CS 180 HW #1
Narek Daduryan

1 a) In the G-S algorithm, each man will propose at least once, and in the order of their preference list. Thus, $m_1$ will at some point propose to $w_1$, since it's his top preference. $w_1$ will accept, because $m_1$ is ranked higher than any other man (except $m_2$, who will propose to $w_2$ before $w_1$).
Similarly, $m_2$ will propose to $w_2$, who will accept, since she prefers $m_2$ to any other man (except $m_1$, who will propose to $w_1$ before $w_2$).
Once the matching $(m_1, w_1)$, $(m_2, w_2)$ is made, they will never unmatch, since $w_1$ won't accept any other proposals unless from $m_2$, and $w_2$ won't accept proposals from any man except $M_1$. However, neither $m_1$ nor $m_2$ will propose to another $w$ since they will not lose their partner. Thus, $(m_1, w_1)$ and $(m_2, w_2)$ will remain matched.

b) Proof by Contradiction!

Let's assume a stable matching where $m_1$ is matched to some $w_K$ (not $w_1$ or $w_2$) and $w_1$ is matched to some $m_K$ (not $m_1$ or $m_2$).

By their preference list, we know $m_1$ prefers $w_1$ to $w_K$, and similarly, $w_1$ prefers $m_1$ to $m_K$. Thus, since both $m_1$ and $w_1$ prefer each other over their current match, this is by definition an unstable match, and a contradiction.

Similarly, for any matching where $m_1$, $m_2$, $w_1$, $w_2$ are not matching is an unstable match, since each $m_{n \in 1,2}$ prefers either $w_{K \in 1,2}$ to any other outside $w$, and vice-versa.

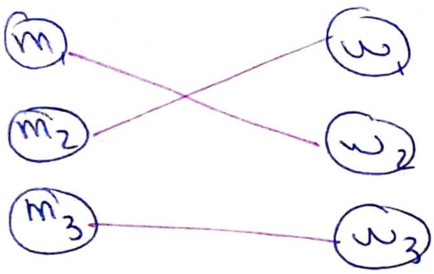
2a) There exists the following example:

$m_1: w_2 > w_1 > w_3$ $\qquad$ $w_1: m_1 > m_2 > m_3$

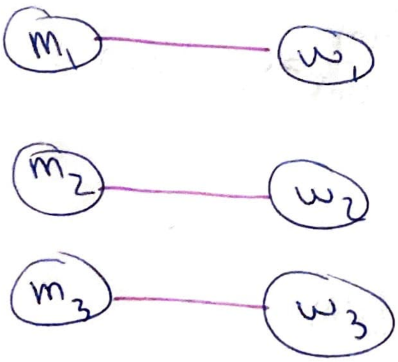
$m_2: w_1 > w_2 > w_3$ $\qquad$ $w_2: m_2 > m_1 > m_3$

$m_3: w_1 > w_3 > w_2$ $\qquad$ $w_3: m_3 > m_2 > m_1$

With this "truthful" preference list, running G-S algorithm results in following matching:




Let us now create an "untruthful" preference list, where $w_1$ claims to prefer $m_3$ to $m_2$. We update $w_1$'s preference list as follows: $w_1: m_1 > m_3 > m_2$ (Notice the swapped $m_2$ and $m_3$). Now running G-S algorithm, we get following matching:




Notice in the "truthful" (original) matching, $w_1$ ends up with $m_2$. In "untruthful" (modified) matching, she ends up with $m_1$, whom she truly prefers to $m_2$. Thus, the untruthful switch leads $w_1$ to have a better partner. ∎

3a) $f(n) = O(g(n))$ implies $\exists$ $c_1$ s.t. $c_1 \cdot g(n) > f(n)$ for $n >$ some $n_1$.

Thus, exponentiate both sides,

$$2^{(c_1 \cdot g(n))} > 2^{(f(n))} \quad \text{for } n > n_1$$

$$2^{c_1} 2^{g(n)} > 2^{f(n)}$$

let $c = 2^{c_1}$. Thus, $c\, 2^{g(n)} > 2^{f(n)}$. Thus, by definition

$2^{f(n)} = O(2^{g(n)})$. B.

3b) Test $n^n = \Theta(n!)$

Use limit check: $\lim_{n \to \infty} \frac{n!}{n^n} = \frac{1 \times 2 \times 3 \times \ldots \times n}{n \times n \times n \times \ldots \times n} = 0$

(Note: this limit can be solved using squeeze theorem, but I decided to use a limit calculator.)

Thus, by the limit check, if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then

$f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$.

Thus, $n! = \Omega(n^n)$ and $n! \neq \Theta(n^n)$.

We know if $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$.

Thus, since $n! \neq \Theta(n^n)$, we know

$n^n \neq \Theta(n!)$ B.

3c) $f(n) = O(g(n))$

$\exists\ c_1$ s.t. $c_1 \cdot g(n) > f(n)$ for $n >$ some $n_1$

Multiply both sides by $g(n)$.

$c_1 \cdot g(n) \cdot g(n) > f(n) \cdot g(n)$. for $n >$ some $n_1$

$c_1 \ (g(n)^2) > f(n) \cdot g(n)$.

Thus, by definition of big-O,

$f(n) \cdot g(n) = O(g(n)^2)$. $\square$.

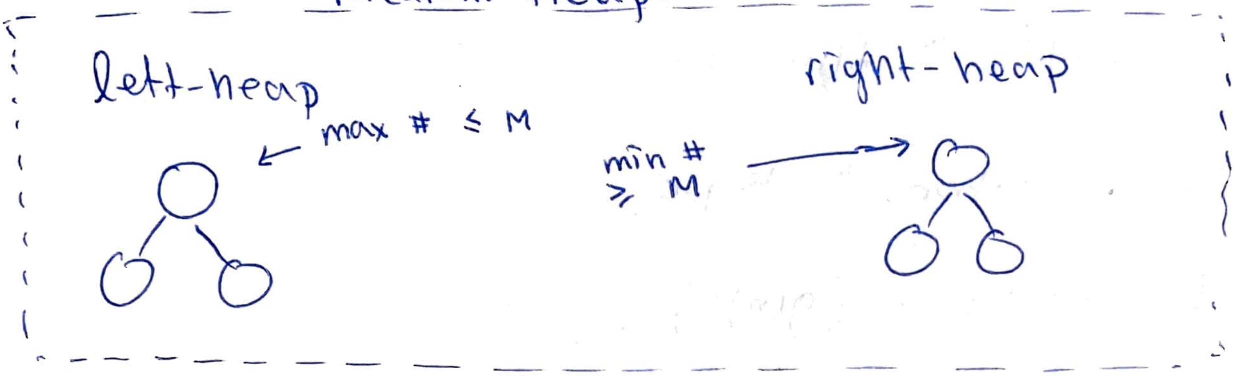
4). Idea: use both max-heap and min-heap.

For a range of numbers A, and median* (A) = M,

Store a left_heap, which is a max-heap of numbers in A ≤ M.

Store a right_heap, which is a min-heap of numbers in A ≥ M.

Medium Heap.




Define algorithm push (n):

let m = find_medium ← O(logn) by specification

if (n < m) left_heap.push(n) ← O(log n) by design of max- or min- heap

else right_heap.push(n) ←

if (left-heap.size() > right-heap.size() + 1) ← size should be O(1) as heap can store internal size var.

let node = left_heap.pop() ← O(log n), since removal is O(log n)

right_heap.push (node) ← O(log n)

else if (right_heap.size() > left_heap.size() + 1)

let node = right_heap.pop()

left_heap.push (node)

Since all operations in push(n) are O(logn), and do not occurr more than once, push is $O(c_1 \cdot \log n)$, or just O(logn), n being the number of elements.

Define algorithm find-medium:

size is $O(1)$

```
if (left-heap.size() == right-heap.size() + 1)
    return left-heap.first()
else if (right-heap.size() == left-heap.size() + 1)
    return right-heap.first()
else
    return (left-heap.first() + right-heap.first()) / 2
```

getting root of heap is $O(1)$

Since all operations in find-medium are $O(1)$ and occurr at-most once, find-medium is $O(c_1 \cdot 1) = O(1) = O(\log n)$.

5) Define function find_i (A,)

```
let n = A.size()
let left = 0
let right = n

while (left < right)
    let i = (right - left) / 2 + left

    if (i == A[i]) return True
    else if (i < A[i]) right = i
    else if (i > A[i]) left = i
return False
```

Notice in every iteration we split the input array A in 2, and continue iterating until there are no elements left. Since we divide by 2 on each iteration, we will iterate $\log_2(n)$ times, where $n = |A|$. Thus, find-i is $O(\log n)$.

Since A is sorted, distinct elements, for any element index i, every element will be strictly greater to the right, and strictly less on the left. Thus, if $A[i] > i$, then every element $j > i$ (to the right of i) will also have property $A[j] > j$, since each element is greater than previous. Similarly, if $A[i] < i$, every element to its left will also have $A[j] < j$. By this rule we can eliminate half of the array every time until we find a match or exceed the array.