# Recitation 3 (9/19-9/23)

## Reminders

List collaborators (only 2 other ppl and yourself) AND sources on your HW.

No collaborators for Q1. If you are found to have been discussing any part of question 1 with someone else, you will fail.

## Gale-Shapely Recap

No notes. Go over the algorithm in recitation.

## HW 2 - Q1

**Understanding the problem statement.**

m hospitals, each with a preference list of all n students
n students, each with a preference list of all m hospitals

Difference from Stable Matching Problem in class:
The number of hospitals and the number of students are different.
One hospital can have more than on open slot, but each student is only assigned to one hospital.
There are more students than there are total number of available slots for all m.

*Goal: output 1 stable matching assignment of hospitals to students.*

Given:
n hospitals and m students.

1. There are students $s$s and $s$'s′, and a hospital $h$h, so that:
    - s is assigned to h, and
    - s' is assigned to no hospital, and
    - h prefers s' to s.

E.g. h1(2 spots): $s_1 > s_2 > s_3 > s_4$ $s_1$: $h_2 > h_1$ $s_3$: $h_1 > h_2$

h2 (1 spot): $s_2 > s_4 > s_3 > s_1$ $s_2$: $h_2 > h_1$ $s_4$: $h_1 > h_2$

$(h_1, s_3)$ $(h_1, s_4)$ $(h_2, s_2)$

Unstable because $s_1$ was not assigned to anyone, but is highest on $h_1$'s preference list.

2. There are students ss and s′s′, and hospitals hh and h′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'.

E.g. h1(2 spots): $s_1 > s_2 > s_3 > s_4$ $s_1$: $h_2 > h_1$ $s_3$: $h_1 > h_2$

h2 (1 spot): $s_2 > s_4 > s_3 > s_1$ $s_2$: $h_2 > h_1$ $s_4$: $h_1 > h_2$

$(h_1, s_1)$ $(h_1, s_2)$ $(h_2, s_4)$

Unstable because $h_2$ prefers $s_2$ to current ($s_4$) and $s_2$ prefers $h_2$ to current ($h_1$)

Example: (input and output and why this is stable)

```
Input:

                3                       <- Number of hospitals
                5                       <- Number of students
                1   2   3   5   1   4   <- h1 with 1 opening & preference list
                1   5   1   2   4   3   <- h2 with 1 opening & preference list
                2   5   2   3   1   4   <- h3 with 2 openings & preference list

                2   1   3               <- s1 preference list
                3   2   1               <- s2 preference list
                3   1   2               <- s3 preference list
                1   2   3               <- s4 preference list
                1   2   3               <- s5 preference list

Output a stable matching:

              (1, 5)
              (2, 1)
              (3, 2)
              (3, 3)
```

Let's take a look at why this output is stable.

(1,5) -> Take a look at [2,3] who rank higher in h1's preference list. If 2 or 3 prefer 1 over their current partner or are unmatched, then there would be an instability.

(2,1) -> [5] is higher on h2's preference list. If s5 is unmatched or prefers h2 to their current partner, then there would be an instability.

Repeat for (3,2) & (3,3).

Go over this process fully in recitation.

### Asymptotic Notation:

**O(n) -** upper bound for worst case run time

an upper bound, not the exact growth rate of the function

**Ω(n) -** lower bound for worst case run time (**NOT THE BEST CASE RUNTIME)**

a low bound, not the exact growth rate

**Θ(n) -** exact run time - if you can show something is **O(n)** *and* **Ω(n)**

Simple example with search:

Given a string s, search for a character, c, in the string, most basic way. Iterate over list.

```
for a in s:
        if (a == e):
                return a
return -1
```

O(n) worst case we will need to iterate over every element in the list. As our list grows, our worst case scenario grows linearly.

Storing the length in a function before even storing the string. Store s = "cat" len = 3.

Calling len will take the same amount of time no matter how big the string is, therefore it takes constant time.

O(1) since worst case we find the length of the string in constant time.
Ω(1) since we cannot take less than constant time to find the length of the string
Since big O and big omega are the same, this solution runs in Θ(1)

Common runtimes of algorithms:

**O(n) - linear**

examples: computing the maximum in a list, merging two sorted lists

**O($n^2$) - quadratic**
Arises naturally from two nested for loops, why?

examples: brute force seeing two numbers in a list add up to another, brute force substring

**O(log n)**
Example in class with binary search. Recap of why this is true.

[1,2,3,4] → takes 2 steps with a list of size 4
[1,2][3,4] $\log_2 4 = 2$
[1][2][3][4]

[1,2,3,4,5,6,7,8] → takes 3 steps with a list of size 8 (double the size)
[1,2,3,4][5,6,7,8] $\log_2 8 = 3$
[1,2][3,4][5,6][7,8]
[1][2][3][4][5][6][7][8]

[1,2,3,4,5] best case: searching for 3, worst case: searching for 5
[3,4,5]
[4,5]
[5]

O(log n) & Ω(1)

**O(n log n)** - Common with any algorithm that splits the input into two and solves each piece recursively and combines the two pieces in linear time.

Ex: Mergesort, so also common with any algorithm that sorts first then performs a linear operation on the input since O(n log n) + O(n) = O(n log n)