Linear Time Sorting
We proved that when we are using a comparison-based model, the best we could do was $\Omega(n \log n)$ runtime in the worst case of an algorithm.

What if we change models?

– Might we have new restrictions (recall that our sorts so far work on any types of comparable data).

– Could we have certain types of data where the runtime is vastly improved?
Memory “Sort” on Unique Integers

Create \texttt{arr[MAXINT]} = \{0\}

For each value in the input,
\texttt{arr[value]}=1;

Traverse \texttt{arr} from 1 to \texttt{MAXINT} and rebuild sorted list of values!

\# of comparisons to data?

amount of work done?
Counting Sort

Done on integers.
Values do not need to be unique.

```plaintext
max=Max(arr);
min=Min(arr);
range=max-min+2;
count=new array[range];
temp=new array[n];
for i=1 to range
    count[i]=0;
for i=1 to n
    count[arr[i]-min+1]++;
for i=2 to range
    count[i]+=count[i-1];
for i=n to 1
    temp[count[arr[i]-min+1]] = arr[i];
count[arr[i]-min+1]--;
for i=1 to n
    arr[i]=temp[i];
```
Radix Sort

The ordering of the data must be such that if we perform a stable sort based on the individual positions of the data going “right to left” we will end up with the data correctly sorted.

ie: sort padded integers by column:

738
059
132
007
561
Question (1)

If we have $n$ b-bit integers, can we sort them in $\Theta(b \cdot n)$ time?
Question (2)

How many bits are used to represent the numbers in the range 0…n-1?
Question (3)

What if we group the bits into clusters of size $r$?
Question (4)

Can we sort $n$ values that are in the range $0..n^2$ in $O(n)$ time?
“When/How” Radix Sort

We have seen that certain approaches are really order $n \log n$ while others are really linear.

How do we know the “right” radix to use (what should we cluster together as a column)?

In a generic sense, we can show that the best radix sort for $n$ values over a range of $s$ is order $n \log s / \log n$.

So, in the case of $n$ values over a range of $n^n$ the radix sort would at best give us a runtime of $n^2$. 
Bucket Sort

The sort runs in linear expected time.

We assume input that is uniformly distributed across the range of values.

Consider integers between 1 and 1000. Create $n$ “buckets” of equal range size (eg: $1 \ldots \frac{1000}{n}$, $\frac{1000}{n} + 1 \ldots 2\frac{1000}{n}$, ...). If each bucket is ordered, then we can obtain an ordered full list by traversing the buckets in order.