Lecture 1: Introduction and course overview

What is the course about?

The goal of the course is to introduce some of the basic principles behind designing and analyzing algorithms. We will cover some of the standard paradigms, such as divide and conquer, dynamic programming, greedy optimization, etc. We will then discuss the power of randomness in algorithm design. It turns out that in certain algorithms, randomly making choices can avoid "pathologies", thus resulting in *expected* running times that are much better than *worst case* running times. We will also see how to formulate computational problems in the language of optimization (e.g., linear programs or other convex formulations). This turns out to be another powerful paradigm in designing algorithms. Finally, we will discuss the question: are there fundamental limits to algorithm design? I.e., could it be that for certain problems, there are NO efficient algorithms that work for all inputs? Studying the *complexity* of various problems is one of the major research directions in computer science.

The computational model

When we talk about algorithms, we often think of all the input being stored in memory (in appropriate data structures), and we design algorithms that have random access to the memory -- this is known as the **RAM model**. This is realistic in most ``simple'' settings, but in more and more modern systems, data can be too large to fit in memory. Here, other considerations come into algorithm design (e.g., data movement, access patterns, etc.). However, in this course, we will stick to the RAM model. It turns out that the general *analysis ideas* will apply to other models.

Background

As we proceed with the course, we will assume that students are familiar with basic proof techniques (induction, proof by contradiction, etc.). Students are also assumed to be comfortable with basic discrete mathematics and data structures. For instance, some concrete topics include:

- Basic data structures: lists, arrays, binary search trees. (All three data structures are ways to store *sets of integers*, but each one offers different ways to manipulate the set. For example, inserting a new element can take more time in a binary search tree than in a list, but searching for an element is significantly faster. Such trade-offs are common in data structures.)
- Analysis of algorithms: big-oh, little-oh, Θ, notation, asymptotic analysis. (*Aside:* aren't constants important? why does asymptotic analysis ignore them? In practice, constants certainly matter -- e.g., an algorithm with a running time of $10^{100}n$ might be, for small enough n (<1000, say), much less practical than one with a running time of $(1.01)^n$. However, asymptotic bounds give us a good sense of how the running time scales with n (e.g., if we double n, what happens to the run time?). Further, in practice, constants can often be improved via a more careful implementation.

- Graphs: storing graphs (adjacency lists, adjacency matrix), breadth-first and depth-first search for answering *reachability* questions on graphs.
- Pseudo-code: throughout the course, we will use pseudocode to describe and write down algorithms. This allows us the flexibility to reason about the algorithm without being bogged down by the artefacts of the implementation (example below).

Analyzing running times -- basic example

Consider the following procedure Sort(A) (this is called BubbleSort). !! *Note:* by sorting, we mean sorting in increasing order.

```
Input: array A[0, ..., N-1], all distinct elements

Procedure Sort(A):
  while array A is not sorted, do:
    for i in 0, ..., N-2:
      if (A[i] > A[i+1]) swap(A[i], A[i+1])
```

We would like to analyze the *worst case* running time of this procedure. I.e., we wish to show that for *any* input array A[], the procedure takes time that is at most some function of N, the size of the array.

Analyzing any algorithm. There's only one way to analyze the running time of an algorithm, and that is by going line by line and understanding the complexity of different loops. In this case, we have a while loop. Checking the condition of the while loop takes $O(N)$ time. Each iteration of the while loop then takes $O(N)$ time as well (because the algorithm simply iterates over the array, potentially swapping neighboring entries). Thus the running time is $O(N)\times$(the number of iterations of the while loop). Thus we only need to bother about the latter quantity.

A note on the pseudocode: notice that we've not bothered to write down how we check the condition of the while loop. Likewise, we haven't written down the code for swapping. As long as we are not doing anything non-standard, this will be the norm in the course.

Some thoughts:

- There are simple examples that show that in the worst case, the while loop can get executed $O(N)$ times. For example, imagine that the array has numbers $1, 2, \ldots, N$, and that 1 appears at the very end (i.e., A[N-1] $= 1$). In every iteration, the 1 moves at most one step to the left.
- This shows a *lower bound* on the running time, while what we need is a statement that is more general -- we need to show that the algorithm always terminates after a certain number of iterations of the while loop.
- Thus, in a sense, we need to argue that the algorithm is making some progress.

The key insight is the following observation:

Observation 1. After the first iteration of the while loop, the largest element in the array moves to its *correct* position (in sorted order, which is A[N-1]).

The proof is immediate, because wherever the largest element is to start with, it will always get swapped with the element on its right.

Motivated by this, we have a more general observation:

Observation 2. For any $1 \leq t \leq N$, After the first t iterations of the while loop, the t largest elements of the array are all in their correct positions.

When $t = 1$, this is precisely the same as the first observation. This suggests a proof by induction.

Proof. The base case is $t = 1$ (observation 1). Thus, let us suppose that the statement is true for some t. I.e., after the first t iterations, the t largest elements are all in their correct positions in the sorted order.

Now consider the $(t+1)$th largest element. Suppose that after the tth iteration, it is in the position A[j]. In the $(t+1)$th iteration of the while loop, this element moves over to $A[N-t-1]$, but not any further (because it is larger than all the elements A[j+1], A[j+2], ..., but smaller than $A[N-t]$). Further, no swaps occur after $A[N-t]$, because by the inductive hypothesis, these elements are all already in their correct positions. This shows that the statement holds for $(t+1)$ as well, thus completing the proof by induction. QED.

Once we have observation 2, note that we immediately have that the overall running time is $O(N^2)$ -- this is because setting $t = N$, we have that the array is fully sorted after N iterations of the while loop, and since each iteration takes $O(N)$ time, the bound follows.

Comment: note that the proof itself is relatively straightforward -- the key was to come up with the right observations/inductive statements. This is typically the case with analysis of algorithms. Often, trying out small examples, and understanding ``what the algorithm is doing'' is key to coming up with the analysis.

A second example

Let us consider a second example:

Problem. given a sorted array A, with n integers, i.e., A[0] < A[1] < ... < A[n-1], and another integer x, find if x is in the array or not.

This, of course, can be done via the classic *binary search* procedure. The idea is to first compare x with A[n/2]. If $x > A[n/2]$, then we know that x, if it is present at all in the array, has to be in the sub-array $A[\frac{n}{2}+1], A[\frac{n}{2}+2], \ldots, A[n-1]$. On the other hand if $x < A[n/2]$, then x, if it is present in the array, must be in the sub-array $A[0], \ldots, A[\frac{n}{2}-1]$. (And of course, if $x = A[n/2]$, we have found x!)

Let us write the procedure more formally. Since we are searching for x in different sub-arrays of A[], it makes sense to define a procedure that takes as input the start and end index of the sub-array.

```
Procedure BinarySearch(x, start_idx, end_idx)
  if end_idx - start_idx <= 1, check if x is one of the end points and return YES/NO
  set midpoint = (start_idx + end_idx)/2    //formally, we need the floor of this qty
  if x > A[midpoint], return BinarySearch(midpoint, end_idx)
  if x < A[midpoint], return BinarySearch(start_idx, midpoint)
  if x == A[midpoint], return YES (i.e., x is found in the array)
```

We will (a) prove that this procedure always returns the right answer (i.e., YES if x is in the array and NO otherwise), and (b) analyze the running time.

Correctness. Note that the procedure itself is a recursive one. Thus it is natural to try proving the correctness by induction. Let us define L to be the length of the sub-array we are considering, i.e., $L =$ end_idx - start_idx $+ 1$. We will show the correctness via induction on L.

First off, if $L \leq 2$, then clearly the algorithm returns the right answer (as it simply checks both the values). Thus the base case is obvious.

Next, suppose the algorithm returns the right answer for all $L \leq t$, for some $t \geq 2$. Now consider a sub-array of size $t + 1$. In this case, it is easy to see that midpoint does not equal either the start_idx or end_idx. Thus both the recursive calls will be to strictly smaller sub-arrays (and thus the inductive hypothesis holds). Further, as we argued earlier, since A[] is sorted, x is in A[start_idx, end_idx] if and only if x is in the appropriate sub-array.

Thus if the algorithm returns the right answer for all $L \leq t$ for some $t \geq 2$, it also returns the right answer for $t + 1$. This completes the proof by induction.

Running time

Next, consider the running time of the procedure. We start with start_idx = 0 and end_idx = N-1, and thus the array length is N. In the next recursive call, the length of the array is $\lfloor N/2 \rfloor \leq N/2$. Likewise, in the next call, it is $\leq N/4$, and so on. Specifically in the tth recursive call, the length of the sub-array is $\leq N/2^t$. If this number is ≤ 2, then we have the base-case. In each of the recursive calls, the algorithm just performs one comparison (except in the base case, where it performs two). Thus the overall running time is at most $t + 2 = O(t)$, where t is an integer such that $N/2^t \leq 2$. This simplifies to $t + 1 = \log_2 N$. Thus the overall running time is $O(\log N)$.

Trinity of algorithm, correctness, complexity analysis

Throughout the course, this will be the way we present algorithms (pseudocode, argument about correctness, and analysis of the complexity). As you get used to it, we will do things less formally, but you should be able to write out the details as needed -- to convince yourself or others.