that comes ~~later~~ last get to stop there.

The preference list for port would be in reverse in respect to time.

BWC: If there is a ship s that visits port p, and s' is there also. Then p would have a preference list like [...s...s'], for s it's [p...p'] because we reach contradiction because s' and p can never match together given the preference lists we proposed in the algorithm!

4. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$g_1(n) = 2^{\sqrt{\log n}}$

$g_2(n) = 2^n$

$g_4(n) = n^{4/3}$

$g_3(n) = n(\log n)^3$

$g_5(n) = n^{\log n}$

$g_6(n) = 2^{2^n}$

$g_7(n) = 2^{n^2}$

$2^{\sqrt{\log n}} < n(\log n)^3 < n^{\frac{4}{3}} < n^{\log n} < 2^n < 2^{n^2} < 2^{2^n}$

① $2^{\sqrt{\log n}} < n(\log n)^3$ because

$n(\log n)^3 = (n(\log n))^3 = n^3(\log n)^3$.

Next, we divide both sides by $2^{\sqrt{\log n}}$ to isola
n^3 on one side.

$$\frac{2^{\sqrt{\log n}}}{2^{\sqrt{\log n}}} < \frac{n^3(\log n)^3}{2^{\sqrt{\log n}}}.$$

$$1 < \frac{n^3(\log n)^{2.5}}{2}.$$

Now we need to prove if the inequa
holds.

As n grows, $n^3(\log n)^{2.5}$ definitely grows faster than 2, so the inequality persists..

② $n(\log n)^3 < n^{\frac{4}{3}}$ because if we multiply them by log on both side:

$$\log n(\log n)^3 = \log n + 3\log(\log n) < \log(n^{\frac{4}{3}}) = \frac{4}{3}\log n = \log n + \frac{1}{3}\log$$

③ $n^{\frac{4}{3}} < n^{\log n}$ because.

left: $\log(n^{\frac{4}{3}}) = \frac{4}{3}\log n$

$<$

right $= \log(n^{\log n}) = (\log n)^2$.

④ $n\log n < 2^n$ because.

left: $\log(n^{\log n}) = \underline{(\log n)^2}$

$<$

right: $\log(2^n) = \underline{n\log 2}$.

(a) $2^n < 2^{n^2}$ because $n < n^2$

(b) $2^{n^2} < 2^{2^n}$ because $n^2 < 2^n$.

---

5. Using induction:

a. Prove that sum of the first n integers (1+2+....+n) is n(n+1)/2.
b. Find what the sum of $1^2+2^2+3^2+...+n^2=?$ is equal to. HINT: the result above is a factor for this part's solution.

a. Base Case: $n=1$.

$$1 = \frac{1(1+1)}{2}$$

$$= \frac{2}{2}$$

$$= 1.$$

Assume the formula holds for $n=k$, where k is an arbitrary number:

$$1+2+3+\dots+k = \frac{k(k+1)}{2}.$$

We want to prove the formula holds for $n=k+1$.

$$1+2+3+\dots+k+(k+1) = \frac{(k+1)(k+1)+1)}{2}$$

$= \frac{(k+1)(k+2)}{2}$.

we now need to prove that

$$\frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2}.$$

left: $\frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$

= right hand side $= \frac{(k+1)(k+2)}{2}$.

thus we proved the claim by induction.

b. From the hint, we know left hand side of the equation can be expressed in terms of the sum of the first n positive integers, which means there exists a k that

$$1^2 + 2^2 + 3^2 + \dots + n^2 = k \cdot \frac{n(n+1)}{2}.$$

If we try to plug in several values of n,

$S_1 = 1^2 = 1$

$S_2 = 1^2 + 2^2 = 5$

$S_3 = 1^2 + 2^2 + 3^2 = 14$

$k_1 \cdot \frac{1(1+1)}{2} = 1.$ $\quad k_1 = 1$

$k_2 \cdot \frac{2(2+1)}{2} = 5$ $\quad k_2 = \frac{5}{3}$ $\quad \frac{6}{2} = 3$

$k_3 \cdot \frac{3(3+1)}{2} = 14.$ $\quad k_3 = \frac{7}{3}.$

We observe the patten of k and how it's related by n. $k = \frac{2n+1}{3}$.

So $1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)}{2} \cdot \frac{2n+1}{3}.$

$= \frac{n(n+1)(2n+1)}{6}$

6. Given an array A of size N. The elements of the array consist of positive integers. You have to find the SMALLEST element with MINIMUM frequency. For example:

Example:

Input:

N=5, A=[0 0 5 50 1]

Output:

1

Explanation:

All values have frequency 1 except the values 0. 1 is the smallest element with minimum frequency.

① Create a dictionary for counting the frequency of each element in the array. Iterate over the array and update the dictionary:

- if we don't have the current iterating elemt in the dictionary, add it and update the count as 1.
- if we have it in dictionary, increament the count by 1.

② After we have the dictionary with all the frequencies, we need to iterate over it to find the ones.

with minimum frequency.

③. If there is only one number /elements with minimum frequency, we don't do anything, but if there are multiple elements with same minimum frequency, we need to set a variable that holds the current minimum frequency and iterate over the dictionary again.

- If the frequency is indeed the minimum frequency, we check if the current element we are iterating through is less than the smallest value we current have stored in the variable.

- If it's true, we update

the smallest value!

— If not, we don't do anything.