CIS 320: Algorithms September 13, 2021

Lecture 3

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Sorting

In this lecture, we'll recall a classic problem that you will have seen before: sorting. The task is simple and ubiquitious: given an array of numbers, permute the array so that the numbers are in ascending (or descending) order. We can define the problem formally as follows. Below we write $[n]$ to denote $[n] = \{1, \ldots, n\}$, A[i] to denote the i'th entry of a matrix A, with indexing starting at 1, and $A[i, \ldots, j]$ to denote the subarray from entries i to j (inclusive).

Definition 1 *An instance of the sorting problem is given by an array A of n numbers such that for $i \in [n]$, $A[i] \in \mathbb{R}$. The goal is to return an array B that is a permutation of A in ascending order. That is, we want that:*

1. *There exists a permutation $\pi : [n] \to [n]$ such that for each $i \in [n]$, $B[i] = A[\pi(i)]$*
2. *For each $i \in [n-1]$: $B[i] \leq B[i+1]$*

Suppose that we start with $B = A$. A simple operation that we might perform is a *swap*, that simply swaps the elements in the i'th and j'th entry of B. We will write this operation as $\text{swap}(B, i, j)$. If we design algorithms such that we only modify B by performing swap operations, then we will be guaranteed that B remains a permutation of A, and we will only need to worry about proving that our algorithms result in B being output in ascending order. We will imagine that we can perform a swap operation in constant time (in other words, the amount of time it takes us to perform $\text{swap}(B, i, j)$ is independent of n, the length of the array, which is a good model of reality at least for arrays that can fit in memory). Another operation that will be useful is finding the minimum element in some subarray $B[t+1, n]$.

Algorithm 1 FindMinIndex(B[t+1,n])

```
Let MinIndex = t + 1.
for i = t + 2 to n do
  if B[i] < B[MinIndex] then
    MinIndex = i.
  end if
end for
Return MinIndex
```

It should be easy to convince yourself of the following claim:

Claim 2 *FindMinIndex($B[t+1, n]$) runs in time $O(n-t)$ and returns an index $t+1 \leq MinIndex \leq n$ such that for all $t+1 \leq i \leq n$: $B[MinIndex] \leq B[i]$.*

Proof It might be a helpful refresher on the principles of proof by induction to give a short proof of this. ■

We can now combine these two operations to derive a simple sorting algorithm, *Selection Sort*.

Algorithm 3 Selection Sort

```
Let B = A
for t = 1 to n − 1 do
  Let j = FindMinIndex(B[t + 1, n).
  if B[j] < B[t] then
    Swap(B, t, j).
  end if
end for
Return B.
```

The run time for Selection sort is easy to analyze:

Theorem 3 *Selection Sort runs in time $O(n^2)$.*

Proof We iterate t from 1 to $n-1$: in each iteration, we might conduct a Swap (constant time) and run FindMinIndex[B[t+1,n]), which by Claim 2 takes time $O(n-t)$. In all, the running time is:

$$O\left(\sum_{t=1}^{n-1} n - t\right) = O\left(\sum_{t=1}^{n-1} t\right) = O(n^2)$$

■

Now, can we prove that Selection "sort" deserves the name?

Theorem 4 *Selection Sort solves the sorting problem — i.e. for any A, it returns an array B that is a permutation of A in ascending order.*

Proof As we have observed, because Selection Sort only modifies the original ordering via the Swap operation, it is guaranteed to return a permutation of A — so it remains to prove that at completion, B is in ascending order. We prove this by induction on t. Our inductive hypothesis is that after the completion of iteration t, two things are true:

1. The sub-matrix $B[1, \ldots, t]$ is in ascending order, and
2. The entries before t are all less than the entries after t: for all $i \leq t$ and for all $j > t$, $B[i] \leq B[j]$.

We can establish the base case at $t = 1$: After iteration 1, we have that $B[1]$ contains a minimum value element in B, which satisfies both claims. To see this observe that j is the index of the minimum value in $B[2, \ldots, n]$ which we swap with the first element in the event that $B[j] < B[1]$.

To prove the inductive case, note that at the start of round $t+1$, we have that $B[j] \leq B[j']$ for all $j' \geq t+2$ by the guarantee of FindMinIndex, and that $B[j] \geq B[t]$ by the 2nd part of the inductive hypothesis. By the first part of the inductive hypothesis, $B[1, \ldots, t]$ is in sorted order, and in particular $B[j] \geq B[i]$ for all $i \leq t$. Thus when we perform Swap$(B, t+1, j)$ we obtain that $B[1, \ldots, t+1]$ is in sorted order, and that for all $i \leq t+1$ and all $j' > t+1$, $B[i] \leq B[j']$, establishing the inductive case.

Thus we have that t round $t = n$, by the first part of the inductive claim, that B is in ascending order. ■

Huzzah, we have a solution to the sorting problem! Can we do better? Perhaps we can improve on the $O(n^2)$ running time to get an algorithm that runs in time $O(n)$? To answer this question we need to be more precise about what a "solution" can do.

Selection sort inspects the input data using only a single operation: a comparison (i.e. its branching condition is of the form "If $B[j] \leq B[t]$ then...). It is the result of these comparisons (and nothing else) that determines which swaps are performed, which comparisons are performed next, and ultimately which permutation π of the input array A is finally output. That is to say, Selection Sort operates in the comparison based model of computation:

Definition 5 (Comparison Model) *An algorithm operates in the* Comparison Model *if it can be written as a binary decision tree in which:*

1. *Each vertex is labelled with a fixed comparison (i.e. $B[i] < B[j]$ for particular i, j)*
2. *Computation proceeds as a root-leaf path down the tree, branching left if the comparison evaluates to TRUE and right otherwise, and*
3. *The leaves are labelled with the output of the algorithm (in this case, permutations)*

In this model, the running time of the algorithm corresponds to the depth of the tree.

As it turns out, we can prove an easy lower bound for sorting algorithms in the comparison model. Lower bounds of this sort serve as a guide: *either* we should not waste effort trying to derive algorithms that improve on the lower bound, *or*, we should find techniques that step outside of the model in which the lower bound is proven.

Theorem 6 *Any algorithm that solves the sorting problem int he comparison model must have run time at least $\Omega(n \log n)$.*

Proof The proof is a simple counting argument. An algorithm that solves the sorting problem must output an array B in sorted order for *any* input A. Since A can be an arbitrary permutation of B, this requires that each permutation π is output by the algorithm on some input A. In the comparison based model, this means the decision tree must have at least one leaf for each permutation π: There are $n!$ such permutations, so we must have at least $L > n!$ leaves.

On the other hand, a binary tree of depth d has $L \leq 2^d$ many leaves. Here d is the running time of our algorithm, and so combining these two bounds, we have that:

$$2^d \geq n!$$

taking the log of both sides, we have:

$$d \geq \log(n!) = \Omega(n \log n).$$

■