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

Lecture 1

September 5, 2019

Instructor: Sepehr Assadi

1 Problems and Algorithms

- **Problem:** A *mapping* from the set of all potential inputs to the correct answer(s) for each input. For instance:
  1. Finding maximum:
     Input: any given array of numbers, Output: the largest element in the array;
  2. Sorting:
     Input: any given array of numbers, Output: a permutation of the array in non-decreasing order;

     ⋮
- **Algorithms:** A sequence of *simple* and *well-defined* steps for outputting the correct answer to *any* input of a given problem. For instance:
  1. An algorithm for finding maximum:
     (a) Pick the first element of the array as the candidate maximum.
     (b) Iterate over the elements one by one and if the current element is larger than the candidate maximum, choose this element as the new candidate maximum.
     (c) Output the candidate maximum at the end.
  2. NOT an algorithm for finding maximum:
     (a) Find the maximum of the array.
     (b) Output this number.
     (the first step is not simple enough for <u>the purpose of this problem</u>)

Our goal in this course is learning how to design *efficient* algorithms for different problems and how to analyze them rigorously. We will get to the definition of "efficiency" of the algorithms shortly. But we first need an example.

2 Chip Testing Problem

Let us consider the following problem.

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 a working 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.

First Step: Developing Intuition

The first step in designing an algorithm for a problem is to develop *intuition* about the problem – this is a necessary step that typically is done *implicitly* and is quite crucial for the success of the algorithm design; at the same time, it is also *never sufficient* for obtaining the final answer; In other words, intuitions are different from final solutions. Let us first develop some intuition about this problem.

- It <u>seems that</u> our job would be "hardest" when all defective chips "cooperate": this means that (i) whenever we connect two defective chips together, they always output (working, working) (which confuses us with the case when we connect two working chips together), and (ii) whenever we connect a defective chip to a working one, the answer is always (defective, defective) (which confuses us with which one of the two chips is indeed defective).
- We can think of the chips as forming two different teams, the working team (good guys) and defective team (bad guys).
- Each team is trying its best to make us pick one of their team members as the solution.

We *emphasize* that we did not prove anything yet nor designed any algorithm. We simply "made an educated guess" that the above setting should be most challenging and are trying to solve this challenge. Now let us develop our intuition further:

- The above analogy suggests that each working chip has the "same power" as each defective chip: if we connect them together, they both will declare each other defective and if we connect them to their "teammates", they both will be considered working.
- The crucial observation is that the number of working chips is strictly more than the defective chips: this basically means that the "power" of good guys should be strictly more than bad guys and so they should win.
- A bit more concretely, suppose we pair the chips arbitrarily with each other; the following can happen for each pair (note that we are only considering the scenario above – this is *not* a general solution):

  1. Both are working; we expect the answer to be (working, working).
  2. Both are defective; we expect the answer to be (working, working).
  3. One is working and the other is defective; we expect the answer to be (defective, defective).

  The key observation is that number of first type of pairs has to be more than second type of pairs (this requires a proof). So if we entirely ignore the third types of pairs, we still have a pile of chips containing more working pairs than defective ones. We should now think of solving the problem *recursively.*

Examples. In the process of developing intuition, it sometimes also helps to consider simple examples of the problem. For instance, how do we solve the problem where there are exactly three chips and only one of them is defective?

A simple solution is as follows:

- Test any two of the chips against each other.
- If the answer is (working, working) output either of them – we are correct because the only other way for outputting a (working, working) answer is if *both* chips are defective: but we know that there is only one defective chip!
- If the answer is anything else, output the third chip – any answer other than (working, working) means that at least one of the chips is defective: since there is only one defective chip, we know that the third chip should be working.

You are encouraged to also consider the example when there are 5 chips and only 2 of them are defective. How do you solve the problem then?

Second Step: Designing an Algorithm

At this stage, we developed a rough intuition about the problem – it is now the time to use this intuition to design an algorithm. You are encouraged to consider this algorithm in the context of the above intuition and see how we used each part of the intuition in designing the algorithm. However, we emphasize that we *no longer assume* the above scenario (of defective chips behaving as a team) and consider the most general case of the problem. Our algorithm is recursive and is defined as follows:

1. **Base case:** If $n = 1$ return the single chip; if $n = 0$ return 'empty';[1] 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 (again ignoring the potentially one chip we set aside).
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'.

The above is a complete algorithm for this problem – we clearly specified the steps needed and each step is simple enough for the purpose of this problem[2]. We are *not* done however and the main task is actually ahead of us: we need to *analyze* the algorithm and *rigorously prove* that it indeed solve the problem for us. This part is left for the next lecture.

An Alternative Solution for Even n

The following solution for the case when n is *even* was given by one of your classmates. This is truly a brilliant algorithm and is much simpler than the algorithm above – the main catch is that it is not as "efficient" as the algorithm described above; however, as we have not even defined what do we mean by efficiency so far, this issue can be ignored for now. The algorithm is as follows:

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 a *strict majority* of other chips consider C as working (remember n is even and so there are odd number of remaining chips and hence there is always a strict majority), output this chip as the correct answer. Otherwise, continue to the next chip.

An analysis of this algorithm and its extension to odd choices of n is given in the next lecture.

[1]It may sound silly to consider a case where $n = 0$, namely, when there are no chips at all! However, as our algorithm is recursive, this case is actually needed to make our analysis work (although technically we can circumvent this by writing a slightly more complicated algorithm).

[2]This algorithm is slightly more complicated than the one described in the class; if you remember, the algorithm for the class required us to assume n is even (in fact throughout *all* recursive calls) and it was stated that some more care is needed for general case.