Name: Jahan Kuruvilla Cherian

Discussion 1D
4:00pm – 6:00pm
TA: Arman

CS 180 – Homework 1

Exercise 4 – Page 23

The solution for this problem follows very closely to the normal stable matching algorithm as formulated by Gale and Shapley. Essentially we list the hospitals and students, and we loop through from the start of the hospital list and go through each hospital along with each of its students from the preference list. We perform a matching between the current hospital and all of its students in order from highest priority to lowest priority. Continuing on to the next hospital we repeat the process, but taking into account the two possibilities. If the hospital tries to acquire a student in its priority list that is already assigned to a hospital, we begin to consider the perspective of the student. If the hospital that is trying to acquire the student is higher in the student's priority list than the one it is already matched to, we delete the previous match and match with the current hospital; else if the current hospital is lower down on the student's priority list, then the hospital won't match with the student, and will move on to the next student in its priority list. The pseudo code for such an algorithm would go as follows:

```
While hospital $h_i$ has availability
    $h_i$ offers spot to student $s_j$ from its priority list
    if $s_j$ has not been matched
        $s_j$ accepts and matches with $h_i$
    else
        if $h_i$ is higher in $s_j$'s priority list
            $s_j$ commits to $h_i$
            $h_i$'s availability increases by one
            previously committed hospital's availability decreases by 1
        else
            $s_j$ remains committed to previous hospital
```

The runtime of this algorithm would be *O(mn)*. The algorithm works under the rule of filling up each hospital's availability to its fullest, which will mean that even though it may lose out some student's to other colleges it will always remain with at least one student because there are fewer positions available than there are total number of hospitals as defined by the problem statement.
Verification of stability:

1.) If we have the case where hospital *h* accepts student *s* but prefers student *s'* we notice a clear contradiction. This is due to the fact that *h* will pick its students in order of preference, and because student *s'* is preferred over *s*, there will be a match between *h* and *s'* before there is a match with *s*.
2.) Let us assume the instable pair (*h, s*) and (*h', s'*). Assume that we start by running our matching algorithm on hospital *h*. In this instance, because preference matters, *h* will pick *s'* over *s*. When we go onto picking students for hospital *h'* we will see two possible conditions:
    a. *h'* wants *s'* over *s*. However, this condition will not be met, because *s'* prefers *h* to *h'* and thus will remain committed to *h*.
    b. *h'* wants *s* over *s'*. In this case, we have a successful match because *s* will not have been matched with any hospital, and will thus accept *h'*.

   This clearly shows that the instability pairs are impossible.

With the satisfaction of these, we see our algorithm matches every hospital to at least one student, while maintaining stability within *O(mn)* runtime.

Exercise 6 – Page 25

In order to find a perfect schedule for each ship such that we have a stable match, we model the problem as a stable matching problem, wherein stability is defined in the following sense – a ship has stopped off at a port that no other ship stops off at beyond the stopping day, until the end of the month.

In order to do this, we have to assign a priority for each ship and likewise for each port. For the ships, it's relatively straightforward in that we simply put the ports in its priority list based on chronological order of visiting days during the month. That is if port P_1 is a stopping point for ship S_1 before port P_2, then P_1 comes at a higher priority than P_2. For the priority of the ports, we order the ships in reverse chronological order. The algorithm then boils down to our generic matching problem algorithm (as described in Exercise 4's solution above, except each port accepts only one ship). Thus we can detail it as follows:

```
For port $P_i$ in all ports
    $P_i$ offers spot to ship $S_j$ from its priority list
    if $S_j$ has not been matched
        $S_j$ accepts and matches with $P_i$
    else
        if $P_i$ is higher in $S_j$'s priority list
```

```
            S_j commits to h_i
            Previously matched port now has availability to accept another ship
        else
            S_j remains committed to previous port
```

We can prove that this is a stable matching algorithm. Let us assume that the assignment is instable and violates the condition † as described in the problem. That is S_i passes P_k after S_j has already stopped at this port. But because P_k is higher in S_i's priority list, and vice-versa (because of the reverse chronological ordering for the ports), then we see that the initial matching would not occur and thus the possibility for an instable match runs into contradiction making it impossible.

Exercise 1 – Page 67

We look at the changes in runtime when (i) doubling the input size and (ii) when increasing the input size by 1. Thus we have the following:

a.) For the runtime of n^2 we have the following:
   i. $(2n)^2 = 4n^2 \rightarrow Slower\ by\ a\ factor\ of\ 4$
   ii. $(n+1)^2 = n^2 + 2n + 1 \rightarrow Slower\ by\ an\ additive\ of\ 2n+1$

b.) For the runtime of n^3 we have the following:
   i. $(2n)^3 = 8n^3 \rightarrow Slower\ by\ a\ factor\ of\ 8$
   ii. $(n+1)^3 = n^3 + 3n^2 + 3n + 1 \rightarrow Slower\ by\ an\ additive\ of\ 3n^2 + 3n + 1$

c.) For the runtime of $100n^2$ we have the following:
   i. $100(2n)^2 = 400n^2 \rightarrow Slower\ by\ a\ factor\ of\ 4$
   ii. $100(n+1)^2 = 100n^2 + 200n + 100 \rightarrow Slower\ by\ an\ additive\ of\ 200n + 100$

d.) For the runtime of $nlogn$ we have the following:
   i. $2nlog(2n) = 2nlog(n) + 2nlog(2) = 2nlog(n) + 2n \rightarrow Slower\ by\ a\ factor\ of\ 2\ and\ an\ additive\ of\ 2n$
   ii. $(n+1)\log(n+1) = nlog(n+1) + \log(n+1) \rightarrow Slower\ by\ an\ additive\ of log(n+1) + n[\log(n+1) - \log(n)]\ \{found\ by\ subtracting\ nlogn\ from\ the\ result\}$

e.) For the runtime of 2^n we have the following:
   i. $2^{2n} = (2^n)^2 \rightarrow Slower\ by\ the\ square\ of\ the\ previous\ runtime$
   ii. $2^{n+1} = 2^n \bullet 2 \rightarrow Slower\ by\ a\ factor\ of\ 2$

Exercise 8 – Page 69

a.) We start by dropping the glass jar from the nth rung, this way we can eliminate the highest rung. From this point onwards we drop it at $\sqrt{n}$, and then $2\sqrt{n}, 3\sqrt{n}\ldots$

We do this as an assumptive step, choosing the square root of the height. We then notice the general pattern that is in the first step we continue dropping at $j\sqrt{n}\ for\ 1 \leq j < \sqrt{n}$ and once we find a breaking point at $m\sqrt{n}$, then we know that the breaking point lies in between $(m-1)\sqrt{n}\ and\ m\sqrt{n}$ going up one rung at a time. This way we drop the glass jars at most $\sqrt{n}$ times for a total of $2\sqrt{n}$ times. This also applies for non-perfect square values of n, wherein we would have multiples of $\lfloor\sqrt{n}\rfloor$. Thus our overall algorithm will run in $O(\sqrt{n})$ time, which is indeed than linear time. Thus by a method of assumption we have proved that this algorithm works and is faster than $O(n)$.

b.) Let's assume that $f_k(n) \leq 2kn^{\frac{1}{k}}$ $[I]$.
Looking at $k = 1\ and\ k = 2$ obtaining runtimes of $n\ and\ \sqrt{n}$ respectively, we notice that our drop intervals can be multiples of $\lfloor n^{(k-1)/k}\rfloor$.
Now let us choose $k = 1$, and we see that the first jar will be dropped at most $\frac{2n}{n^{(k-1)/k}} \rightarrow 2n^{1/k}$ times. This narrows the set of rungs to an interval of length at most $n^{(k-1)/k}$. Now let us assume $[I]$ is true for k jars.
Now through induction we can try this equation for $k-1$ jars, seeing $2(k-1)(n^{\frac{k-1}{k}})^{\frac{1}{k-1}} \rightarrow 2(k-1)n^{1/k}$ drops in total. Looking at the $\leq 2n^{1/k}$ drops by the first jar, we have shown through induction that our upper bound of number of drops for a potentially infinite set of jars as $2kn^{1/k}$.

Problem 5 from website

We can employ an algorithm that employs a shrinking window solution. If we have two pointers at the head and tail of the sequence of real numbers, we first apply the black-box function to the entire sequence. If this returns YES, then we move our head pointer forward else we know that there exists no subset that sums to k in the whole array. We run the function again on the array[start+head:end] where in this instance head is 1 and start is 0. We keep doing this, incrementing head until we hit a NO. Upon hitting a NO, we move the head back and move the tail back. If now we have array[start+head:end-tail] and the blackbox returns YES to this, then we move the tail backwards until we hit our first NO from the black box, and then we have our subset.

As we can see from above, this is a linear runtime algorithm as we are just moving through the array in one sweep. If the subset exists towards the end, then we will only move the head forward and never move the tail backwards, and likewise for the opposite case but move the tail backwards. The tricky case is when the subset exists in the middle, but because we are looping with moving pointers, we will always terminate in *O(n)*.