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

Lecture 2

September 9, 2019

Instructor: Sepehr Assadi

1 Chip Testing Problem (Continued)

Recall the chip testing problem from the last lecture.

Problem 1 (Chip Testing)**.** We are given n chips which may be *working* or *defective*. A working chip behaves as follows: if we connect it to another chip, the original chip will *correctly* output whether the new connected chip is working or is defective. However, if we connect a defective chip to another chip, it may output any arbitrary answer. For instance, if we connect a defective chip to a working chip, the answers can be either (defective, defective) or (working, defective) (the first coordinate shows the answer of first (defective) chip and the second coordinate is for the second (working) chip).

We are promised that *strictly* more than half the chips are working. Design an algorithm for finding one of the working chips.

So far, we developed some intuition for the problem (first step) and designed an algorithm for it (second step). In the next step, we prove the correctness of this algorithm. The algorithm is repeated here for your convenience.

1. **Base case:** If $n = 1$ return the single chip; if $n = 0$ return 'empty'; otherwise, continue as follows.
2. Pair the chips arbitrarily into $\lfloor n/2 \rfloor$ groups of size 2 leaving one aside in case n is odd. Connect each pair together and test the answers.
3. For each pair, if the chips output (working, working), *keep* one of the chips in the pair arbitrarily and *discard* the other one. Otherwise, if the answer is anything else, *discard* both the chips from the pile.
4. Run the algorithm recursively on the set of remaining chips (ignoring the set aside chip).
5. If the recursive call returns a chip, output that chip as the final answer.
6. Otherwise, if the recursive call returns 'empty' do as follows: if n was originally odd and we set aside one chip, output that chip as the answer; otherwise if n was even, return 'empty'.

Third Step: Proving the Correctness of the Algorithm

Proving the correctness of an algorithm is one of the main steps (if not *the* main step) in algorithm design. An algorithm without a proof of correctness is *never* complete. Let us prove the correctness of the above algorithm. We prove this *by induction* on the values of n. Our induction hypothesis is as follows:

Induction hypothesis: For any number of chips $n \geq 0$:

1. if the number of working chips is *strictly more* than defective ones, then the algorithm returns a working chip; moreover,

2. if the number of working chips is *equal* to the number of defective chips, then the algorithm *either* returns a working chip or returns 'empty'.

Let us emphasize that the induction hypothesis above is rather delicate; even though we technically only care about the case when number of working chips is more than the defective ones (namely the first part), proving the correctness of the algorithm requires us to even argue something about the case when number of working chips is equal to the defective ones (namely the second part). This is a common occurrence in analyzing algorithms (and in fact many other mathematical proofs specially the ones based on induction: in order to prove a specific statement, sometime it helps to consider more general variants of the statement).

Let us first consider the base case(s), namely $n = 0$ and $n = 1$.

- If $n = 1$, then we only need to prove the first part of the induction hypothesis (because number of working chips is one while there is no defective chip): this is certainly true as the algorithm returns the only working chip in this case.
- If $n = 0$, then we only need to prove the second part of the induction hypothesis (because both number of working chips and defective ones are zero, hence, technically, they are equal): again this is true since the algorithm returns 'empty' in this case.

Notice that similar to most inductive proofs, proving the base case is typically very easy (but nevertheless *crucial* – you have probably seen "wrong proofs" in your previous courses that arise when we do not pay attention to the base case of inductions; so never forget to prove the base case of inductions as well even if it sounds completely straightforward to you).

The main step is to prove the induction step. Suppose the induction hypothesis is true for all n up to i and we will prove it for $i + 1$. We emphasize that we need to prove *both* parts of the induction hypothesis. The following claim is the main part of the argument.

Claim 1. *When grouping the chips into groups of size* 2 *and discarding them according to rules in Line* (3)*:*

1. *if the original number of working chips was strictly more than defective ones, then the remaining number of working chips is also* strictly more than *the defective ones; similarly,*
2. *if the original number of working chips was at least as large as the defective ones, then the remaining number of working chips is also* at least as large as *the working ones.*

(the original number of chips refer to the case after we set aside one chip when n *is odd).*

Proof. The high level idea of the proof is as follows: whenever we discard both chips in Line (3) we can be sure that one of them is at least defective (as a working-working pair never outputs anything other than (working, working)). As such, in this case, we are sure that we discarded *at least as many* defective chips as working chips. In the other case, either both chips are working or both are defective and thus removing one of them keeps the ratio of working to defective chips unchanged. Let us now formalize this.

Let W denote the original number of working chips and D denote the number of defective chips. Similarly, define G_1 as the number groups in which both chips are working, G_2 as the ones in which both chips are defective, and G_3 as the remaining groups, namely, the ones with one of each types of chips (note that this classification is based on the *actual* status of the chips not what they report). We first have,

$$W = 2G_1 + G_3 \qquad \text{(since both chips in } G_1 \text{ are working and exactly one chip in } G_3 \text{ is working)}$$
$$D = 2G_2 + G_3 \qquad \text{(since both chips in } G_2 \text{ are defective and exactly one chip in } G_3 \text{ is defective)}$$

On the other hand, notice that in the remaining chips, we keep *exactly* one chip from each group in G_1 (because those chips definitely answer (working, working)), *at most* one chip from each group in G_2 (because those chips may answer (working, working) as they are both defective, but potentially, they may answer something different as well), and *no* chip from groups in G_3 (because those chips have one working and one defective chip and thus the working chip would declare defective always). Let W' be the new number of working chips and D' be for defective ones. We have $W' = G_1$ and $D' \leq G_2$. Moreover, if $W > D$, we have,

$$W > D \implies 2G_1 + G_3 > 2G_2 + G_3 \implies G_1 > G_2 \implies W' > D',$$

proving the first case. A similar calculation when $W \geq D$ shows that $W' \geq D'$, finalizing the proof. $\square$

We are now ready to prove the induction hypothesis. Considering that the algorithm behaves slightly differently when n is odd versus when it is even, it makes sense to consider these cases separately.

- *When n is odd*: Note that when n is odd, it cannot ever be the case that the number of working chips is equal to defective chips (otherwise total number of chips would definitely be even), thus we only need to prove the first part of hypothesis for odd n. Let C denote the chip we set aside in this case.

  Suppose first that C was a working chip. As we had strictly more working chips then defective ones, after removing C, number of working chips is at least as large as the defective ones. By Part (2) of Claim 1, after Line (3), the number of remaining working chips is also at least as large as the defective ones. However, now the total number of chips, denoted by n', is strictly smaller than n. Thus, by the induction hypothesis, the recursive call to the algorithm either returns a working chip among them, in which case the algorithm also returns it as the answer, or the recursive call returns empty, in which case the algorithm returns C as the answer – this is a correct answer in both cases, proving the first part of induction step when n is odd and C was a working chip.

  Now suppose C was a defective chip. This means that the number of working chips after removing C is definitely larger than the defective ones. By Part (1) of Claim 1, after Line (3), we will also have more working chips remained than defective ones. By (the first part of) the induction hypothesis, the recursive call returns a working chip and the algorithm gives it as the answer – this is again a correct answer, finalizing the proof of the induction step when n is odd.

- *When n is even*: When n is even, we may either have strictly more working chips (part one of the induction hypothesis), or the number of working chips can be equal to the defective ones (part two).

  Suppose first we have strictly more working chips. Similar to before, by Part (1) of Claim 1, after Line (3), we will also have more remaining working chips and thus by (the part one of) the induction hypothesis for the strictly smaller number of remaining chips, the algorithm returns a working chip, proving the first part of the induction step.

  Finally, suppose we have equal number of working and defective chips. By Part (2) of Claim 1, we will also have the number of working chips among the remaining chips after Line (3) is at least as large as the defective ones. Hence, by the induction hypothesis, the algorithm will also either returns a working chip in this case or 'empty', finalizing the proof of the induction step.

The correctness of the algorithm now follows from the first part of the induction hypothesis for all n.

Final Step: Analyzing the Efficiency of the Algorithm

We are almost done, modulo a key factor: how "efficient" is the algorithm that we designed?

In general, there are many different notions of efficiency for an algorithm. However, throughout the course, we almost always focus on the *running time* of the algorithm as its measure of efficiency, namely, our goal is to design algorithms with small running time. However, to emphasize that time is not the only measure, in this chip testing problem, we are in fact interested in the *number of tests* done by our algorithm.

An important notion about efficiency is that we typically consider *worst case* behavior of the algorithm when analyzing its efficiency: for instance, in the context of our chip testing problem, we are interested in analyzing the *maximum* number of tests done by our algorithm on *any* input. This means that some algorithms may in fact be very efficient in most scenarios and for most inputs but we still consider them "inefficient" because their worst-case behavior on at least one particular input is inefficient.

Let us analyze the number of tests in the algorithm. There are two main observations:

- Number of tests in the body of the algorithm before the recursive call, namely, in Line (2) is $\lfloor n/2 \rfloor$.
- Number of remaining chips after the first iteration is $n' \leq \lfloor n/2 \rfloor$ – this is because we keep at most one chip from each test.

Considering our algorithm itself is also recursive, it makes sense to analyze number of its tests using a *recursion* as well. Define $T(n)$ to be the *worst-case* number of tests done by the algorithm on *any* pile of n chips. Using the above observation, and since the algorithm recursively solves the problem for the remaining $n' \leq \lfloor n/2 \rfloor$ chips, we have,

$$\begin{aligned} &T(n) \leq T(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor; \\ &T(1) = 0, \qquad T(0) = 0. && \text{(we should never forget the base case)} \end{aligned}$$

Later in the course, we will learn different ways of "solving" this recursion, namely, finding a closed-form solution for its value for all n. For now, we use a fairly simple method:

$$\begin{aligned} T(n) &\leq T(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor \leq T(\lfloor \lfloor n/2 \rfloor /2 \rfloor) + \lfloor n/4 \rfloor + \lfloor n/2 \rfloor \leq T(\lfloor n/4 \rfloor) + \lfloor n/4 \rfloor + \lfloor n/2 \rfloor \\ &\leq T(\lfloor n/8 \rfloor) + \lfloor n/8 \rfloor + \lfloor n/4 \rfloor + \lfloor n/2 \rfloor \leq \cdots \\ &\leq 0 + 1 + 2 + \cdots + n/8 + n/4 + n/2 \leq n. \qquad \text{(by the formula for the sum of geometric series)} \end{aligned}$$

Final Thoughts

Congratulations! We just completed designing and analyzing our first algorithm in this course. Throughout the course, we will design and analyze many more algorithms for different problems. But the good news is that the hard part is already over – none of the remaining algorithms and problems are going to feel as "hard" as the first one we just analyzed!

Let us conclude this part with the following remark. Algorithm design is in fact often times not a *linear* process (even though the example above may suggest otherwise). We typically start with our intuition, design the algorithm, and when analyzing it (either the correctness or efficiency), we realize that something is missing – either the algorithm is wrong or inefficient or we simply cannot prove its guarantee. When this happens, we should use our knowledge at this point, go back to the first step and try to incorporate this into our intuition, and then design a new (or modified) algorithm. This process is then repeated again and again until we converge to the final algorithm.

A Simpler But Less Efficient Alternative Solution

Before we move on, let us also revisit the much simpler alternative solution to the problem that was proposed by one of your classmates and was described in the previous lecture notes. In the following, we tweak it slightly so that it works for all values of n (not only odd ones) and *sketch* the proof of its correctness and efficiency – the proofs however are not completely done and missing steps are left as exercise for the students.

1. Iterate over the chips one by one and and for each one do as follows:
   (a) Test this chip, denoted by C, against all the remaining chips.
   (b) If $\lceil (n-1)/2 \rceil$ of other chips consider C as working output C as a correct answer and terminate. Otherwise, continue to the next chip.

Proof of Correctness: Suppose n is even first and let $n = 2k$ for some integer k:

- If C is a working chip: there are still k other working chips outside C and thus C is considered working by at least k chips while $k = \lceil (n-1)/2 \rceil$, hence C will be output as a working chip correctly.
- If C is a defective chip: there are at most $k-2$ defective chip outside C and thus C is considered working by at most $k-2$ chips while $k-2 < \lceil (n-1)/2 \rceil$, hence C will not be output as a working chip and we go to the next chip.

Now suppose n is odd and let $n = 2k+1$ for some k:

- If C is a working chip: there are still k other working chips outside C and thus C is considered working by at least k chips while $k = \lceil (n-1)/2 \rceil$, hence C will be output as a working chip correctly.
- If C is a defective chip: there are at most $k-1$ defective chip outside C and thus C is considered working by at most $k-1$ chips while $k-1 < \lceil (n-1)/2 \rceil$, hence C will not be output as a working chip and we go to the next chip.

Thus, no matter whether n is even or odd, the algorithm continues checking chips until C becomes one of the working chips in which case it outputs it correctly.

Efficiency: Let us again consider the number of tests done by the algorithm. In the worst-case, it can be that all the defective chips appear first as a candidate for C, and the number of defective chips can be as large as $\lfloor n/2 \rfloor - 1$. Taking to account, we need to also examine one working chip against all other chips, it can be that the algorithm considers at most $\lfloor n/2 \rfloor$ chips as a candidate for C. For each chip also, we make $n-1$ tests, thus the total number of tests done by the algorithm is at most $\lfloor n/2 \rfloor \cdot (n-1) \leq n^2$. Note however that one can also create an example that requires the algorithm to make $\lfloor n/2 \rfloor \cdot (n-1)$ tests (try it!) which is much more than the n tests done by the original algorithm.

2 Runtime Efficiency and Asymptotic Analysis

As we stated before, in this course, we are primarily interested in the running time of the algorithms as their main measure of *efficiency*. However, computing the *exact* running time of the algorithms is quite hard (and is specific to the "computer" that runs the algorithm). Instead, we consider the following criteria for measuring the runtime of an algorithm:

1. Count the number of *basic* operations: simple arithmetic (e.g. adding or subtracting two numbers), conditional-statements (e.g., if- or while-statements), memory access (e.g., reading from or writing to an array cell), etc.
2. We count the number of operations *asymptotically*: this basically means that we *ignore* constant factors and assume the *input size* is very large (goes to infinity in limit).

   Example: How many basic operations are done in the following algorithm? Iterate over the elements of an integer array of size n: for each element, output that element multiplied by 10.

   At this stage, it is virtually impossible to *exactly* determine the number of operations done by the above algorithm: Is the algorithm maintaining a counter when iterating over the array? Do we need to also account for updates to that counter? Is outputting the element multiplied by 10 a single operation or multiple ones (e.g., computing the multiplication, storing it in a temporary variable, and then outputting it)? Is checking whether we reach the end of the for-loop yet another operation? If so, how many times we are running this operation?

   On the other hand, it is easy to see that this algorithm makes $\Theta(1)$ (namely some *constant* number of) operations for each element of the array, there are n elements in the array, and any extra book keeping, initialization etc. takes another $\Theta(1)$ operations. Thus, the runtime of the algorithm is $\Theta(n)$.

3. We analyze the *worst-case* runtime of the algorithm, i.e., the maximum time it may take on *any* (valid) input (unless specifically stated otherwise).

   Example: What is the runtime of the following algorithm? Iterate over the elements of an integer array A of size n: if for the current index i, $A[i] < A[i-1]$ terminate; otherwise continue.

   Determining the runtime of this algorithm (even asymptotically) crucially depends on the extra knowledge about the array A. For instance, if $A[1] > A[2]$, this algorithm only takes $\Theta(1)$ time. On the other hand, if all elements of the array are sorted in the increasing order, then this algorithm in fact takes $\Theta(n)$ time, and this is the maximum it may take on any input. As such, we say that the worst-case runtime of the algorithm is $\Theta(n)$.

A review of asymptotic notation: For a function $f(n)$ we use the following notations:

- Big Oh-notation ('$\leq$') – to give an *upper bound in the limit*: we write $f(n) = O(g(n))$ iff:

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C. \qquad \text{(where } C \text{ is some } \textit{constant} \text{ independent of } n\text{)}$$

  Example.

  $$100n^2 + 10n = O(n^2) \qquad \sqrt{n} \neq O(\log n) \qquad \log n = O(\sqrt{n}).$$

- Big Omega-notation ('$\geq$') – to give a *lower bound in the limit*: we write $f(n) = \Omega(g(n))$ iff:

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq C. \qquad \text{(where } C \text{ is some } \textit{constant} \text{ independent of } n\text{)}$$

  Example.

  $$100n^2 + 10n = \Omega(n^2) \qquad \sqrt{n} = \Omega(\log n) \qquad \log n \neq \Omega(\sqrt{n}).$$

- Theta-notation ('=') – to give *"asymptotic equality" in the limit*: we write $f(n) = \Theta(g(n))$ iff:

  $$f(n) = O(g(n)) \qquad \text{and} \qquad f(n) = \Omega(g(n)).$$

  Example.

  $$100n^2 + 10n = \Theta(n^2) \qquad \sqrt{n} \neq \Theta(\log n) \qquad \log n \neq \Theta(\sqrt{n}).$$

- small oh-notation ('<') – to give *strict upper bound in the limit*: we write $f(n) = o(g(n))$ iff:

  $$f(n) = O(g(n)) \qquad \text{and} \qquad f(n) \neq \Theta(g(n)).$$

  Example.

  $$100n^2 + 10n \neq o(n^2) \qquad \sqrt{n} \neq o(\log n) \qquad \log n = o(\sqrt{n}).$$

- small omega-notation ('>') – to give *strict lower bound in the limit*: we write $f(n) = \omega(g(n))$ iff:

  $$f(n) = \Omega(g(n)) \qquad \text{and} \qquad f(n) \neq \Theta(g(n)).$$

  Example.

  $$100n^2 + 10n \neq \omega(n^2) \qquad \sqrt{n} = \omega(\log n) \qquad \log n \neq \omega(\sqrt{n}).$$

To further review asymptotic notation, read Chapter 3 of the CLRS book.