CIS 320: Algorithms September 13, 2021

Lecture 4

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Divide and Conquer: Merge and Quick Sorts

In this lecture we'll give a sorting algorithm with running time that matches the $\Omega(n \log n)$ lower bound we proved for sorting algorithms in the comparison model in the last class. Our goal here is not just to understand sorting, but to see an example of a more general algorithmic paradigm called *divide and conquer*. The idea (at a high level) is to solve a problem by splitting it into two smaller problems, solving the smaller problems, and then combining the solution. Along the way, we'll get practice solving recurrence relations, another useful trick.

The main observation behind merge sort is that if we have two arrays A and B of length t that are *already in sorted order*, then we can merge them into a single array C also in sorted order in $O(t)$ time, using the following merge operation. In practice you would want to perform the merge in place — i.e. without having to allocate another array C, but our goal here is clarity rather than memory efficiency.

Algorithm 1 Merge(A, B)

Let $\ell_A = \text{length}(A)$, $\ell_B = \text{length}(B)$, C be an array of length $\ell_A + \ell_B$.
Let $p_A = 1, p_B = 1, p_C = 1$.
while $p_C \leq \ell_A + \ell_B$ **do**
  if $p_A > \ell_A$ or $A[p_A] > B[p_B]$ **then**
    $C[p_C] = B[p_B]$
    $p_C += 1$, $p_B += 1$
  else if $p_B > \ell_B$ or $A[p_A] \leq B[p_B]$ **then**
    $C[p_C] = A[p_A]$
    $p_C += 1$, $p_A += 1$
  end if
end while
Return C

Claim 1 *Given arrays A and B of length ℓ_A and ℓ_B respectively, both in ascending order, Merge(A, B) runs in time $O(\ell_A + \ell_B)$ and returns an array C containing the concatenation of A and B sorted in ascending order.*

Proof Prove this for yourself as an exercise if you want to practice your induction skills. ■

The Merge operation tells us that if we have two halves of our array that are each *already* in sorted order, then with $O(n)$ additional work, we can combine them to get our array in completely sorted order. This suggests a natural idea for sorting:

1. Split our array into two parts;
2. Sort each part, and;
3. Merge the two parts.

Of course, step 2 is "Sort each part" — seemingly the exact problem we are trying to find a solution for! But we can *recurse* and solve these sub-problems the same way, observing that eventually, when we get down to an array of size 1, it will already be sorted without us having to do any additional work. In the following, we will assume for simplicity that n is a power of 2, but you should be able to convince yourself that handling the case in which n is not a power of 2 is simple and can be done without asymptotically affecting the analysis of our algorithm.

Algorithm 2 MergeSort(A)

Let $\ell = \text{length}(A)$
if $\ell = 1$ **then**
  Return A
else
  Let $A_1 = A[1, \ldots, \ell/2]$ and $A_2 = A[\ell/2 + 1, \ell]$
  Let $B = \text{MergeSort}(A_1)$ and **Let** $C = \text{MergeSort}(A_2)$
  Return $\text{Merge}(B, C)$.
end if

Theorem 2 *MergeSort(A) halts and returns an array sorted in ascending order.*

Proof We first observe that MergeSort halts: Recall that we have assumed that n is a power of 2, and hence at every call of MergeSort, $\ell/2$ is an integer. Hence each recursive call to MergeSort is well defined and applied to a matrix of strictly smaller length, and bottoms out at $\ell = 1$, which returns immediately. Hence all calls return.

We can prove correctness by induction on ℓ. The base case is when $\ell = 1$: in this case, A is by definition in sorted order, and MergeSort correctly returns A.

For the inductive case, we assume that the inductive hypothesis holds for arrays of length ℓ, and conclude that it also holds for arrays of length 2ℓ. If A is of length 2ℓ, then A_1 and A_2 are both of length ℓ, and so by the inductive assumption, B and C are permutations of A_1 and A_2 in sorted order. Thus, by Claim 1, our returned value $\text{Merge}(B, C)$ is a permutation of A in sorted order. ■

We now analyze the running time of MergeSort. It is a recursive algorithm (i.e. it makes calls to itself on smaller instances), and so it is natural to write down a recursive equation for its running time. Lets let $T(n)$ denote the running time of MergeSort on an input of length n. MergeSort does two things of note: It makes two recursive calls on instances of length $n/2$, and it performs the Merge operation, which by Claim 1 takes time $O(n)$. Hence, for some constant c, we can conclude:

$$T(n) \leq 2T(n/2) + cn$$

We need to solve for $T(n)$. It turns out that if we have a good guess, then establishing the guess as correct is a simple exercise in induction:

Theorem 3 *MergeSort runs in time $O(n \log n)$.*

Proof We prove this by induction: Our inductive hypothesis is that for an input of size $\ell \geq 2$, there is a constant c such that $T(\ell) \leq c\ell \log \ell$. In the base case when $\ell = 2$, MergeSort returns in constant time, and so this is satisfied. In the inductive case, we assume this to be true for inputs of size ℓ and show that it is true for inputs of size 2ℓ. Using our recursive bound on the running time of MergeSort and substituting in for our inductive assumption, we get:

$$\begin{aligned} T(2\ell) &\leq 2T(\ell) + 2c\ell \\ &\leq 2c\ell \log \ell + 2c\ell \log 2 \\ &= 2c\ell(\log \ell + \log 2) \\ &= c(2\ell) \log(2\ell) \end{aligned}$$

which completes the proof. ■

How would you come up with such a guess in the first place, to plug into the induction? Its helpful to draw out the recursion as a tree, and count the running time per level of the tree, as well as the depth of the tree. We drew out a tree like this in class.

Lets do one more — Quick Sort! This will also be a comparison based sorting method, so we know from our lower bound that it cannot beat MergeSort's asymptotics. But it can be implemented very efficiently; for our purposes, we'll be interested in it because it is a randomized algorithm.

Like MergeSort, QuickSort is a recursive algorithm based on a divide and conquer methodology. But instead of Merge, the base operation will be *Partition*: Given a pivot element z, we will divide an array A into two arrays B and C such that every entry of B is less than z, and every entry of C is greater than z. Once again we will prioritize clarity over efficiency of the details of implementation (so we will not sort in place, as you would want to in practice).

Algorithm 3 Partition(A, z)

Let $\ell = \text{Length}(A)$ and **initialize** new arrays B and C.
Initialize indices $p_B = 1$, $p_C = 1$.
for $p_A = 1$ to ℓ **do**
  if $A[p_A] \leq z$ **then**
    $B[p_B++] = A[p_A]$
  else
    $C[p_C++] = A[p_A]$
  end if
end for
Return B, C

Claim 4 *Partition(A, z) runs in time $O(\ell)$ on an input A of length ℓ, and outputs a pair of arrays B, C such that $B[i] \leq z$ for all $i \leq$ Length(B) and $C[i] > z$ for all $i \leq$ Length(C).*

Proof Left as an exercise. ■

The idea behind quick sort will be to pick a pivot element z at random, then to partition A around z, and recursively call QuickSort on B and C.

Algorithm 4 QuickSort(A)

Let $\ell = \text{Length}(A)$
Pick $i \in \{1, \ldots, \ell\}$ uniformly at random and let $z = A[i]$.
Let $A_{-i} = A[1, \ldots, i-1] \circ A[i+1, \ldots \ell]$
Let $(B, C) = \text{Partition}(A_{-i}, z)$
Let $B_s = \text{QuickSort}(B)$, $C_s = \text{QuickSort}(C)$.
Return $B_s \circ z \circ C_s$

Theorem 5 *QuickSort(A) returns a permutation of A in sorted order.*

Proof We prove this by induction on ℓ. In the base case $\ell = 1$, QuickSort returns $z = A$, which is in sorted order by definition. In the inductive case, we have that B_s and C_s are in sorted order. We also have from Claim 4 that for all i, j: $B_s[i] \leq z \leq C_s[j]$. Thus $B_s \circ z \circ C_s$ is also in sorted order. ■

It remains to analyze the running time of QuickSort. Note that since we pick the pivot element z at random, we need to analyze the expected running time. We observe the running time of the algorithm is dominated by the for loop in Partition, each iteration of which performs a comparison. Hence to bound the running time asymptotically, it suffices to bound the number of comparisons made in total.

Theorem 6 *QuickSort has expected running time $O(n \log n)$ when run on an input A of length n.*

Proof We will count the expected number of comparisons that QuickSort makes over the run of the algorithm — each of which occurs within a call to Partition. Observe that recursive calls are made on disjoint parts of the array, so elements are compared at most one time.

Let A_s denote A in sorted order. Let $X_{i,j}$ denote the indicator that $A_s[i]$ is compared to $A_s[j]$ at some point during the run of the algorithm. Since elements are compared at most one time, this means we can bound the total number of comparisons X in expectation as

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}] \end{aligned}$$

where the second equality follows by linearity of expectation. Thus we can focus on the term $\mathrm{E}[X_{i,j}]$.

To analyze this, observe that if some point $z = A_s[k]$, where $i < k < j$ is chosen as a pivot *before* either i or j are chosen, then $A_s[i]$ and $A_s[j]$ will never be compared to one another, because they will be separated into disjoint arrays after each is compared to z. Hence, i and j will be compared to one another *only if* either $A_s[i]$ or $A_s[j]$ are the *first* elements to be chosen as pivots amongst the set $A_s[i, \ldots, j]$. Because we choose pivots uniformly at random, and there are $j-i+1$ elements in $A_s[i, \ldots, j]$, the chance that the first one chosen is in the set $\{A_s[i], A_s[j]\}$ is:

$$\mathrm{E}[X_{i,j}] = \frac{2}{j-i+1}.$$

Thus the expected number of comparisons, and hence the expected running time can be bounded as:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{i,j}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\ &\leq \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} \\ &= 2 \cdot \sum_{i=1}^{n-1} H_n \\ &= O(n \log n) \end{aligned}$$

Here, $H_n = \sum_{k=1}^{n} \frac{1}{k} = O(\log n)$ is the n'th Harmonic number.

■