Recap:

Comparison-based Sorting algorithms (w/ perfect correctness) require $\Omega(n \log n)$ comparisons in the worst case!

But...

Lower bound can be circumvented if not comparison based.

e.g. Counting Sort $O(n+k)$. k is range of the numbers/keys

1. Counting Sort

A

3	5	5	2	5	6	key
"U"	"V"	"W"	"X"	"Y"	"Z"	data

- Sort the array according to the keys
- Assume keys are in $\{1, \dots, k\}$

count

0	1	1	0	3	1
1	2	3	4	5	6
	↓	↓		↓	↓
	"2"	"3"		"5 5 5"	"6"

- Algorithm works, but we need to preserve the data associated with each key, not just the keys themselves!

e.g. Think about sorting the final grades

```
CountingSort(A, k):
    L = array of k lists                        } O(k)
    For i = 1 to n:                             } O(n)
        L[A[i].key].append(A[i])     } O(1)
    output = []                                 } O(1)
    For j = 1 to k:                             } O(n+k)
        output.append(L[j])          } O(|L[j]|)
```

<u>Correctness</u>: trivial by design

<u>Runtime</u>: $O(n+k)$. So if $k = O(n)$, then runtime is $O(n)$.

Also <u>stable</u>!

But using these lists are kinda "meh..."
lots of copying around.

Next up: More practical version using 3 arrays

```
CountingSort(A, B, k)
    For j = 1 to k:
        C[j] = 0
    For i = 1 to n:
        C[A[i].key]++
    // C[j] is now # of elements with key j
    For j = 2 to k:
        C[i] += C[i-1]
    // C[j] is now # of elements with key ≤ j
    For i = n to 1:
        B[C[A[i].key]] = A[i]
        // There are k = C[A[i].key] # of elems with key
        // ≤ A[i].key, so we put A[i] into index k
        C[A[i].key]--
```

<u>Runtime</u>: $O(n+k)$

Can circumvent the lower bound because we don't use comparison!

2. Radix Sort

- Assume keys are in $\{0, \dots, k^d - 1\}$
- Think of them as d-digit numbers, with each digit in $\{0, 1, \dots, k-1\}$
- Put elements into "buckets" according to their $1^{st}, 2^{nd}, \dots, d^{th}$ digit. Also called "bucket sort".

2 ways to do this:

LSDRadixSort(A, k, d)
  For i = 1 to d:
    Stable sort entire array based on the i-th least significant digit (counting from the right)

MSDRadixSort(A, k, d, i) // call with i = 1
  Stable sort entire array based on the i-th most significant digit (counting from the left)
  For each subarray with the same i-th digit:
    Call MSDRadixSort recursively with i' = i + 1

Runtime: $O(d(n+k))$ (using counting sort)

If $k = n$, and the keys are bounded by n^c for some constant c, then runtime is $O(c \cdot (n+n)) = O(n)$!

$\boxed{\begin{array}{l}\text{Counting Sort: keys in } \{1, 2, \dots, k\},\ O(n+k) \\ \text{Radix Sort: keys in } \{1, 2, \dots, k^d - 1\},\ O(d(n+k))\end{array}}$