Sorting in O(n) time:

Assume all keys are integers in a bounded range 0…m.
Example: Exam scores are typically in range 0…100
Example: ASCII characters convert to range 0…127

```
CountingSort (A[1…n], m) {               // each A[k] in range 0…m
    allocate new arrays B[1…n] and C[0…m];
    for (j=0; j<=m; j++)
        C[j] = 0;
    for (k=1; k<=n; k++)
        C[A[k]]++;
    //  now each C[j] = number of copies of key j in array A
    for (j=1; j<=m; j++)
        C[j] += C[j–1];
    //  now each C[j] = total number of elements <= j in array A
    for (k=n; k>=1; k--) {
        B[C[A[k]]] = A[k];
        C[A[k]] --;
    }
    for (k=1; k<=n; k++)
        A[k] = B[k];
    }
}
```

Analysis of counting sort:

- $\theta(m+n)$ time, which is $\theta(n)$ if m is O(n)
- $\theta(m+n)$ space

Example: n=14, m=3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	2	1	0	2’	1’	1’’	3	0’	2’’	3’	1’’’	2’’’	0’’	1’’’’

	0	1	2	3
C	3	5	4	2

	0	1	2	3
C	3	8	12	14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B	0	0’	0’’	1	1’	1’’	1’’’	1’’’’	2	2’	2’’	2’’’	3	3’

	0	1	2	3
C	0	3	8	12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	0	0’	0’’	1	1’	1’’	1’’’	1’’’’	2	2’	2’’	2’’’	3	3’

Bin sort (also called Bucket sort):

Each bin or bucket is a queue with duplicate keys from A

```
BinSort (A[1…n], m) {                    // each A[k] in range 0…m
    allocate new array B[0…m];
    for (j=0; j<=m; j++)
        B[j] = new Queue( );
    //  first copy all the keys from A to the bins
    for (k=1; k<=n; k++)
        B[A[k]].enqueue (A[k]);
    //  next copy all the keys from the bins to A
    k = 0;
    for (j=0; j<=m; j++)
        while (! B[j].isEmpty( ))
            A[++k] = B[j].dequeue( );
}
```

Analysis of bin sort:

- Total number of iterations of the while-loop is n
- θ(m+n) time, which is θ(n) if m is O(n)
- θ(m+n) space
- The array B is similar data structure to the hash table with separate chaining

Example: n=14, m=3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	2	1	0	2’	1’	1’’	3	0’	2’’	3’	1’’’	2’’’	0’’	1’’’’

	B
0	0, 0’, 0’’
1	1, 1’, 1’’, 1’’’, 1’’’’
2	2, 2’, 2’’, 2’’’
3	3, 3’

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	0	0’	0’’	1	1’	1’’	1’’’	1’’’’	2	2’	2’’	2’’’	3	3’

Any sorting algorithm is said to be ***stable*** if duplicate keys always remain in the same relative order that they were originally (such as 1, 1’, 1’’, 1’’’, 1’’’’).

- Counting sort is a stable sorting algorithm.
- Bin sort is a stable sorting algorithm.

Radix sort:

Assume all keys are integers in a bounded range $0 \ldots r^d - 1$

- r = radix (or base)
- d = number of digits, each digit is in range $0 \ldots r-1$

Examples:

- CWID are 8-digit numbers $\Rightarrow$ range $0 \ldots 10^8 - 1$
- SSN are 9-digit numbers $\Rightarrow$ range $0 \ldots 10^9 - 1$
- Unsigned 32-bit integers $\Rightarrow$ range $0 \ldots 2^{32} - 1$ or $0 \ldots 16^8 - 1$
- ASCII character strings of fixed length L (or padded with blanks or null up to maximum length L) $\Rightarrow$ range $0 \ldots 128^L - 1$

```
RadixSort (A[1…n], r, d) {          // each A[k] in range 0…r^d –1
      for (k=1; k<=d; k++)
            do any stable sort using the kth rightmost digit as key;
}
```

Analysis of radix sort:

If repeat either counting sort or bin sort d times,

then the total time for radix sort is $\theta(d(m+n))$,

which is $\theta(d(r+n))$ time because $m = r-1$.

Example: keys in range 0…9999, which is 0…$10^4 - 1$

A:	k=1:	k=2:	k=3:	k=4:
3579	3578	3568	3468	2468
3578	3568	3468	2468	2469
3569	3478	2568	3469	2478
3568	3468	2468	2469	2479
3479	2578	3569	3478	2568
3478	2568	3469	2478	2569
3469	2478	2569	3479	2578
3468	2468	2469	2479	2579
2579	3579	3578	3568	3468
2578	3569	3478	2568	3469
2569	3479	2578	3569	3478
2568	3469	2478	2569	3479
2479	2579	3579	3578	3568
2478	2569	3479	2578	3569
2469	2479	2579	3579	3578
2468	2469	2479	2579	3579

Why does radix sort work?

At end of k iterations, A has been sorted by rightmost k digits