• 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.

• – Optional class, logistical items deferred to Thursday.
  – Efficiency, and why efficient?
  – Two example algorithmic problems:
    1. Inversion count
    2. (multiple) path finding

1 Runtime Efficiency, and why?

This class, in a nutshell, is about designing algorithms with good provable per-
formances. These performances will be measured using the big-O notation. The
formal definition of big-O notation is:

Definition 1.1. 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 $f(n) \leq c \cdot g(n)$.

This notation essentially lets one compare between two functions parameterized by
input sizes. For example, an algorithm running in $O(n \log n)$ time is faster than an
algorithm that runs in $O(n^2)$ time because $n \log n$ is $O(n^2)$, but we cannot say $n^2$
is $O(n \log n)$.

We can also define the reverse direction, $\Omega(\cdot)$, via:

Definition 1.2. A function $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$.

We say an algorithm runs in $O(f(n))$ time if on any input of size $n$, the amount of
operation that it takes is $O(f(n))$. We can also define memory usage similarly. Note
that the running time can also be parameterized using two different parameters, e.g.
on graphs both the number of vertices an edges affect the running time.

The main reason for emphasizing algorithmic efficiency are that computers are actu-
ally quite computationally limited: a modern CPU performs around $10^9$ operations
a second, and a data center has up to $10^7$ machines.
On the other hand, a webgraph has about $10^9$ vertices, while a medical image has about $1024^3 \approx 10^9$, so even $O(n^2)$ time algorithms are cost prohibitive on these problems. The situation gets even worse when we get to problems where we don’t know routines that run faster than $O(2^n)$ time. Of course, real world data often tend to be significantly easier than worst case inputs. However, designing algorithms that work in the worst case is an approach that has proven to work well for many problems, and modeling more complicated input distributions is beyond the scope of this course.

2 A Concrete Problem: Inversion Counting

We now move into a concrete problem that’s representative of what we’ll focus on during this course. It’s one of Richard’s favorite problems on efficient algorithms. Given an array,

$$\text{Arr}[1 \ldots n],$$

we’d like to compute the number of pairs of indices $i < j$ such that

$$\text{Arr}[i] > \text{Arr}[j].$$

This can be done in $O(n^2)$ time by enumerating over all pairs $i$ and $j$. This is not too surprising since the number of potential inversion pairs is $\Omega(n^2)$.

On the other hand, one can get far more efficient routines ($O(n \log n)$ time or better). Here the general scheme is to partition the list into two halves of roughly equal size,

$$\text{Arr}[1 \ldots m], \text{Arr}[m + 1 \ldots n],$$

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

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.

As we will see later, such a routine becomes much faster if both $A$ and $B$ are sorted. It’s actually closely related to a key routine in the merge-sort algorithm: we will see more about it next class.