CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #1

September 19, 2019

Name: BUCH HIMESH *Extension: Yes*

Solution 1a:

Since we know that one of the chips is defective, and we also know that defective chips return arbitrary result, meaning they can either return "working" or "defective" about the chip that is attached to it, there is no way to know which chip is working because of the arbotary behaviour of the defective chip.

These are the results we get when a (working, defective) pair is connected,

- (defective, working)
- (defective, defective)

Working chip will always "defective" about other chip, but since there is no (working, working) pair, we can't determine which chip is working and which isn't. The only way to determine if a chip is working or not is by having more working chips than defective, which is not true in this case.

By considering the chip testing example done in class,

> We proved that if the returning pair is (working, working), then and only then it means that both chips are working, and we set any one of those two chips aside and test it with every other chip, in order to find the defective chip. And, for every other result, such as (working, defective), (defective, working), (defective, defective) we discard both the chips. Since, in this case, there is no way we are going to have a (working, working) pair, we will end up discarding all the chips (according to the algorithm) and won't get any output.

Hence, there is no way of proving that the chip is working.

Solution 1b: From the chip testing algorithm that we did in class, we basically saw that if the output is a (working, working) chip, we take one chip out, and for any other output we discard both the chips in the pair.

Now, following the same argument, if we have a (working, working) pair, we will take one chip out and connect it with every other chip. We know that working chip will always return *defective* if the other chip is actually defective. Hence, we got the pair of a (working, defective) chip.

There are basically four case that can be,

- (working, defective)
- (defective, working)
- (defective, defective)
- (working, working)

Here is the argument if we can't find a working chip:
For the cases, (working, defective) and (defective, working), we return the pair itself, which proves the hypothesis. For any other or (defective, defective) pair, we discard both of them.

Hence, proved

Solution 2a:

- $\lim_{n\to\infty} \frac{10^{10^{10}}}{\log\log n} = 0$

  $Hence, f(10^{10^{10}}) = O(f(\log\log n))$

- $\lim_{n\to\infty} \frac{\log\log n}{\log n^{100}} = 0$

  $Hence, f(\log\log n) = O(\log n^{100})$

- $\lim_{n\to\infty} \frac{\log n^{100}}{(\log n)^{100}} = 0$

  $Hence, f(\log n^{100}) = O(f((\log n)^{100}))$

- $\lim_{n\to\infty} \frac{(\log n)^{100}}{(2^{\sqrt{(\log n)}})} = 0$

  $Hence, f((\log n)^{100}) = O(f((2^{\sqrt{(\log n)}})))$

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

  $Hence, f((2^{\sqrt{(\log n)}})) = O(f(\sqrt{n}))$

- $\lim_{n\to\infty} \frac{\sqrt{n}}{n/\log n} = \frac{\log n}{\sqrt{n}} = 0$

  $Hence, f(\sqrt{n}) = O(f(n/\log n))$

- $\lim_{n\to\infty} \frac{n/\log n}{2n} = \frac{1}{2\log n} = 0$

  $Hence, f(n/\log n) = O(2n)$

- $\lim_{n\to\infty} \frac{2n}{10n} = 1/5$

  $Hence, f(2n) = O(f(10n))$

- $\lim_{n\to\infty} \frac{10n}{n^2} = \frac{10}{n} = 0$

  $Hence, f(10n) = O(f(n^2))$

- $\lim_{n\to\infty} \frac{n^2}{n^{1/\log\log n}} = 0$

  $Hence, f(n^2) = O(f(n^{1/\log\log n}))$

- $\lim_{n\to\infty} \frac{n^{1/\log\log n}}{n^{\log\log n}} = 0$

  $Hence, f(n^{1/\log\log n}) = O(f(n^{\log\log n}))$

- $\lim_{n\to\infty} \frac{n^{\log\log n}}{2^n} = 0$

  $Hence, f(n^{\log\log n}) = O(f(2^n))$

- $\lim_{n\to\infty} \frac{2^n}{10^n} = 0$

  $Hence, f(2^n) = O(f(10^n))$

- $\lim_{n\to\infty} \frac{10^n}{n!} = 0$

  $Hence, f(10^n) = O(f(n!))$

- $\lim_{n\to\infty} \frac{n!}{n^n} = 0$

  $Hence, f(n!) = O(f(n^n))$

Solution 2b:

- For function n! :

  (a) $\lim_{n\to\infty} \frac{n!}{(n-1)!} = \infty$;which is not constant

  $Hence, f(n!) \neq \Theta(f((n-1)!))$

  (b) $\lim_{n\to\infty} \frac{n!}{(n/2)!} = 2$;which is constant

  $Hence, f(n!) = \Theta(f((n/2)!))$

  (c) $\lim_{n\to\infty} \frac{n!}{\sqrt{n}!} = 1$;which is constant

  $Hence, f(n!) = \Theta(f(\sqrt{n}!))$

  (d) $\lim_{n\to\infty} \frac{n!}{(\log n)!} = \infty$ (Here n! goes faster than $\log n!$, so n! takes precedence, which tends to ∞)

  $Hence, f(n!) \neq \Theta(f(\log n)!))$

- For function $\log n$:

  (a) $\lim_{n\to\infty} \frac{\log n}{\log(n-1)} = 1$;which is constant

  $Hence, f(\log n) = \Theta(f(\log n - 1))$

  (b) $\lim_{n\to\infty} \frac{\log n}{\log n/2} = \frac{\log n}{\log n - \log 2} = 1$;which is constant

  $Hence, f(\log n) = \Theta(f(\log n/2))$

(c) $\lim_{n\to\infty} \frac{\log n}{\sqrt{(}\log n)} = \frac{\log n}{\log n/(1/2)} = 2$;which is constant

$Hence, f(\log n) = \Theta(f(\sqrt{(}\log n)))$

(d) $\lim_{n\to\infty} \frac{\log n}{\log\log n} = \infty$ (Here $\log n$ goes faster than $\log\log n$, so $\log n$ takes precedence, which tends to ∞)

$Hence, f(\log n) \neq \Theta(f(\log\log n))$

- For function 2^n :

(a) $\lim_{n\to\infty} \frac{2^n}{2^{(n-1)}} = \frac{2^n}{2^n/2} = 2$;which is constant

$Hence, f(2^n) = \Theta(f(2^{n-1!}))$

(b) $\lim_{n\to\infty} \frac{2^n}{2^{n/2}} = 1$;which is constant

$Hence, f(2^n) = \Theta(f(2^{n/2}))$

(c) $\lim_{n\to\infty} \frac{2^n}{2\sqrt{n}} = \infty$ (Here 2^n goes faster than $\sqrt{2}^n$, so 2^n takes precedence, which tends to ∞)

$Hence, f(2^n) \neq \Theta(f(2^{\sqrt{n}}))$

(d) $\lim_{n\to\infty} \frac{2^n}{2^{\log n}} = \frac{2^n}{n} = \infty$ (Here 2^n goes faster than n, so 2^n takes precedence, which tends to ∞)

$Hence, f(2^n) \neq \Theta(f(2^{\log n}))$

- For function n^2 :

(a) $\lim_{n\to\infty} \frac{n^2}{(n-1)^2} = \frac{(n-1)^2}{n^2-2n+1} = 1$;which is constant

$Hence, f(n^2) = \Theta(f((n-1)^2))$

(b) $\lim_{n\to\infty} \frac{n^2}{(n/2)^2} = 4$;which is constant

$Hence, f(n^2) = \Theta(f((n/2)^2))$

(c) $\lim_{n\to\infty} \frac{n^2}{\sqrt{n^2}} = \infty$;which is not constant

$Hence, f(n^2) \neq \Theta(f((\sqrt{n})^2))$

(d) $\lim_{n\to\infty} \frac{n^2}{(\log n)^2} = \infty$ (Here n^2 goes faster than $(\log n)^2$, so n^2 takes precedence, which tends to ∞)

$Hence, f(n^2) \neq \Theta(f((\log n)^2))$

- For function 10 :

(a) $\lim_{n\to\infty} \frac{10}{9} = 1.11$;which is constant

$Hence, f(10) = \Theta(f(9)!)$

(b) $\lim_{n\to\infty} \frac{10}{5} = 2$;which is constant

$Hence, f(10) = \Theta(f(10/2))$

(c) $\lim_{n\to\infty} \frac{10}{\sqrt{10}} = \sqrt{10}$;which is constant

$Hence, f(10) = \Theta(f(\sqrt{10}))$

(d) $\lim_{n\to\infty} \frac{10}{\log 10} = 10$;which is constant

$Hence, f(10) = \Theta(f(\log 10))$

- For function 2^{2^n} :

(a) $\lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{n-1}}} = 1$;which is constant

$Hence, f(2^{2^n}) = \Theta(f(2^{2^n} - 1))$

(b) $\lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{n/2}}} = \frac{2^{2^n}}{2^{\sqrt{2}^n}} = 1$

$Hence, f(2^{2^n}) = \Theta(f(2^{2^{n/2}}))$

(c) $\lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{\sqrt{n}}}} = 1$;which is constant

$Hence, f(2^{2^n}) = \Theta(f(2^{2^n}/2))$

(d) $\lim_{n\to\infty} \frac{2^{2^n}}{2^{2^{\log n}}} = \frac{2^{2^n}}{2^n} = \infty$

$Hence, f(2^{2^n}) \neq \Theta(f(\log(2^{2^n})))$

Solution 3a:

- **Base Case:** when array is of size n = 1; it is already sorted, and the algorithm returns array, which is sorted
- **Inductive Step:** There are basically two things happening in insertion sort,
  1. Sorting the A[1,. . . ,(n-1)] array
  2. Inserting n^{th} element at the end of the sorted array

Suppose $A[1, \ldots, (i-1)]$ is sorted. By looking at the recurrsive for loop, we notice that it goes by moving A[i-1], A[i-2], A[i-3] and so on by one position to the right until it finds the proper position for A[i]. Once it is done shifting, it inserts the value of A[i]. The subarray $A[1, \ldots, i]$ is in sorted order now. Hence, we proved for the case where there an array consist of n elements.

Now for n+1 elements,
The condition causing the recurrsive algorithm to terminate is that i > n. Because each loop iteration increases i by 1, we must have i = n+1 at that time. We have already shown that the subarray A[1,. . . ,n+1 -1] = A[1,. . . ,n] consists of the elements originally in A[1,. . . ,n], but in sorted order.
In other words, the algorithm works the same way, first sorts the aray of n elements and then inserts $n + 1^{th}$ element in the sorted array. Which proves the algorithm for any n+1 elements.

Solution 3b: If there is only one element in the array (base step in induction), it will take O(1), constant, time to sort it.

For any other n number of elements, we have an array of size [n-1], and in order to sort that array, it will take some time T[n-1], which will be the worst case. This step follows recurssion, and after two iterations, the array will have [n-2] elements, and the worst case will be T[n-2]. Since it is a recurrsive algorithm, the above process repeats and the time for each iteration will be T[n-i].

The final step returns the sorted array, which takes some O(1) constant time. Now, combining all the results, we get,

$$O(n) + O(n-1) + O(n-2) + ... + O(1) = O(n^2)$$

Solution 3c:

- We are given a promise here that each element $A[i]$ is at most k indices away from its index

  We also stated in part 3b, that in order to sort single element, it takes O[1], some constant time. So, for n elemtents it clearly takes O[n] time (linear).

- According to the promise, in order to move the element some k elements, we need to repeat the process k times. Hence,

$$\begin{aligned} \text{Total time} &= kO(n) \\ &= O(kn) \end{aligned}$$

  Hence, proved.

Solution 4: The basic approach here is to compare the power of each ship to each base and see if the base can be attacked. In order to do that, we run a linear search algorithm, which compares each power with each defensive strength, and checks if pi > di. Since we are doing a linear search here,

Total time of comparing = O(1)
Repeating the process m (number of base) times = m * O(1) = O(m)
Since we have n ships = n * O(m) = O(nm)

Hence, Total time = O(mn)

This algorithm works but is not efficient enough. So, another approach is to sort the array of defensive power of each base. We use merge sort to do that. Sorting array d (using merger sort),

Total time of sorting: $mO(\log m)$ (proved in class)

Now, we use binary search to compare the power of each ship with defensive strength of each base. Binary search (Time complexity= $O(\log m)$) is faster than linear search (Time complexity = O(n)). We will also sort the array of Gold.

Following the binary search algorithm, we will check *if d[i]* $\geq$ *p[i]*, which is the first element in array P. We need to find this index i which satisfies the above argument. Because, once we find this index, we will know (from binary search), that every indices from i to 0, can be stolen. Thus, we can simply add them to get the total amount of gold. Again, we can do this because of the binary search's property, which gives us bases that has defensive power strictly less than ship's power. Hence total time complexity,

Total time binary search for one ship $= O(\log m)$
Since we have n ships, Total time of search $= nO(\log m)$

$$\begin{aligned}\text{Total time} &= \text{Time of sorting} + \text{Time of searching} \\ &= mO(\log m) + nO(\log m) \\ &= ((m+n)\log m)\end{aligned}$$

Hence, proved.