.2

CS 218 : Design and Analysis of Algorithms

Lecture 5.2: Sorting

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

Let us apply the design principles to sorting. Assume the input array is $A[1, \ldots, n]$.

The first thing we try is: Recursively sort the first $n-1$ elements.
Find the position of $A[n]$ among the first $n-1$ elements. Say it is r. Shift the elements from r to $n-1$ to the right by one and insert the new element at the correct position. We find the position by binary search. That takes $O(\log n)$ comparisons. However the shifting takes $O(n)$ time in the worst case.

Exercise. Write the recurrence for the number of comparisons and solve it.

We focus on the number of shifts which dominates the running time. Let $T(n)$ denote the time taken by this algorithm. The recurrence defining $T(n)$ is $T(n) \leq T(n-1) + n$. The solution to this recurrence is $O(n^2)$. We would like to reduce the running time. This is a common phenomenon in algorithm design. You run through the design, end up with an algorithm and then try and improve it. It is here that your thinking begins.

There are two ways in which one may proceed. One is to see if a better data structure may be used. The other is to try other ways of dividing the input. For instance into two equal parts.

We begin with the first option. To translate the algorithm design issue to a data structure question, we ask: What operations do we perform during the algorithm? Here the operations we perform with $A[n]$ is find and insert. Note that we are maintaining a (sorted) array to do this. We should not be maintaining a sorted array under inserts. A data structure now suggests itself! We should be using a BBST. If we do this, we scan the array from left to right inserting the elements into the BBST. The final sorted order can be recovered by doing an inorder traversal of the binary tree. This yields an $O(n \log n)$ algorithm. It also motivates the use of BBSTs.

The other way is to consider a different order. Say split the input in two equal parts. Recurse on the two halves and then put the solutions together. We started designing an algorithm for sorting, but now we have a new problem to solve to implement the *putting the solutions together*.

The new focus: Design an algorithm to merge two sorted arrays or lists. How do we do this? Recursion again. We compare the first element of the two arrays/lists and pick the minimum of them and recurse on the leftovers. Let m and n be sizes of two lists to be merged. The number of comparisons now has the recurrence:

$T(m, n) \leq 1 + \max\{T(m-1, n), T(m, n-1)\}$. The max is because we do not know what the result of the first comparison will be. Prove that the solution to this recurrence is $m + n - 1$.

The iterative solution to the merging problem is this: maintain two pointers on the two arrays/lists. Say the pointers are L and R. Compare the two minimums and move the minimum of the two as the first element of the result. Now advance one of the pointers. The analysis follows by observing that with each comparison you move either L or R. So the total number of comparisons will be the total number of pointer movements which is $m + n - 1$.

Returning to the analysis of the sorting algorithm, our recurrence is $T(n) \leq 2T(n/2) + n$. *Exercise* Prove that this is $O(n \log n)$. This algorithm is called *Mergesort*.

Here is the other way to recurse. Find the minimum element of the array. Put it in the first place and recurse on the rest. Given that finding minimum takes $O(n)$ time, this yields an

$O(n^2)$ algorithm. To improve this notice that the operations we need are *findmin* and *deletemin*. The data structure should be obvious.

Exercise. Complete this $O(n \log n)$ algorithm to sort. This algorithm is called *Heapsort*.