CS3230 Design & Analysis of Algorithms **January 13, 2022**

Lecture 1: Introduction

Lecturer: Warut Suksompong

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*[1]

Welcome to CS3230!!

1.1 Module Logistics

Please see accompanying slides.

1.2 What is an Algorithm?

Before talking about algorithms, let's first be clear about what a *computational problem* is. In general, a computational problem is defined by a (possibly infinite) set of pairs (x, S) where x describes a valid input[2] and S is the set of valid *solutions* for the input x.

Example 1. *Let* Mul *be the problem of multiplying two integers.* $\mathsf{Mul} = \{((x, y), \{xy\}) : x, y \textit{ integers}\}$.

Example 2. *Let* Factor *be the problem of finding some non-trivial factor of an input positive integer. Then,* $\mathsf{Factor} = \{(x, \{y : y \textit{ divides } x, y \neq 1, y \neq x\}) : x \textit{ positive integer}\}$. *Note that if x is prime, then the set of solutions S is empty.*

We'll see many, many computational problems throughout this module, where the inputs may be numbers, arrays, matrices, graphs, functions, circuits, DNA sequences, databases, etc. The key point to understand here is that the notion of a problem is purely mathematical. It gives no hint as to *how* to find a solution for a given input.

An algorithm provides the how. For a computational problem, it is a well-defined fully-specified procedure that outputs a valid solution, given a valid input. We won't really formalize what "well-defined" means, but the idea is that the procedure should be implementable by a reasonable computing device. Throughout this module, we'll be using pseudocode to describe procedures that one should be able to run on a conventional computer (using, e.g., C or Python). In other contexts, one can also consider more powerful devices (e.g., quantum computers) or more restricted devices (e.g., ruler and compass for constructing angles).

Example 3. *Here is an algorithm* BruteForce *for the* Factor *problem above.*

```
BruteForce($x$)
1  for $y = 2$ to $x - 1$
2      if $y$ divides $x$
3          return $y$
4  return None
```

BruteForce *simply checks if any integer between 1 and x divides x, and returns the first one it finds.*

The distinction between problems and algorithms is fundamental. There can be many (in fact, infinitely many) algorithms for any particular problem. Even for the problem Mul, the primary school algorithm for multiplication, is not the only one, and in fact, there are superior alternatives when the input is very large!

[1]Lecture notes in this module based on Arnab Bhattacharyya's lecture notes.

[2]Sometimes, the input to a computational problem is also called an *instance*.

Remark. We will sometimes discuss settings which go beyond the traditional input-output model. In the traditional model, the algorithm has access to the whole input x, and so it can use all of x to come up with a valid solution. In other settings, it may not be reasonable to assume that x is available to the algorithm. For example, x may be a huge graph (e.g., a social network), so all of it may not even be readable by a single algorithm. Or x may be partially hidden due to noise, missing data, or for privacy reasons. Or in an online setting, x may be revealed step-by-step.

1.3 What do we want out of an algorithm?

In this module, we will focus on two key aspects of algorithms: their **correctness** and **running time**.

Correctness. Given a computational problem Π, an algorithm for Π is said to be *correct* if for every valid input, it halts with a valid solution. This is a worst-case notion of correctness, in the sense that even the worst input cannot make the algorithm give an incorrect solution. In some cases, worst-case correctness is too much to ask for, and one can consider more relaxed notions, e.g., that the algorithm is correct on a "random" input or that the algorithm is correct on every input "with high probability" or that the algorithm's output is "approximately correct." We will return to these notions later, but unless stated otherwise, we will by default focus on worst-case correctness.

Running time. The point of running time analysis is to determine how the number of steps executed by an algorithm scales with its *input size*. What the input size is depends on the particular problem. For example, if the input is an array, then its size is typically taken to be the number of array elements. If the input is an integer, then its size is typically taken to be the number of bits to write the input in binary. If the input is a graph, then its input size is measured by the number of vertices and number of edges in the graph. It is necessary to specify what the notion of input size is whenever we talk about running time.

For a particular input x, the *running time* of an algorithm on x is the number of "steps" executed by it according to a *computational model*. The computational model explicitly defines what types of computations can be performed in one step. In the next lecture, we will define the *random-access machine (RAM) model* which we will use for the rest of the course. In this lecture, for pedagogical reasons, we will work with simpler computational models in which the operations allowed in any step are very restricted.

The *worst-case running time* of an algorithm is the longest running time for any input of size n.

Example 4. *Let us measure the runnning time of* BruteForce *by the maximum number of times line 2 in* BruteForce *is invoked. Let $T(n)$ denote the worst-case running time for any input x of size n, where size is measured by the number of bits needed to write x.*

If x is[3] *a prime of input size n, then* BruteForce *will iterate over all the numbers $2, \ldots, x-1$. Hence, the algorithm's runtime for such an input is at least $2^{n-1}-1$. For any x of size n, clearly, the runtime is at most $2^n - 3$. So, $2^{n-1} - 1 \le T(n) \le 2^n - 3$.*

From the next lecture, we will start using asymptotic notation for running time. This will make qualitative comparison between different algorithms much easier.

Often, we will prove *upper* and *lower* bounds for the worst case running time $T(n)$, i.e., something like $\ell(n) \le T(n) \le g(n)$ (as in the example above). Let's interpret this carefully:

- $T(n) \le g(n)$ means that for every input of size n, the running time is at most $g(n)$.
- $T(n) \ge \ell(n)$ means that there exists an input of size n for which the running time is at least $\ell(n)$.

An upper bound on $T(n)$ is a very strong guarantee because it's a guarantee for any input.

As we alluded to for correctness, one can also consider weaker notions such as *average-case running time* or *expected running time*. We will return to these ideas later in the course.

[3]Such an x exists for every n by the Bertrand-Chebyshev theorem.

Other desiderata. Besides correctness and running time, there may be other considerations which are also important in algorithm design. Here are a few.

- **Simplicity.** An algorithm whose code is too complex may be hard to implement and maintain, even if it enjoys correctness and running time guarantees. Note that a simple algorithm could have high running time. For example, consider the following algorithm Collatz:

  ```
  Collatz($t$)
  1 while $t > 1$
  2     if $t \equiv 0 \pmod 2$
  3         $t = t/2$
  4     else
  5         $t = 3t + 1$
  6 return
  ```

  The algorithm's (pseudo-)code is simple enough, but whether its runtime is finite or not for every positive integer t is the notorious Collatz Conjecture.

- **Space Usage**. One can also consider how much memory is used by the algorithm. This might depend on the cache and memory resources of the computer the algorithm is run on.

- **Energy Consumption**. The amount of power needed to run the algorithm. This is different from runtime analysis as modern processors allow dynamic frequency scaling so as to minimize power usage when not needed.

- **Parallelism**. Modern multicore processors allow for extensive multithreading. In such situations, one would prefer algorithms that can be parallelized versus those that are inherently sequential.

- **Fairness and Ethics.** Increasingly, the outputs of algorithms are used to make decisions that have profound consequences for human lives. For example, automated decisions to extend loans or not, or automated decisions on fines or prison sentences. In these situations, one must also judge algorithms by whether they adhere to legal regulations, ethical norms, and social equity standards.

1.4 Concrete Models of Computation

To begin our study of algorithms, let us consider simplified computational models in which the running time is measured by the number of times a particular fixed operation is performed. The reason we study these models is that here, we can exactly pin down the running time of the optimal algorithm for many interesting problems. Such detailed understanding seems far beyond our reach for more realistic models of computation (like the RAM model we introduce in the next lecture).

1.4.1 Comparison Model

Definition 1.4.1. *In the* comparison model*, the input is an array A of n numbers, and an algorithm can compare two elements ("Is $x > y$, $x < y$, or $x = y$?") in one time unit. No other*[4] *operations on the elements are allowed. The array can be manipulated (e.g., permuted or broken into two sub-arrays) without operating on the elements at no cost.*

The running time in the comparison model is simply the total number of pairwise comparisons made. Note that here, the array can be re-ordered at no cost; so it may not be particularly realistic for some algorithms.

[4] For example, we cannot use an array element as an index to another array (e.g., a hash table), and we cannot perform arithmetic on the array elements.

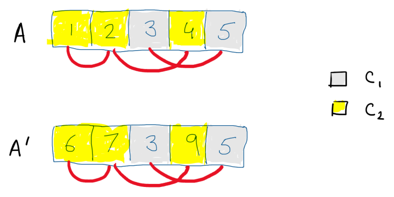

Figure 1.1: Finding the maximum in an array of size 5

1.4.1.1 Maximum

Given an array A of n distinct elements (denoted $A_1, \ldots, A_n$), let Max denote the computational problem of finding the largest element in A.

Claim 1.4.2. *There is an algorithm for* Max *with running time at most $n-1$ in the comparison model.*

Proof. Obviously, the following works:

```
RunThru($A$)
1  $cur = 1$
2  $n = A.length$
3  for $i = 2$ to $n$
4      if $A_i > A_{cur}$
5          $cur = i$
6  return $A_{cur}$
```

□

Note that RunThru makes exactly $n-1$ comparisons for any input; so, the upper bound in Claim 1.4.2 is tight. In fact, we can argue that $n-1$ comparisons are necessary not just for RunThru but for any algorithm!

To build intuition, let's first consider $n = 2$. If there's an algorithm $\mathcal{M}$ that makes less than $n - 1 = 1$ comparison, then $\mathcal{M}$'s output is independent of its input. This is clearly impossible. Next, let's look at $n = 3$ and an algorithm $\mathcal{M}$ that makes less than $n - 1 = 2$ comparisons. For instance, if the input is $A = [1, 2, 3]$, suppose $\mathcal{M}$ only compares A_1 and A_3. But now, if $\mathcal{M}$ is run on $A' = [1, 4, 3]$, it will also only compare A'_1 and A'_3. But because $A'_1 = A_1$ and $A'_3 = A_3$, the algorithm cannot distinguish between A and A'. Since the maximum for A and A' occur at different indices, the algorithm must be wrong on at least one of these inputs.

Claim 1.4.3. *Any algorithm solving* Max *must have running time $\geq n-1$ in the comparison model.*

Proof. Fix any algorithm $\mathcal{M}$ that correctly solves Max on all inputs. The idea of the proof is to show that if $\mathcal{M}$ always makes less than $n-1$ comparisons, then there are two arrays A and A' so that the two cannot be differentiated based on $\mathcal{M}$'s comparisons. However, the maximum elements of A and A' occur at different locations, and so, $\mathcal{M}$ must err on either A or A', a contradiction.

Take an input A for which $\mathcal{M}$ makes $< n-1$ comparisons. Construct a graph G on n nodes (indexed by $1, \ldots, n$), where nodes i and j are adjacent iff $\mathcal{M}$ compares A_i and A_j. Since G has $< n-1$ edges, it is disconnected. That is, there exists a partition of the nodes into C_1 and C_2 such that for any $i \in C_1, j \in C_2$, there is no edge between i and j, and so, A_i and A_j are not compared by $\mathcal{M}$.

Suppose given A as input, $\mathcal{M}$ outputs A_{i^*} as the maximum. Without loss of generality, let $i^* \in C_1$. Now, consider a new array A', such that if $i \in C_1$, $A'_i = A_i$, but if $i \in C_2$, $A'_i = A_i + m$ where m is sufficiently large that the maximum element of A' is in C_2. See an example in Figure 1.1.

Observe that the comparisons made by $\mathcal{M}$ cannot distinguish between A and A', as the only comparisons which are different are between $i \in C_1$ and $j \in C_2$, and $\mathcal{M}$ does not make such comparisons. So, $\mathcal{M}$ must give a wrong answer for either A or A'.[5] □

This proof is an example of an **adversary argument**. You can think of the input being decided on-the-fly by an adversary who keeps her options open about what the actual input is. She makes sure that if the algorithm makes too few comparisons, then there are at least two different inputs which are consistent with the results of these comparisons, and yet, the solutions for the two inputs are different.

1.4.1.2 Second Largest

Let SecondLargest denote (as you'd expect from the name) the problem of finding the second largest element in an array of n distinct numbers. One quick observation is that the proof of Claim 1.4.3 can also be adapted to SecondLargest; so, it follows that any algorithm must make at least $n-1$ pairwise comparisons.

What about an upper bound? The following algorithm clearly makes $(n-1)+(n-2) = 2n-3$ comparisons.

```
DOUBLERUN($A$)
$n = A.length$
$max1 = 0$
for $i = 1$ to $n$
    if $max1 = 0$ or $A_i > A_{max1}$
        $max1 = i$
$max2 = 0$
for $i = 1$ to $n$
    if $i \neq max1$ and ($max2 = 0$ or $A_i > A_{max2}$)
        $max2 = i$
return $A_{max2}$
```

Can we do better? Yes, as was observed by Charles Lutwidge Dodgson (better known as Lewis Caroll, author of *Alice in Wonderland*). Consider arranging the items in a tennis tournament structure. So, in the first round, elements $2i-1$ and $2i$ are compared for all i. The winners of this round are then again matched and compared in the second round. If after some round, the number of winners is odd, then in the next round, one of the winners is not compared while the rest are paired and compared. When we reach a round with one element, that element is declared the winner.

Claim 1.4.4. *The winner of the tournament played on elements of an array A is the maximum of A. The total number of comparisons to determine the winner is $n-1$, where n is the length of A.*

Proof. We can prove this by *induction*. If $n = 2$, both parts of the claim are obvious. For $n > 2$, let x be the winner, and suppose in the last round, x is compared with y. Now, x and y are winners of smaller tournaments on n_1 and n_2 elements respectively, where $n_1 + n_2 = n$. By induction, x is the maximum of the n_1-length subarray, and y is the maximum of the n_2-length subarray. Clearly, then if $x = \max(x, y)$, x is the maximum of the entire array. Also by induction, the number of comparisons until the last round is $(n_1 - 1) + (n_2 - 1)$ to find x and y; so the total number of comparisons is $1 + (n_1 - 1) + (n_2 - 1) = n - 1$. □

Claim 1.4.5. *The winner must have been compared against the second-largest element in some round of the tournament.*

Proof. The second-largest element is the winner against any element that is not the maximum. So, it must get compared against the maximum in some round, since it is not the overall winner. □

[5] Note that the proof actually shows that any correct algorithm $\mathcal{M}$ must make $\geq n-1$ pairwise comparisons for *every* input A, whereas to show a lower bound on the running time, we only needed that there *exists* an input requiring $\geq n-1$ comparisons. So, we are in fact proving something stronger above.

Claim 1.4.6. *The winner is directly compared to at most* $\lceil \lg n \rceil$ *elements.*[6]

Proof. If the number of elements at some round is $i > 1$, then the number of elements entering the next round is $\lceil i/2 \rceil$. Then, the claim is easy to check. □

Now, we are ready to describe the algorithm for SecondLargest. Given an input array A, first run the tournament algorithm to determine the winner x. Moreover, maintain a set S that records all the elements x was compared against in the tournament. Output the maximum of S using RunThru.

The correctness of the algorithm follows from Claims 1.4.4 and 1.4.5, which together show that the maximum of S must be the second-largest element. The running time of this algorithm in the comparison model is $n - 1$ (cost for determining x from Claim 1.4.4) plus $\lceil \lg n \rceil - 1$ (cost for determining the maximum of S, where size of S is bounded using Claim 1.4.6), which is $n + \lceil \lg n \rceil - 2$. This is in fact known to be optimal!

1.4.1.3 Sorting

Given an input array A of n distinct numbers, the Sort problem is to output a re-ordering B of the array A, such that $B_1 < B_2 < \cdots < B_n$. For simplicity, in this section, assume n is a power of two.

Claim 1.4.7. *There is an algorithm for* Sort *whose running time is* $< n \lg n$ *in the comparison model.*

Proof. Consider the classic MergeSort algorithm. We will discuss this in detail in Lecture 3 (see also Chapter 2 of CLRS), but informally, it divides A into two equal halves, recursively sorts the two halves, and then merges them into one sorted array. Given two sorted arrays of length $n/2$ each, the number of comparisons needed to merge them is $\leq n - 1$. Hence, the total number of comparisons is:

$$n - 1 + 2\left(\frac{n}{2} - 1\right) + 4\left(\frac{n}{4} - 1\right) + \cdots + \frac{n}{2}(2 - 1) = n \lg n - n + 1.$$

□

We now argue a lower bound on the running time of any algorithm for Sort.

Claim 1.4.8. *Any algorithm solving* Sort *must have running time at least* $\lg(n!)$ *in the comparison model.*

Proof. We will again give an adversary argument, where the adversary does not fix an input in advance but instead ensures that if the algorithms makes too few comparisons, there are at least two inputs with two different solutions that are both consistent with the results of the comparisons made by the algorithm.

In the beginning, there are $n!$ permutations of the set $\{1, \ldots, n\}$ that the adversary could possibly choose as an input. Each of these permutations needs to be re-ordered differently to get sorted. Call this set $\mathcal{U}$. Every time the algorithm issues a comparison query (of the form "Is $A_i > A_j$?"), the adversary checks whether $\mathcal{U}_{\text{yes}} = \{A \in \mathcal{U} : A_i > A_j\}$ is of size at least $|\mathcal{U}|/2$. If so, it replies YES to the algorithm and sets $\mathcal{U}$ to be $\mathcal{U}_{\text{yes}}$. Otherwise, it replies NO and sets $\mathcal{U}$ to be $\mathcal{U} \setminus \mathcal{U}_{\text{yes}}$.

Clearly, all the arrays in $\mathcal{U}$ are consistent with the comparisons made by the algorithm. Now, if the algorithm makes $< \lg(n!)$ many queries, then the adversary makes sure that $\mathcal{U}$ contains at least two distinct arrays, since the size of $\mathcal{U}$ decreases by at most $1/2$ for every comparison. But then, the algorithm will re-order both of them in the same way, and so must err on one of them. □

The proof is short but idea is extremely powerful and general. Make sure you understand every step. Although we phrased the proof as an adversary argument, there is a natural way to view it "information-theoretically." The idea is that each comparison yields one bit of information ($>$ or $<$), while the algorithm needs to choose between $n!$ different ways to re-order the input. Since you need $\lg(m)$ many bits to specify an arbitrary element of a set of size m, at least $\lg(n!)$ comparisons are needed by the sorting algorithm.

It's known that $n! \geq (n/e)^n$. So, the above lower bound is approximately $n \lg n - 1.44n$. This compares to the upper bound of $n \lg n - n + 1$ from Claim 1.4.7. Work continues to this day to find tighter bounds.

[6] Throughout the course, to be consistent with the textbook, we will use lg to denote $\log_2$, i.e., logarithm with base 2.

1.4.2 Query Models

Definition 1.4.9. *In the* string query model*, the input is a string of n bits. In one time unit, an algorithm can query one bit of the string. All other operations on the string can be performed at no cost.*

The total number of bits of the input string queried by an algorithm is called its *query complexity*. Let us begin with an easy example.

Claim 1.4.10. *Let* Zero *be the problem of deciding whether an input string x is the all-zero string or not. The optimal algorithm for* Zero *has query complexity n.*

Proof. Clearly, the algorithm which just checks whether each bit is zero has query complexity n. To show that this is optimal, we use a simple adversary argument.

Suppose $\mathcal{M}$ is an algorithm for Zero making $< n$ queries. Consider an adversary that replies to each of $\mathcal{M}$'s query with zero. After $\mathcal{M}$ halts, there is at least one coordinate i^* that it hasn't queried. Thus, $\mathcal{M}$ cannot distinguish between z and z^*, where z is the all-0 string and z^* equals z in all coordinates except i^*. □

A problem requiring n query complexity, like checking whether all bits are zero, is called **evasive**. The notion of evasiveness also extends to *graph* properties.

Definition 1.4.11. *In the* graph query model*, the input is the symmetric adjacency matrix G of an n-node undirected graph. In one time unit, an algorithm can query one entry in the matrix G. All other operations on G are free.*

In the context of the graph query model, a problem is evasive if it requires $\binom{n}{2}$ queries.

Claim 1.4.12. *The problem of determining whether a graph is connected is evasive.*

Proof. We again use an adversary argument. Let $\mathcal{M}$ be an algorithm making $m < \binom{n}{2}$ queries which decides whether its input is connected or not. It issues a sequence of queries $(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$, and the adversary replies with $b_1, b_2, \ldots, b_m$.

We now define how the adversary replies to each query. Suppose the adversary receives the ℓ'th query for $1 \leq \ell \leq m$. The adversary's strategy for choosing b_ℓ is as follows. She considers the graph $G^{(\ell)}$ defined by: $G^{(\ell)}[i_1, j_1] = b_1, \ldots, G^{(\ell)}[i_{\ell-1}, j_{\ell-1}] = b_{\ell-1}$, $G^{(\ell)}[i_\ell, j_\ell] = 0$ and all other entries equal to 1. That is, $G^{(\ell)}$ is the graph which is consistent with the previous query replies, does not have the edge (i_ℓ, j_ℓ) and has all other unqueried edges. If $G^{(\ell)}$ is connected, she replies $b_\ell = 0$; otherwise, she replies $b_\ell = 1$.

Now, after $\mathcal{M}$ makes all its m queries, we can define two graphs. The first graph, which we call G_0, is consistent with the adversary's replies but sets all the unqueried edges to 0. The second graph, which we call G_1, is also consistent with the adversary's replies but sets all the unqueried edges to 1. $\mathcal{M}$ cannot distinguish between G_0 and G_1 because both are consistent with its queries. We argue below that G_0 is disconnected while G_1 is connected, and hence, $\mathcal{M}$ must err on one of them, finishing the proof.

Clearly, G_1 is connected by the definition of the adversary's strategy.

Lemma 1.4.13. *G_0 is disconnected.*

Proof. Consider a pair of nodes i and j such that (i, j) was not queried by $\mathcal{M}$. We claim that there is no path between i and j in G_0.

Let's argue by contradiction. Suppose there is a path between i and j in G_0. This path only consists of queried edges to which the adversary replied 1. Let (i', j') be the edge on this path that was queried last by $\mathcal{M}$. Now, observe that according to the adversary's strategy given above, there is a path from i' to j' in the graph G_{con} she considers at this stage. This is because in G_{con}, the path from i' to i, the path from j to j', and the edge from i to j are all present. Thus, she must reply 0 to the query (i', j'), a contradiction! □

□

1.5 Takeaways

(1) Computational problems specify what we want to solve. Algorithms specify how we solve them.

(2) Worst-case correctness and running time guarantees hold for all valid inputs to the algorithm.

(3) Adversary arguments are used to prove lower bounds for the running time in concrete computational models. The adversary makes sure that if an algorithm takes too few steps, then it can come up with two inputs with different solutions that cannot be differentiated by the algorithm.