• DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  – Homework 1 out, homework policies.
  – Recursive algorithm for inversion counting.
  – Guess-and-Check on $T(n) = 2T(n/2) + O(n \log n)$.
  – Master Theorem for solving runtime recurrences.

• Comments from last time
  – T-square is empty! Course webpage: http://www.cc.gatech.edu/~rpeng/3510F16/.
  – Guess and check is confusing, why does $-C + O(1)$ cancel?
  – Recursive structure of the algorithm unclear.
  – Marker too thin, write higher.

We continue with the inversion counting problem, which is given array Arr[1…n], finds the number of pairs $i < j$ such that $Arr[i] > Arr[j]$.

Last time we established that the number of inversions that involves $j$ in some range $[L, R]$ and some $i < L$ is the rank of $Arr[i]$ among the numbers $Arr[L \ldots R]$. Also, sorting on a set, and binary searching on it gives a way to compute the rank of a query element in a list in $O(n \log n)$ pre-processing time and $O(\log n)$ query time.

With this routine in mind, we get a divide-and-conquer algorithm where we first try to sort out all the inversion pairs crossing the middle, $i \leq m < j$. This sub-problem is significant because for every $i$, the set of possible choices of $j$ is fixed: it’s just $Arr[m + 1 \ldots n]$. This enables us to reuse the work done by sorting $Arr[m + 1 \ldots n]$ as a pre-processing step. In contrast, the naive algorithm of computing exactly the rank of each $Arr[i]$ in $Arr[i+1 \ldots n]$ is expensive because the list where it asks for rank on is changing at every step.

Going back to the example earlier, we split the array into

$$Arr[1 \ldots 4] = 8, 7, 12, 3; \quad Arr[5 \ldots 8] = 5, 11, 2, 9,$$

The rest of this class will discuss ways of bounding the performance of this algorithm. The cost of a recursive call on an array of size \( n \), \( T(n) \), is:

- Sorting the second half, \( O(n \log n) \) time.
- Performing a binary search for each element in the first half on the sorted copy of the second, \( O(n) \cdot O(\log n) = O(n \log n) \) time.
- Two recursive calls to problems half as big, \( 2 \cdot T(n/2) \).

Adding these together gives the runtime recurrence

\[
T(n) = 2T(n/2) + O(n \log n),
\]

which we then solve with guess-and-check.

1. Guess: \( T(n) \leq Cn \log^2 n \).
2. Assume \( T(n/2) \leq C n \log^2 (n/2) = C(n/2)(\log_2 n - 1)^2 \).
3. Plug into recurrence and check:

\[
2C(n/2)(\log_2 n - 1)^2 + Cn \log_2 n \leq Cn \log^2 n
\]

This process seems a bit magical. To get a better feeling of where the runtime is coming from, consider the size of the arrays that we sort/query on. At the topmost level, it’s one copy of size \( n/2 \). This copy makes two recursive calls, each to instances of size \( n/4 \). These two instances then makes a total of 4 recursive calls, to instances of size 8. This leads to a recursion tree, where at each level, the total size of the instances is at most \( n/2 \), and the size of the arrays called on halves. Therefore the total size of the arrays that we perform queries on is \( O(n \log n) \). Incorporating the cost of the sort / binary search then gives \( O(n \log^2 n) \).

This kind of analysis can also be formalized. It leads to the Master theorem for analyzing runtime recurrences.

**Theorem 0.1.** If \( T(n) = aT(n/b) + O(n^d) \) for constants \( a > 0 \), \( b > 1 \), and \( d \geq 1 \), then

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a.
\end{cases}
\]

In our case with the recurrence \( T(n) = 2T(n/2) + O(n \log n) \), if we ignore the additional \( \log n \) factor, we have \( a = b = 2 \), and \( d = 1 \). This is the case with \( d = \log_b a \), so we get the result is \( T(n) = n^d \log n \). Putting the extra factor of \( \log n \) back then gives \( n \log^2 n \).

By the way, partition into two-halves, and stitch results together routine is very close to merge-sort. There the result returned by the recursive calls is that the (sub) array is sorted. Merging two sub-arrays then takes \( O(n) \) time, giving the \( O(n \log n) \) total there. This routine is discussed in Chapter 2.3. of the textbook.