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

Lecture 3

September 12, 2019

Instructor: Sepehr Assadi

1 Proof by Induction

Let us consider some simple examples of a proof by induction. We prove the following two for all integers n:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}; \tag{1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}. \tag{2}$$

Proof of Eq (1)**.**

- *Induction base:* For $n = 1$, $\sum_{i=1}^{1} i = 1$ and $\frac{1 \cdot (1+1)}{2} = 1$ so the hypothesis is true.
- *Induction step:* Suppose the hypothesis is true for all $n \leq k$ and we prove it for $n = k + 1$. We have,

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= \sum_{i=1}^{k} i + (k+1) && \text{(we simply took out the last term in the sum to add it explicitly)} \\
&= \frac{k \cdot (k+1)}{2} + (k+1) && \text{(since the induction hypothesis is true for all } n \leq k\text{)} \\
&= (k+1) \cdot (\frac{k}{2} + 1) && \text{(basic arithmetic: by factoring out } (k+1) \text{ from the two terms above)} \\
&= \frac{(k+1) \cdot (k+2)}{2} && \text{(basic arithmetic: by writing the second term differently)} \\
&= \frac{n \cdot (n+1)}{2}. && \text{(since } n = k+1\text{)}
\end{aligned}$$

  So we proved the induction step, which finalizes the proof. In other words, we prove that for *all* n:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}.$$

  Note that we do *not* need to do anything else – the rest of the proof is taken care of by the "magic" of the induction itself!

Proof of Eq (2)**.**

- *Induction base:* For $n = 1$, $\sum_{i=1}^{1} i^2 = 1$ and $\frac{1 \cdot (1+1) \cdot (2 \cdot 1+1)}{6} = 1$ so the hypothesis is true.

- *Induction step:* Suppose the hypothesis is true for all $n \leq k$ and we prove it for $n = k + 1$. We have,

$$\begin{aligned}
\sum_{i=1}^{k+1} i^2 &= \sum_{i=1}^{k} i^2 + (k+1)^2 && \text{(we simply took out the last term in the sum to add it explicitly)} \\
&= \frac{k \cdot (k+1) \cdot (2k+1)}{6} + (k+1)^2 && \text{(since the induction hypothesis is true for all } n \leq k\text{)} \\
&= (k+1) \cdot (\frac{k \cdot (2k+1)}{6} + (k+1)) \\
& && \text{(basic arithmetic: by factoring out } (k+1) \text{ from the two terms above)} \\
&= (k+1) \cdot (\frac{2k^2 + 7k + 6}{6}) && \text{(basic arithmetic: by simplifying the terms)} \\
&= (k+1) \cdot (\frac{(k+2) \cdot (2k+3)}{6}) \\
& && \text{(basic arithmetic: } 2k^2 + 7k + 6 = (k+2) \cdot (2k+3)\text{; multiply the right hand side to verify)} \\
&= \frac{n \cdot (n+1) \cdot (2n+1)}{6}. && \text{(since } n = k + 1\text{)}
\end{aligned}$$

  So we proved the induction step, which finalizes the proof.

Simple proofs by induction for algorithms? Check "homework 0" (or the binary search proof below).

2 Introduction to Sorting

Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or an array of numbers could be sorted by their values. Sorting is a key primitive in computer science with various applications: the most basic one being a necessary step for efficient *searching* (if you really need a compelling reason as an application of sorting, imagine looking up a some words in a dictionary that contains words in no particular order).

More formally, we define sorting and searching (in a sorted array) as the following two problems:

Problem 1 (Sorting)**.**

- **Input:** An array A of n integers;
- **Output:** The same array A ordered in *non-decreasing* order of its entries.

Note that we defined a very basic variant of sorting above; it turns out this is already enough to capture most other interesting sorting scenarios (sorting with different objective orders, sorting based on a particular key in <key, value> pairs, etc.).

There are numerous efficient and not-so-efficient algorithms for sorting. We will see several canonical examples of these algorithms in this course.

Problem 2 (Searching in a sorted array)**.**

- **Input:** A sorted array A of n integers, and a single *target* integer t;
- **Output:** Output whether t appears in A or not.

Again, we only defined a very basic variant of searching above; this is also enough to capture most other interesting searching scenarios (search for the value of a particular key in <key, value> pairs, etc.).

Unlike sorting that admits various algorithms, there is essentially a single algorithm for searching in a sorted array: *binary search*. We continue the lecture by presenting this basic algorithm.

2.1 Binary Search

Before getting to any sorting algorithm, let us first describe an algorithms for searching. Binary search allows us to determine whether a given integer exists in a sorted array in only $O(\log n)$ time (assuming the array is already in the memory and we do not need to 'read' it of course).

Binary Search Algorithm: The input is a sorted array A of n integers and a single target integer t.

1. If $n = 0$ return 'No' (this is the base case of this recursive algorithm).
2. Let $m = \lfloor n/2 \rfloor$:
   - if $A[m] = t$, output 'Yes' and terminate;
   - if $A[m] > t$, recursively solve the problem for t in the array $A_{<m} := A[1 : m-1]$;
   - if $A[m] < t$, recursively solve the problem for t in the array $A_{>m} : A[m+1 : n]$.

Proof of Correctness: The proof is by induction on size of the array A, i.e., n. The base case of the induction is when $n = 0$ for which the algorithm is definitely correct as an array of size 0 contains no element. Now suppose by induction that the algorithm correctly solves the problem on every array of size up to i and we prove it for $i + 1$.

Consider an array A of size $n = i + 1$ and define $m = \lfloor n/2 \rfloor$ as in the algorithm. If $A[m] = t$, the algorithm correctly returns t belongs to the array. Otherwise, if $A[m] > t$, since A is sorted in non-decreasing order, we know that all entries $A[j]$ for $j > m$ are also *larger* than t; so if t belongs to A it can only because t also belongs to $A_{<m}$ defined in the algorithm. By induction hypothesis, the algorithm correctly determines whether t belongs to $A_{<m}$ or not, hence proving the induction step in this case also. The proof of the case $A[m] < t$ is symmetric.

Runtime Analysis. Let $T(n)$ denote the worst-case runtime of the algorithm on any input of size n. We write the following recursion for the runtime of the algorithm:

$$T(n) \leq T(n/2) + \Theta(1)$$
(since we spend $\Theta(1)$ time to decide whether we terminate or recurse on an array of size $n/2$)
$$T(1) = \Theta(1). \qquad \text{(base case)}$$

To solve this recursion, we do as follows. Firstly, let k be the *smallest integer* such that $n \leq 2^k$. So, by definition, $k = O(\log n)$. Moreover, define the recursion:

$$S(k) = S(k-1) + \Theta(1)$$
$$S(0) = \Theta(1).$$

We claim that $T(n) \leq S(k)$; this can be proven for instance by induction since, $T(n) \leq T(n/2) + \Theta(1) \leq S(k-1) + \Theta(1) = S(k)$.

Now, using the substitution, we can write:

$$S(k) = S(k-1) + \Theta(1) = S(k-2) + \Theta(1) + \Theta(1) = \ldots = \underbrace{\Theta(1) + \cdots + \Theta(1)}_{k} = \Theta(k).$$

This implies that $T(n) = O(k)$ (notice that we switched from Θ-notation to O-notation simply because the inductive proof above gives us an inequality for $T(n)$ not an equality). Finally, since $k = O(\log n)$, we also have $T(n) = O(\log n)$.

Before we move on, let us mention that this is a common type of recursion that appear frequently in this course. So it is worth remembering how to solve such recursions.

2.2 Selection Sort

To conclude these lecture notes, let us describe a very simple (but not that efficient) algorithm for sorting called the *selection sort* which takes $O(n^2)$ time for sorting an array A of size n. The algorithm basically picks the minimum of the array and swap it with $A[1]$; then find the second minimum and swap it with $A[2]$, and continue like this until the whole array is sorted.

Selection Sort Algorithm: The input is an array A.

1. For $i = 1$ to n do:
   (a) For $j = i + 1$ to n do:
      i. If $A[j] < A[i]$, swap $A[i]$ and $A[j]$.

Proof of Correctness: The proof is by induction on the index of the outer loop: we prove that at the end of each iteration i, $A[1 : i]$ contains the *first smallest* i integers in A in the *non-decreasing order*. Thus, at the end of the last iteration, i.e., for $i = n$, we have A sorted in the non-decreasing order. For the base case, we simply take $i = 0$ which ensures the base case holds correctly.

We now prove the induction step. Suppose it is true up to some iteration i and we prove it for $i + 1$. By induction, $A[1 : i]$ already contains the first smallest i integers in A in the non-decreasing order. Now notice that the inner loop is simply computing the *minimum* of the array $A[i + 1 : n]$ and place it in $A[i + 1]$. This can also be proven by induction; in fact we already did this in "homework 0" and we are not going to repeat the argument here. This, together with the above implication of the induction step, implies that $A[1 : i + 1]$ contains the first smallest $i + 1$ integers in A in the non-decreasing order, finalizing the proof.

The type of induction we did here is sometimes called proving a *loop-invariant*: basically, we proved that a certain property holds at each iteration of the (outer) loop in the algorithm and did this similar to an induction, by proving a base case and a step. Using the term loop-invariant is in fact a more accurate description of the proof above because unlike an induction that goes to infinity, we are only interested in proving the loop-invariant for up to the last iteration (iteration n). However, for simplicity, we are going to refer to the above proof also as induction.

Runtime Analysis. The outer loops takes n iteration, for each iteration i, the inner loop has $n - i$ iterations, and each iteration of the inner loop takes $\Theta(1)$ time. As such, the runtime of the algorithm is:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \Theta(1) = \sum_{i=1}^{n} (n - i) \cdot \Theta(1) = \sum_{i=1}^{n} i \cdot \Theta(1) = \Theta(n^2).$$