Discussion Session summary

- (1a) Consider proving the following simpler version first :
  Prove that there is no algorithm that makes just 1 chip-testing query that can determine the working chip correctly.
  In order to do this, look at the possible cases when just 1 query is made - in which of these cases can the working chip be identified?

  Next try to prove that no algorithm that makes t queries is able to correctly determine the working chip. Note that since one chip is defective, it may not be consistent with its output across the t queries. How does consistency help/hurt the problem of identifying the working chip?

- (1b) for the chip testing problem - it must be clear by now that we cannot count on the chips to behave in a certain way - our algorithms should be able to force what we need out of the chips despite their arbitrary behaviour.

  Consider the algorithm discussed in class and the guarantees it provides. Can we use these guarantees to come up with a Working-Defective pair ? Try to analyze the cases where the algorithm succeeds and where it fails separately and for each case show how to come up with a Working-Defective pair.

- (2a) Main idea is to first identify what the final order will look like so that justification is straightforward - for eg. if the final order is f_1, f_2, f_3 then justification will just be $f_1 = O(f_2), f_2 = O(f_3)$.
  Classify each of the functions into the following "buckets":
  log bucket - functions that are either $\Theta(\log n)$ or even smaller eg : $\log(\log n)$

  poly log bucket - functions of the form $\Theta((\log n)^c)$ for some constant c

  sub-polynomial bucket - functions that are larger than poly log but smaller than any polynomial n^c eg : $2^{\sqrt[3]{\log n}}, n^{\frac{1}{\log \log n}}$

  polynomial bucket - functions of the form $\Theta(n^c)$ for some constant c

  quasi-polynomial - functions of the form $2^{poly(\log n)}$ eg : $n^{\log n} = 2^{(\log n)^2}$

  exponential and beyond - functions that look like 2^n or larger eg: $100^n, n!$

  Once this is done, try to establish the order within each bucket and that will lead to the overall order of the functions (since the buckets themselves can be easily placed in order)
  Note : You DO NOT need to compare every pair of functions in order to establish the order.

- (2b) For each function, you need to determine whether each of the four asymptotic equations hold by checking whether $lim_{n\to\infty}\frac{f(n)}{f(g(n))} = k$ for some positive constant k for the different choices of $g(n)$ i.e. $g(n) = \{n-1, \frac{n}{2}, \sqrt{n}, \log n\}$

  In particular, to check if $f(n) = \Theta(f(n-1))$, check whether
  $lim_{n\to\infty}\frac{f(n)}{f(n-1)} = k$ (for some positive constant k)

  Similarly, to check if $f(n) = \Theta(f(\frac{n}{2}))$, check whether
  $lim_{n\to\infty}\frac{f(n)}{f(\frac{n}{2})} = k$ (for some positive constant k)

  Example : For $f(n) = n\log n$, we have

$$lim_{n\to\infty}\frac{f(n)}{f(n-1)} = \frac{n\log n}{(n-1)\log(n-1)} = 1$$

$$lim_{n\to\infty}\frac{f(n)}{f(\frac{n}{2})} = \frac{n\log n}{(\frac{n}{2})\log(\frac{n}{2})} = lim_{n\to\infty}\frac{2\log n}{\log n - 1} = 2$$

$$lim_{n\to\infty}\frac{f(n)}{f(\sqrt{n})} = lim_{n\to\infty}\frac{n\log n}{\sqrt{n}\log(\sqrt{n})} = lim_{n\to\infty}\frac{\sqrt{n}\log n}{\frac{1}{2}\log n} = \infty$$

$$lim_{n\to\infty}\frac{f(n)}{f(\log n)} = lim_{n\to\infty}\frac{n\log n}{(\log n)\log(\log n)} = lim_{n\to\infty}\frac{n}{\log\log n} = \infty$$

  We conclude that for $f(n) = n\log n$,
  $f(n) = \Theta(f(n-1))$ $\quad f(n) = \Theta(f(\frac{n}{2}))$ $\quad f(n) \neq \Theta(f(\sqrt{n}))$
  $f(n) \neq \Theta(f(\log n))$

- (3b) for time complexity analysis, let $T(n)$ denote the time it takes to run this algorithm on an array of size n. Write down the equation that computes the value of $T(n)$ carefully and then establish bounds for the same (this equation may be a recurrence relation).

- (3c) Try to exploit the promise that no element has initial position more than k positions away from its final position - in this case, be careful to state EXACTLY what quantity is bounded due to the promise and what this bound is. This bound would then imply the required asymptotic upper-bound for the time complexity of recursive insertion sort.

- (4) brute force algorithm : for every ship S, check every base and add the amount of gold of that base if the base's defence is strictly smaller than ship's attack.
  Time complexity = $O(mn)$ (not efficient enough)
  clearly, this algorithm isn't efficient enough - so need to bring down its running time. How?
  Well, note that if a ship can attack a base with power P, it can also attack every base with power strictly less than P. Also, once you have figured out

what all bases a ship can attack, it can be time consuming to determine the total gold it can collect. So, try to come up with a way to quickly compute the gold that any ship with power P can collect.