.2

CS 218 : Design and Analysis of Algorithms

Lecture 6.2: Inversion Pairs

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

1 Counting the number of Inversion Pairs

In a array A, a pair of indices $i < j$ is called an *inversion pair* if $A[i] > A[j]$. If we require the array to be sorted in increasing order, an inversion pair is a pair which is not in order.

The Problem:

Input: An array A.

Output: The number of inversion pairs in A.

That is the number of pairs $i < j$ such that $A[i] > A[j]$. We assume that the elements of the array are distinct.

A brute force iterative algorithm compares every pair of elements resulting in $\in O(n^2)$ comparisons.

We begin the design process by assuming we have the solution for the first $n-1$ elements of the array and then see how to extend the solution to include the last element.

The main problem to solve in the design step is that given the element A[n], determine the number of inversion pairs it will be contained in. That is the number of elements amongst the first $n-1$ elements which are larger than $A[n]$. This will take $O(n)$ time yielding an $O(n^2)$ algorithm. Our aim is an $O(n \log n)$ solution.

The key thing to notice is this. If the previous array, $A[1, \ldots, n-1]$, were sorted, we can determine the number of new inversion pairs in time $O(logn)$. This leads to souping up the induction by returning a sorted array, which means that during the recursion, we should also return a sorted array. While doing this we run into a familiar problem: insertion into a sorted array. We have seen two ways to deal with this and we investigate both in turn.

The first is to divide into two equal parts, recurse on both, and then put them together. Recall that the output is both the answer and a sorted array. To get a final sorted array we merge the two sorted array. The only thing missing is the number of inversion pairs i, j where i is in the left subarray and j in the right.

Exercise. Show that this can be determined during the merging step with constant overhead.

Hence this does yield an $O(n \log n)$ algorithm.

The other approach is to design a data structure which supports operations **insert**(x) and **findrank**(y). Recall that the *rank* of an element is the position of the element in the sorted array. The operation **findrank**(y) returns the position of the element y in the sorted array among the elements currently in the data structure.

One way to do this is to augment balanced binary search trees to handle findrank. This is an exercise. The hint is to store an additional value in each node of the BBST. This is not the rank of the element. Note that you must be able to update the value easily during rotations and each operation must still take $O(\log n)$ time.

Street Smart Algorithm Design. It is possible to come up with algorithms without using the discipline that we have outlined so far. It is however imperative that you also have a proof

of correctness to go with the algorithm. For algorithms designed using induction, the proof is usually an obvious induction. We illustrate this with the following exercise.

Consider the following algorithm for computing the number of inversion pairs.
Sort the array A. Let $P(i)$ denote the position of $A[i]$ in the sorted array. Output

$$(1/2)\Sigma_i|i - P[i]|$$

Exercise. Either prove that the above algorithm is correct or find an array on which it gives the wrong answer.