CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 1: Instructions

Harvard SEAS - Fall 2021 *Sept. 7, 2021*

One of the skills we want you to develop in CS120 is to be able to manage the complexity of mathematical proofs by thinking about them at different levels of abstraction, ranging from a high-level intuitive proof outline to being able to work out the low-level formal details. Importantly, these are not two separate skills—rather you need to learn to be able to *move between* these different levels of thinking about proofs, e.g., extracting the high-level intuition from a formal proof, and turning a high-level proof strategy into a detailed formal proof. The reason this skill is particularly valuable for you as computer scientists is that software and hardware systems can be extremely complex to design and understand; to manage that complexity, we need to be able to work at a high level, but to complete an implementation or analysis, we also need to be able to fill in the technical details.

To develop this skill, we will do in-class active learning exercises where half of you (the "Senders") are tasked with reading and understanding a formal proof, and then explaining it a high level to a classmate (a "Receiver"), who is then tasked with writing it down and filling in the formal details. In addition, these exercise will also have the benefit of reinforcing important concepts related to the content of the course.

The structure of the exercise is as follows. *Before class,* both the Sender and Receiver should study the reading assigned to them. The Receiver reading is short and only contains the theorem statement and motivation. The Sender reading contains a proof a proof of the theorem and should be studied more carefully. If you are a Sender, you should try to extract both the main ideas (low-to-high level translation) and the technical details. You may create some bullet notes or a high-level summary for yourself to help you in your presentation during class. Avoid the temptation to write a fully detailed script or to memorize the proof; instead try to internalize the ideas so that you can reproduce the proof based on an intuitive understanding.

In class, the exercise will proceed as follows:

1. Senders and Receivers form pairs (or triples if we end up with imbalanced attendance), introduce themselves to each other, and find an open whiteboard or desk.

2. The Sender describe the theorem and proof to the Receiver through an oral, interactive dialogue. The Sender should not do any writing, but the Receiver should be capturing their understanding by writing pictures and notes. The Receiver should ask questions of the Sender during the interaction. After the high-level understanding is transmitted, the dialogue should continue on to filling in as many formal details as possible.

3. Senders and Receivers can raise their hands if they are jointly stuck, and a member of the teaching staff will come to help.

4. After the exercise is complete (or at the end of class), we will have you fill out a reflection survey, which will provide valuable feedback for us and will give you material for your Participation Portfolios.

CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 1: Reading for Receivers

Harvard SEAS - Fall 2021 *Sept. 7, 2021*

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to practice reasoning about the running time of algorithms
- to see how the choice of a model of computation can affect the computational complexity of a problem

In the previous class, we have seen that the (worst-case) computational complexity of sorting in the comparison model is $\Theta(n \log n)$. That is, there are sorting algorithms that have worst-case running time $O(n \log n)$, and every (correct) sorting algorithm has worst-case running time $\Omega(n \log n)$. This holds even when the keys are drawn from the universe $U = [n]$.

In our first active learning exercise, you will see that for keys drawn from the universe $[n]$, it is actually possible to sort asymptotically faster — in time $O(n)$! How is this possible in light of the $\Omega(n \log n)$ lower bound? Well, the algorithm will not be a comparison-based one; it will directly access and manipulate the keys themselves (rather than just comparing them to each other).

More generally, we will show:

Theorem 0.1. *There is an algorithm for sorting an array of n item-key pairs where the keys are drawn from a known universe of size U with (worst-case) running time $O(n + U)$.*

Since we have not yet precisely defined our computational model or what constitutes a "basic operation," this theorem and its proof are still informal. It is possible to improve the dependence on U in this theorem from linear to logarithmic; we may include that in the next problem set.

Proof. 1. Algorithm:

2. Correctness:

3. Runtime:

□

CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 1: Reading for Senders

Harvard SEAS - Fall 2021 *Sept. 7, 2021*

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to practice reasoning about the running time of algorithms
- to see how the choice of a model of computation can affect the computational complexity of a problem

In the previous class, we have seen that the (worst-case) computational complexity of sorting in the comparison model is $\Theta(n \log n)$. That is, there are sorting algorithms that have worst-case running time $O(n \log n)$, and every (correct) sorting algorithm has worst-case running time $\Omega(n \log n)$. This holds even when the keys are drawn from the universe $U = [n]$.

In our first active learning exercise, you will see that for keys drawn from the universe $[n]$, it is actually possible to sort asymptotically faster — in time $O(n)$! How is this possible in light of the $\Omega(n \log n)$ lower bound? Well, the algorithm will not be a comparison-based one; it will directly access and manipulate the keys themselves (rather than just comparing them to each other).

More generally, we will show:

Theorem 0.1. *There is an algorithm for sorting an array of n item-key pairs where the keys are drawn from a known universe of size U with (worst-case) running time $O(n + U)$.*

Since we have not yet precisely defined our computational model or what constitutes a "basic operation," this theorem and its proof are still informal. It is possible to improve the dependence on U in this theorem from linear to logarithmic; we may include that in the next problem set.

Proof. We assume without loss of generality that they keys come from $[U]$. (Since the universe of size U is known, we can map the keys to $[U]$ while preserving the order of elements.)

Our algorithm is a variant of "Counting Sort". Counting Sort is typically presented for a case where there are no items, and we are just sorting an array of keys from the universe U. In Counting Sort, we initialize an array C of length U to have zeroes in every entry. Then we make a pass over the array A of keys, incrementing $C[A[i]]$ when we are at the i'th element of A. At the end of this pass, for each key $K \in [U]$, $C[K]$ will have a count of the number of elements of A that have key K. We now make a pass over C, filling in our output array A' from beginning to end with $C[K]$ elements of value K as we go. To generalize this idea to sorting arrays of item-key pairs, we replace

the counts in the array C with linked lists of items, yielding the following algorithm:

Input : An array $A = ((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, where each $K_i \in [U]$
Output : A valid sorting of A

1 Initialize an array C of length U, such that each entry of C is the start of an empty linked list.
2 **foreach** $i = 0, \ldots, n-1$ **do**
3 | Append (I_i, K_i) to the linked list $C[K_i]$.
4 Form an array A that contains the elements of $C[0]$, followed by the elements of $C[1]$, followed by the elements of $C[3]$, ...
5 **return** A

Algorithm 1: Counting Sort with Items

To show the correctness of Algorithm 1, we observe that after the loop, for each $j \in [U]$, the linked list $C[j]$ contains exactly the item-key pairs whose key equals j. Thus concatenating these linked lists into a single array will be a valid sorting of the input array.

For the runtime analysis, initializing the array takes time $O(U)$. Each iteration of the loop takes time $O(1)$, for a total loop runtime of $O(n)$. And concatenatening the elements of $C[j]$ into the array A takes time $O(1) + O(|C[j]|)$, where $|C[j]|$ is the length of the linked list $C[j]$. Thus, forming the array A takes time

$$\sum_{j=1}^{U} O(1) + O(|C[j]|) = O(U + \sum_{j} |C[j]|) = O(U + n).$$

Thus, we have a total runtime of $O(U + n) = O(n)$.

□