• **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.

• Administration
  
  − Website: http://www.cc.gatech.edu/~rpeng/3510F16/
  
  − Topics covered:
    * How to provably solve computational problems efficiently.
    * In order of use rate: heuristics, optimization, greedy, dynamic programming, divide-and-conquer.
    * Proofs: why these algorithms work, and when do they break.
    * In order of proof complexity: divide-and-conquer, greedy, graph algorithms, dynamic programming, optimization / approximation heuristics.

  − Course staff:
    * Easiest way to communicate with course staff is Piazza.
    * TAs will hold office hours / recitations.
    * Homeworks uploaded through T-square

  − Evaluation:
    * (25%) Final exam: Friday December 9, 11:30am - 2:20pm.
    * (60%) Best 3 out of 4 tests on Fridays Sep 9, Sep 30, Oct 21, Nov 11.
    * (15%) 5 homeworks due on Fridays Sep 2, Sep 23, Oct 14, Nov 4, Dec 2.
      · Homework submission through T-squared
      · Late homework policy: in the works.

  − Section A: 2pm in KACB 1443, attempts will be made to synchronize schedule.
Things covered today:
- Big-O, Big-Ω.
- What it means to be efficient.
- Example of a divide-and-conquer algorithm.

Inversion Counting
- Given length n array Arr[1...n], count number of pairs i < j s.t. Arr[i] > Arr[j].
- Example: 1, 5, 4, 2, 3 has 5 inversions.
- $O(n^2)$ time algorithm: try all pairs i and j.

Faster?
- Why?
  * Number of operations per second of a CPU: $10^9$.
  * Size of webgraph: $10^9$.
  * Size of medical image: $1024^3 \approx 10^9$.
- Number of inversion pairs is $\Omega(n^2)$.
- Example: $n, n-1, \ldots, 1$.
- Faster algorithms need to not examine all inversion pairs.

How: divide-and-conquer.
- Sub-problem: pick some ‘middle’ $m$, resolve all inversions with $i \leq m < j$.
- Can then recursively count inversion pairs in Arr[1...m] and Arr[m+1...n].
- Breaks Arr[1...n] into Arr[1...m] and Arr[m+1...n].
- Need to know where each element of Arr[1...m] ‘ranks’ in Arr[m+1...n].
- Next time: fast ways of doing this through sorting and binary search.
The inversion counting problem is given an array

\[ Arr[1 \ldots n], \]

computing the number of pairs of indices \( i < j \) such that

\[ Arr[i] > Arr[j]. \]

This can be done in \( O(n^2) \) time by enumerating over all pairs \( i \) and \( j \). We will examine this problem in detail because it is one of the simplest instances where divide-and-conquer gives significant runtime improvements.

The formal definition of big-O is

**Definition 0.1.** We say a function \( f(n) \) is \( O(g(n)) \) if there exists constants \( c \) and \( n_0 \) such that for every \( n \geq n_0 \), we have \( f(n) \leq c \cdot g(n) \).

**note:** Section 0.3 of the textbook has a slightly different definition that only has \( c \) but not \( n_0 \). It is equivalent to this one by choosing a larger value of \( c \). The version stated here is a bit easier to use because it allows one to ignore constant sized special cases.

Such definitions are able to absorb underlying constants in each of the operations. This means we can treat each command in the program as a unit cost operation. In particular, an algorithm that examines the \( \binom{n}{2} \) pairs is in \( O(n) \) by setting both \( c \) and \( n_0 \) to 1 in the above definition.

On the other hand, the number of potential inversion pairs is \( \Omega(n^2) \). Consider a descending list:


Here all \( \binom{n}{2} \) pairs of indices are inversion pairs. Formally, \( \Omega \) means:

**Definition 0.2.** A function \( f(n) \) is \( \Omega(g(n)) \) if \( g(n) \) is \( O(f(n)) \).

Here \( n^2 \) is \( O(\binom{n}{2}) \) by setting \( c = 2 \) and \( n_0 = 1 \).

One way to think about \( O \) and \( \Omega \) is that they essentially behave like \( \leq \) and \( \geq \) for running time functions. \( f(n) \) being \( O(g(n)) \) means \( f \) is at most \( g \), while \( f(n) \) being \( \Omega(g(n)) \) means \( f \) is at least \( g \).

In our case, the fact that the number of inversion pairs is \( \Omega(n^2) \) means if we want to significantly better than \( n^2 \), e.g. obtain an \( O(n^{1.5}) \) time algorithm, we need to have a routine that does not examine all inversion pairs.

This can be achieved through divide-and-conquer. Here the general scheme is to partition the list into two halves of roughly equal size,

\[ Arr[1 \ldots m], Arr[m + 1 \ldots n], \]
where $m = \lfloor n/2 \rfloor$. Inversion pairs with $1 \leq i < j \leq m$ can be counted by recursing on $Arr[1 \ldots m]$, and pairs with $m + 1 \leq i < j \leq n$ can be counted by recursing on $Arr[m + 1 \ldots n]$.

So what’s necessary for a faster algorithm is a very fast way of computing inversion pairs with $1 \leq i \leq m$ and $m + 1 \leq j \leq n$. This problem is simpler than the following:

Given two sets of numbers $A$ and $B$, for each element in $A$, count how many elements in $B$ are less than it.

This is an algorithmic reduction: we’re restating the sub-problem in a way that simplifies how it can be solved. In particular, note that in this setup, we’re completely free to rearrange $A$ and $B$.

In particular, if we have $B$ sorted in an array form:

$$B[1] \leq B[2] \leq \ldots B[k]$$

for a query element $a$, we can find the index $i$ such that

$$B[i] < a < B[i + 1]$$

in $O(\log n)$ time using binary search (we can either assume $B$ is padded with an infinity entry at the end, or deal with the case of $a > B[k]$ separately). The total cost of doing this is:

1. $O(n \log n)$ for sorting $B$.
2. $O(\log n)$ for a binary search per entry of $A$.

As $A$ has $O(n)$ entries, this total cost is $O(n \log n)$.

This gives a sub-quadratic routine for counting all inversion pairs ‘across’ the middle. Note that the total number of inversions may still be $\Omega(n^2)$: everything in $A$ may be greater than everything in $B$. However, in this case, we’re still only spending $O(\log n)$ per element of $A$, instead of the $\Omega(n)$ inversions that involve it. Replacing the number of inversions that $a$ is involved in with the cost of this binary search is precisely where the running time speedup is coming from.

Next time we will see how to complete this divide-and-conquer algorithm.