CS 180

Homework 1

Michael Inoue

405171527

09 October 2019

1 Solution

(3b) For every set of TV shows and ratings, there is not always a stable pair of schedules. We consider the following counterexample. Let schedule S be a set with ratings $s \in S$ for network A, and schedule T be a set with ratings $t \in T$ for network B. Let us define the pair (S, T) as $(S, T) := \{(s_i, t_i) \mid s_i \in S = \{s_1, s_2, ..., s_n\}, t_i \in T = \{t_1, t_2, ..., t_n\}$ for $i = 1, 2, ..., n\}$. We say that schedule S wins a time slot i over T if $s_i > t_i$ and that schedule T wins a time slot i over S if $t_i > s_i$.

Let schedule $S = \{1, 3\}$ and schedule $T = \{2, 4\}$. Then $(S, T) = \{(1, 2), (3, 4)\}$. We see that T wins time slots 1 and 2 over S. However, we see that this pairing is not stable, as S's schedule be adjusted so that it wins more time slots. Let this adjusted schedule $S' = \{3, 1\}$. Then $(S', T) = \{(3, 2), (1, 4)\}$. We see that S has won time slot 1 over T. However, we see once more, that this pairing is not stable, as T's schedule can be adjusted so that it wins more time slots. Let this adjusted schedule $T' = \{4, 2\}$. Then $(S', T') = \{(3, 4), (1, 2)\}$. We are back where we initially started, that is $(S, T) = (S', T')$. Effectively, S can simply adjust its two spots in response to T's schedule, and vice versa, and simply swapping the two positions in each set is sufficient to create a better schedule for either network. This can occur forever, and thus there are no stable pairings for each schedule.

2 Solution

Utilizing the algorithm below, one will always find a stable assignment of students to hospital (proof below).

Algorithm 1: Stable assignment of students to hospitals

```
Input  : List of students S and hospitals H
Output: Stable pairing of S and H
Initially every student s ∈ S and every hospital h ∈ H is free
while There is a student s without a hospital assignment left who hasn't applied to every hospital do
    Choose such a student s
    Let h be the highest ranked hospital in s's list that s hasn't applied to
    if h has room for more students then
        s is accepted by h
    else
        if h prefers s to its lowest preferred student, s' then
            s' is no longer accepted by h
            s is accepted by h
        else
            s is not accepted by h
        end
    end
end
return the set X of student-hospital assignments
```

We now prove that this algorithm yields a stable matching under our definition of stability.

Proof. We wish to prove that our algorithm will always yield a stable matching. We consider the criteria for stability:

An assignment of students to hospitals is stable if neither of the following situations arises:

1. There are students s and s', and a hospital h, so that
   (a) s is assigned to h, and
   (b) s' is assigned to no hospital, and
   (c) h prefers s' to s
2. There are students s and s', and hospitals h and h', so that
   (a) s is assigned to h, and
   (b) s' is assigned to h', and
   (c) h prefers s' to s, and
   (d) s' prefers h to h'

We first prove criteria (1). Suppose, for a contradiction, that there is indeed some $s, s' \in S$ and $h \in H$ such that situation (1) arises. Then either situation occurred:
(i): s' has not applied to h or
(ii): s' was assigned to h at one point and then was dropped at another point before the algorithm finished.

In case (i), s' has not applied to h, but according to the algorithm (line (2)), s' applies to every hospital that has not rejected or dropped it. Thus, this is a contradiction and case (i) cannot occur. In case (ii), at some point, s' was the least preferred student before being dropped. Let $s'' \in S$ be the new least preferred student assigned to h. If s is assigned to h, then s'' is dropped from h, and thus s is preferred over s''. But, if s is preferred over s'', then s is preferred over s', which is a contradiction. Thus, case (ii) is impossible.

Next, we prove criteria (2). Suppose, for a contradiction, that there is indeed some $s, s' \in S$ and $h \in H$ such that situation (2) arises. Then one of two situations occurred:
(i) s' didn't apply to h, or
(ii) s' was (a) rejected or (b) dropped by h. In case (i), because s' is assigned to h', s' must have applied to h' before h in order to not have applied to h'. But because s applies to its highest ranked hospital that it hasn't yet applied to, this implies that it prefers h' to h, which is a contradiction. Thus, case (i) can't occur.

In case (iia), if s' was rejected by h, then s' was preferred less than the least preferred student of h. Either s was assigned to h at the time, or was not. If s was assigned to h, then s is a preferred student of h; but if s' was rejected, then s is preferred over s', which is a contradiction. If s was not assigned when s' was rejected, then let $s'' \in S$ be the new least preferred student assigned to h. Because the least preferred only increases as more students are added, if s was rejected, then s'' is preferred over s. And if s was assigned, then s'' must be dropped. But this means that s' is preferred over s'' and thus over s, which is a contradiction.

In case (iib), suppose s is dropped from h and $s'' \in S$ is the new least preferred student assigned to h. If s is now assigned to h, then s'' is dropped from h, and thus s is preferred over s''. But, if s is preferred over s'', then s is preferred over s', which is a contradiction.

We conclude that neither of criteria (1) or (2) can occur and thus our algorithm yields stable assignments. $\square$

3 Solution

Terminology: Consider a list of ships S and ports P. We say that a ship $s \in S$ is docked at $p \in P$ if its journey stops there for the rest of the month (that is, its journey is truncated). We say that s is visiting p if it is merely stopping at p along its journey and is not docking there for the rest of the month. We say that s is free if its journey is not yet truncated and a p is free if it has no ship docked at it.

Now, consider a schedule X that holds following criteria:

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

We wish to truncate these journeys such that a ship can be docked at a port and remain there for the rest of the month, and thus obey this next criteria:

(2) Every ship is docked at a port at the end of the month.

We say that a schedule that obeys criteria (1) and (2) is a stable truncation. The algorithm below aims to solve this problem.

Algorithm 2: Stable truncation of ships' journeys

Input : List of ships S and ports P
Output: Stable truncation of S's journeys

Initially every ship $s \in S$ and every port $p \in P$ is free
while *There is a ship s that is free and hasn't tried to dock at every port p* **do**
    Choose such a ship s
    Let p be the first port in s's journey that s hasn't tried to dock at
    if *p is free* **then**
        s is docked at p and truncates its journey there
    else
        if *s visits p at a later day than the ship currently docked at p, s'* **then**
            s' is no longer docked at p and is free
            s is docked at p
        else
            s is not docked at p
        end
    end
end
return the set X of truncated journeys

Proof. We now prove that this algorithm always yields a stable truncation of ships' journeys. Suppose, toward a contradiction, that there is some schedule such that criteria (1) holds, but after truncation via the algorithm above, no longer holds. In other words, there is some $s \in S$ at p and $s' \in S$ at p, such that $s \neq s'$.

We consider the cases when this could occur. The first case (1) is that one ship, s is visiting the port p and another ship, s' is visiting the port p. In this case, both s's and s''s journeys are not truncated, and thus up to that point, are one-to-one with s's and s's journeys prior to truncation. But that implies that at some point, the schedule of s and s' had a conflict before truncation, which is a contradiction, as the schedule abides by criteria (1) prior to truncation.

The next case (2) we consider is when s is visiting p and s' is docked at p. In this case, s' must be docked at p prior to s's arrival (as we have already proved case (1) to be impossible). But if s' is docked at p, then by line (8) of the algorithm, s' must have visited p last. This of course, is a contradiction, as s' had to have arrived at p before s.

We now wish to prove criteria (2). Suppose, toward a contradiction, that after truncation via the algorithm above, there exists some $s \in S$, $p \in P$ such that s is free and p is free after the algorithm ends. By line (2) of the algorithm, s must have attempted to dock at every dock if it is free. Thus, the only way that s could be free is that it was dropped by p at some point for some s' that would visit later. But then p would have a docked ship s' and would thus not be free, which is a contradiction.

Thus, we conclude criteria (1) and (2) hold and that our algorithm yields stable truncations. $\square$

4 Solution

Functions listed in order of ascending growth rate, with 1 being the lowest and 7 being the highest:
1.) $g_1(n) = 2^{\sqrt{\log n}}$
2.) $g_3(n) = n(\log n)^3$
3.) $g_4(n) = n^{\frac{4}{3}}$
4.) $g_5(n) = n^{\log n}$
5.) $g_2(n) = 2^n$
6.) $g_7(n) = 2^{n^2}$
7.) $g_6(n) = 2^{2^n}$

Explanation: we see that $g_2(n)$, $g_7(n)$, and $g_6(n)$ are all exponential functions, which have one of largest growth rates, so we put them in spots 5, 6 and 7, respectively (as n is $O(n^2)$ and n^2 is $O(2^n)$, it follows that 2^n is $O(2^{n^2})$ and 2^{n^2} is $O(2^{2^n})$). Next, we analyze $g_4(n)$ and $g_5(n)$. We can analyze the respective logarithmic growth of said functions, $\log(g_4(n))$ and $\log(g_5(n))$, respectively, to determine their relative growth:
$\log(g_4(n)) = \log(n^{4/3}) = \frac{4}{3}\log(n)$
$\log(g_5(n)) = \log(n)\log(n) = \log(n)^2$.
We see that $\log(g_4(n))$ is $O(\log n)$ and $\log(g_5(n)) = O((\log n)^2)$. We thus determine that $\log(g_4(n)) = O(\log(g_5(n))$. Thus, $g_4(n) = O((g_5(n)))$. Applying the same argument, we determine that $\log(g_2(n)) = \log(2^n) = n$. $\log(g_5(n)) = O((\log(n))^2)$, and we conclude $\log(g_5(n)) = O(\log(g_2(n)))$, and thus $g_5(n) = O(g_2(n))$. Looking at $g_4(n)$, we can evaluate $g_4(n) = n^{\frac{4}{3}} = n(n^{\frac{1}{3}})$. Looking at $g_3(n)$, we see that $g_3(n) = n(\log n)^3$. We note that we have some term n multiply some expression for both $g_3(n)$ and $g_4(n)$. Analyzing said expression, we can determine each function's relative growth. We see that $\frac{g_3(n)}{n} = (\log(n))^3$ and $\frac{g_4(n)}{n} = n^{1/3}$. We note that logarithmic functions grow slower than polynomial functions; thus $\frac{g_3(n)}{n} = O(\frac{g_4(n)}{n})$, and so $g_3(n) = O(g_4(n))$. Finally, we consider $g_1(n)$. We note that $g_1(n) = 2^{\sqrt{\log n}} = O(2^{\log n})$. But assuming a base 2 logarithm, $2^{\log n} = n$, and so $g_1(n) = O(n)$ and thus $g_1(n) = O(n(\log n)^3) = O(g_3(n))$.

5 Solution

We consider the following problem:
We wish to find the lowest rung on a ladder such that an egg dropped at said rung will break. We have two eggs. An egg that breaks at a certain rung is assumed to break at a higher rung. To find such a rung, how many drops are necessary in the worst case?

We consider dropping an egg on some arbitrary rung, n. If the first egg breaks on the nth rung, then in the worst case, we must traverse up the remaining rungs from 1 to $n-1$, and thus we will have n drops in the worst case. In order to maintain this worst case number of drops, we traverse $n-1$ rungs up from the nth rung. If the egg breaks at the $n+(n-1)$th rung, then in the worst case, we must check rungs $n+1$ to $n+(n-2)$, or $n-2$ rungs in total. Adding the two additional drops from before yields $1+1+(n-2)=n$ drops total.

We can continue this process, jumping one less increment every time (e.g. jumping (n-2) rungs, then (n-3) rungs, etc.) In each case, if the egg breaks at the rung which we jump to, then we still have in the worst case n drops to find the lowest rung (we jump one less rung at a time to accommodate for the fact that we iteratively drop an additional egg with each jump; thus we are left with n drops total in the worst case).

Because we need to potentially check every rung of the ladder, we aim to choose a rung such that the sum of each jump will be equal to the size of the ladder. More formally,

$$n+(n-1)+(n-2)+...+1=k, \tag{1}$$

where k is the number of rungs in the ladder. The leftside of the equation (proved in solution (6a)) equates to $\frac{n(n+1)}{2}$. Thus, we have

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

Solving for n,

$$n^2+n=2k \tag{3}$$

$$n^2+n-2k=0 \tag{4}$$

$$n=\frac{-1\pm\sqrt{1+8k}}{2} \tag{5}$$

We choose the positive root and take the ceiling, as n $\in \mathbb{N}$, and we must round up to reach every rung of the ladder. Thus, we have

$$n=\left\lceil\frac{-1+\sqrt{1+8k}}{2}\right\rceil \tag{6}$$

(a) Now, for a 200 rung ladder, we simply plug in $k=200$ into equation (6). Thus, we have $n=\left\lceil\frac{-1+\sqrt{1+8(200)}}{2}\right\rceil=20$. Because we defined n to also be the number of drops in the worst case, we have 20 drops in the worst case.

(b) For a ladder with k rungs, by equation (6), we have $n = \left\lceil \frac{-1+\sqrt{1+8k}}{2} \right\rceil$, and thus we have $\left\lceil \frac{-1+\sqrt{1+8k}}{2} \right\rceil$ drops in the worst case.

6 Solution

(a) *Proof.* We wish to prove by induction the following equation:

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}, \tag{7}$$

for all $n \in \mathbb{N}$. To prove by induction, we take equation (7) to be our inductive hypothesis. Now, we see if the equation holds true for our base case, namely $n = 1$. Indeed,

$$\begin{align} 1 &= \frac{(1)(1+1)}{2} \tag{8} \\ &= 1, \tag{9} \end{align}$$

so our base case holds. Now, we assume our inductive hypothesis to be true for some arbitrary $n \in \mathbb{N}$ and must prove that

$$1 + 2 + 3 + ... + n + (n+1) = \frac{(n+1)(n+2)}{2}. \tag{10}$$

Now, utilizing the inductive hypothesis, the left-hand side of (10) can be rewritten as

$$\frac{n(n+1)}{2} + (n+1). \tag{11}$$

Further manipulation yields:

$$\begin{align} \frac{n(n+1)}{2} + (n+1) &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \tag{12} \\ &= \frac{n^2 + 3n + 2}{2} \tag{13} \\ &= \frac{(n+1)(n+2)}{2} \tag{14} \end{align}$$

Thus, we conclude equation (7) to be true for all $n \in \mathbb{N}$. □

(b)

$$1^3 + 2^3 + 3^3 + ... + n^3 = \left(\frac{n(n+1)}{2}\right)^2 \tag{15}$$

Proof. We wish to prove by induction the following equation:

$$1^3 + 2^3 + 3^3 + ... + n^3 = \left(\frac{n(n+1)}{2}\right)^2, \tag{16}$$

for all $n \in \mathbb{N}$. To prove by induction, we take equation (16) to be our inductive hypothesis. Now, we see if the equation holds true for our base case, namely $n = 1$. Indeed,

$$\begin{align} 1^3 &= \left(\frac{1(1+1)}{2}\right)^2 \tag{17} \\ &= 1. \tag{18} \end{align}$$

Thus, our base case holds. Now, we assume our inductive hypothesis to be true for some arbitrary $n \in \mathbb{N}$ and must prove that

$$1^3 + 2^3 + 3^3 + ... + n^3 + (n+1)^3 = \left(\frac{(n+1)(n+2)}{2}\right)^2 \tag{19}$$

Now, utilizing the inductive hypothesis, the left-hand side of (19) can be rewritten as

$$\left(\frac{n(n+1)}{2}\right)^2 + (n+1)^3. \tag{20}$$

Further manipulation yields:

$$\begin{aligned}
\left(\frac{n(n+1)}{2}\right)^2 + (n+1)^3 &= \frac{n^2(n+1)^2}{4} + \frac{4(n+1)(n+1)^2}{4} && (21)\\
&= \frac{(n^2 + 4(n+1))(n+1)^2}{4} && (22)\\
&= \frac{(n^2 + 4n + 4)(n+1)^2}{4} && (23)\\
&= \frac{(n+2)^2(n+1)^2}{2^2} && (24)\\
&= \left(\frac{(n+1)(n+2)}{2}\right)^2 && (25)
\end{aligned}$$

Thus, we conclude equation (16) to be true for all $n \in \mathbb{N}$.

$\square$