Sorting in Linear Time

Can We Sort in Linear Time?

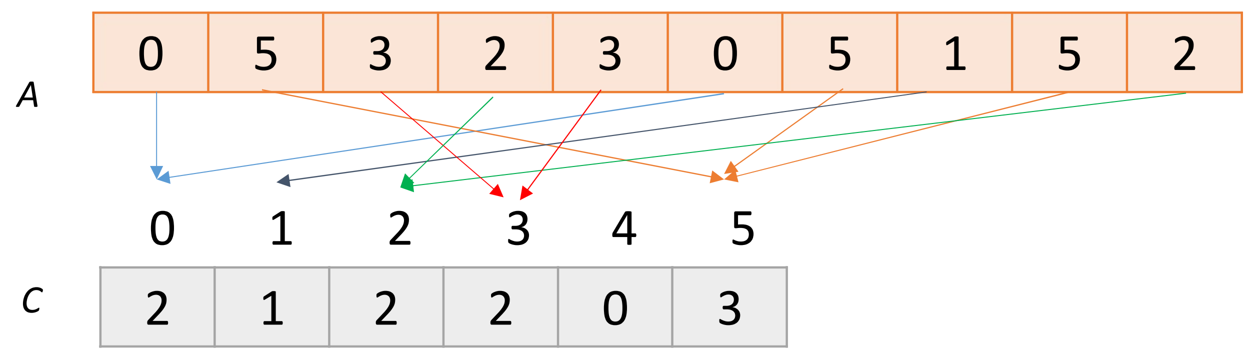
- Merge sort, heapsort and quicksort are all **comparison** sorts.
- Recall that any comparison sorts must make **O(n log n)** comparisons in worst cases.
- To sort faster than this lower bound, we try to sort **without** any comparisons.
- Examples of such algorithms : bucket Sort, **counting** sort, **radix** sort.

Facts About Counting Sort

- It is not a comparison sort.
- Restricts elements in range from 0 to ***K***.
- Runs in linear time when $\boldsymbol{K = O(N)}$.
- ☑ **Stable**
  - It is often used as a **subroutine** in radix sort.
- ☒ Does **NOT** sort in place.

Step 1: Counting

- We prepare an array *C* for counting the occurrences of each element in range 0 to *K*.
  - The length of the array required is *K*.
- We go through each element stored in the original array *A* and **calculate** the count, saving in *C*.



Step 2: Accumulation

- Compute the **accumulative** count on *C*, giving *C'*.
- The count in *C*'[*i*] is the number of elements that are **equal or smaller** than *i*.

C	2	1	2	2	0	3
	0	1	2	3	4	5

C'	2	3	5	7	7	10
	0	1	2	3	4	5

Step 3: Place Elements

- The last step is put the elements in *A* in the sorted order in a new array *B* using *C*'.
- We process each element in *A* in **reversed** order (we'll see why later).
- Lookup the position *j* from *C'* and copy the value of the element to ***B*[*j* – 1]**.
- **Copy contents** of *B* to *A* to finalize the sorting.

A

0	5	3	2	3	0	5	1	5	2

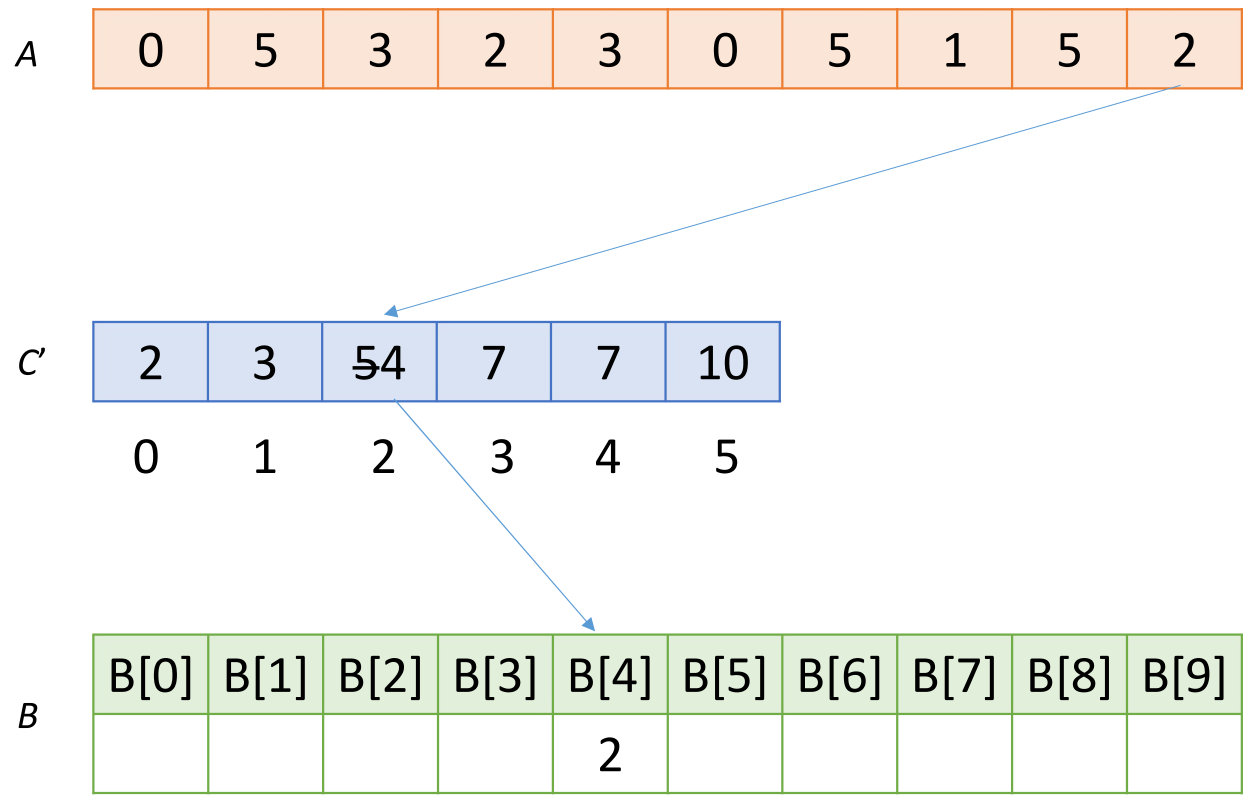
C'

2	3	5	7	7	10
0	1	2	3	4	5

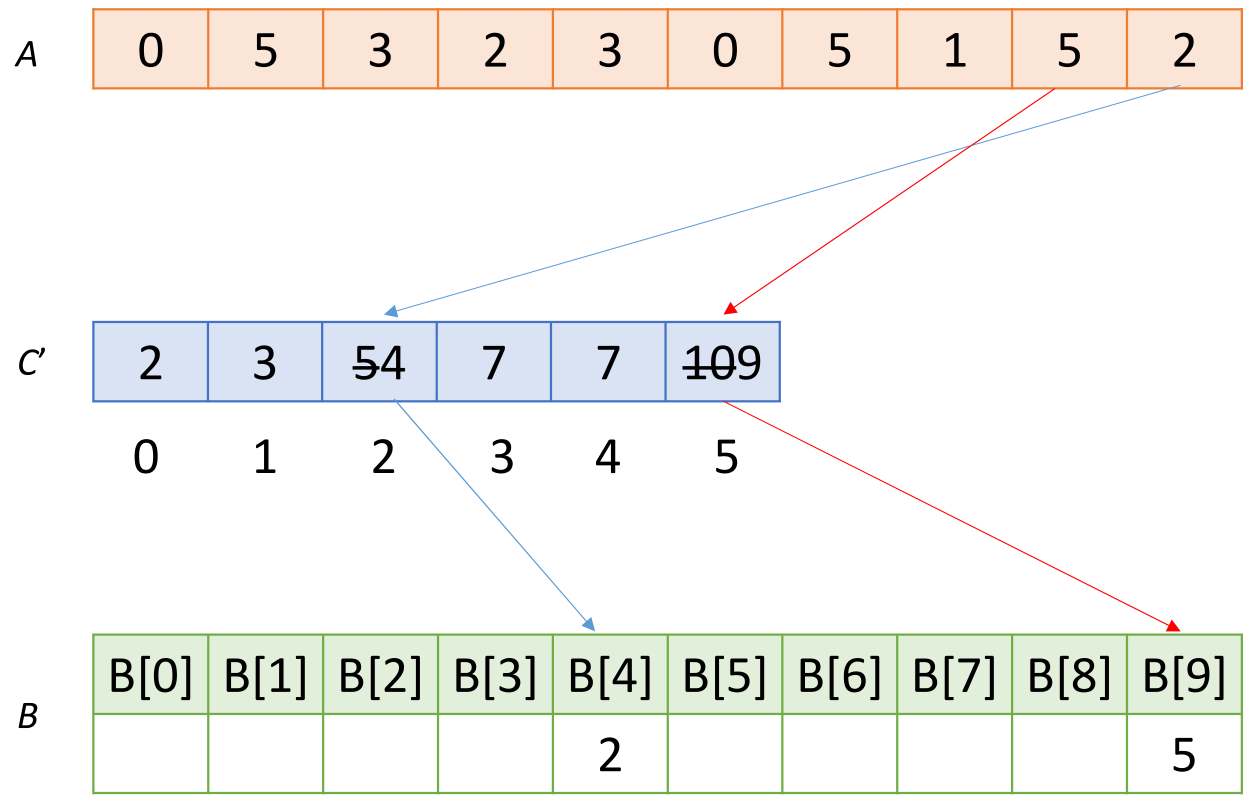
B

B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]
0	0	1	2	2	3	3	5	5	5

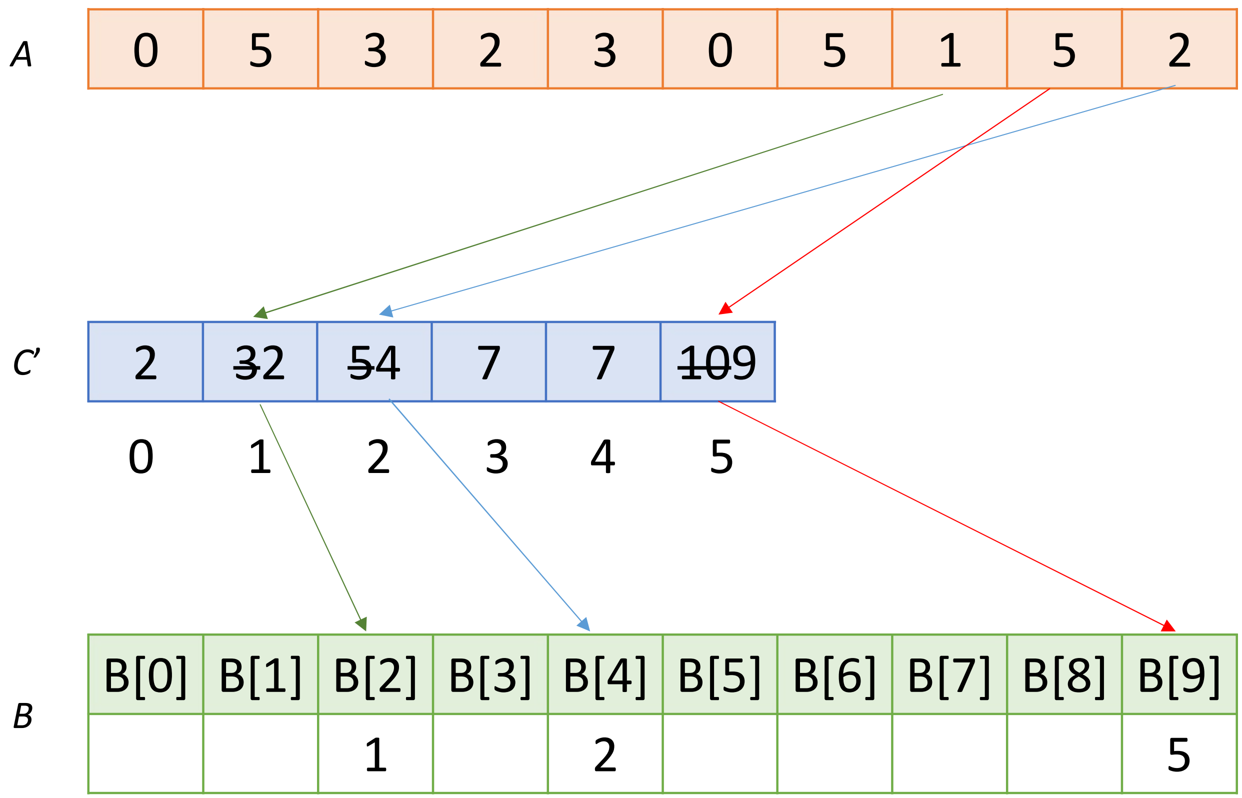
Step 3: Place Elements



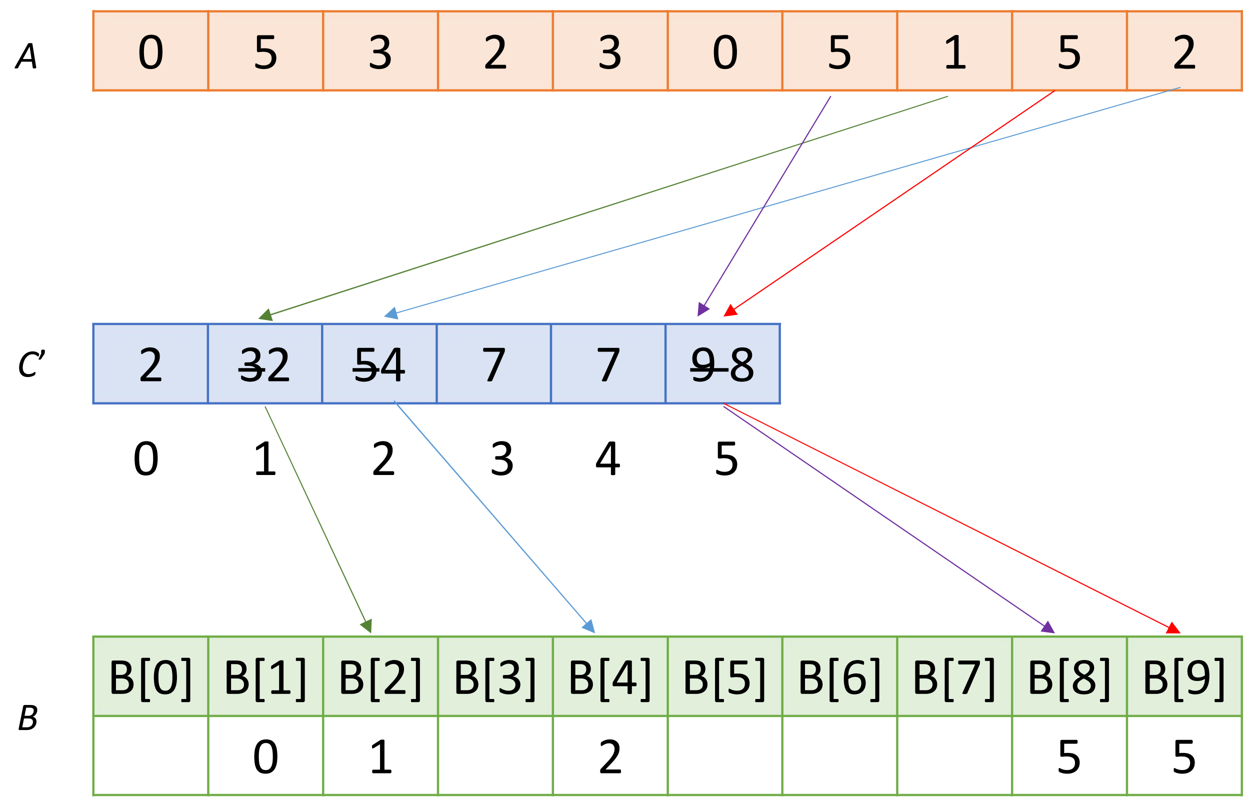
Step 3: Place Elements



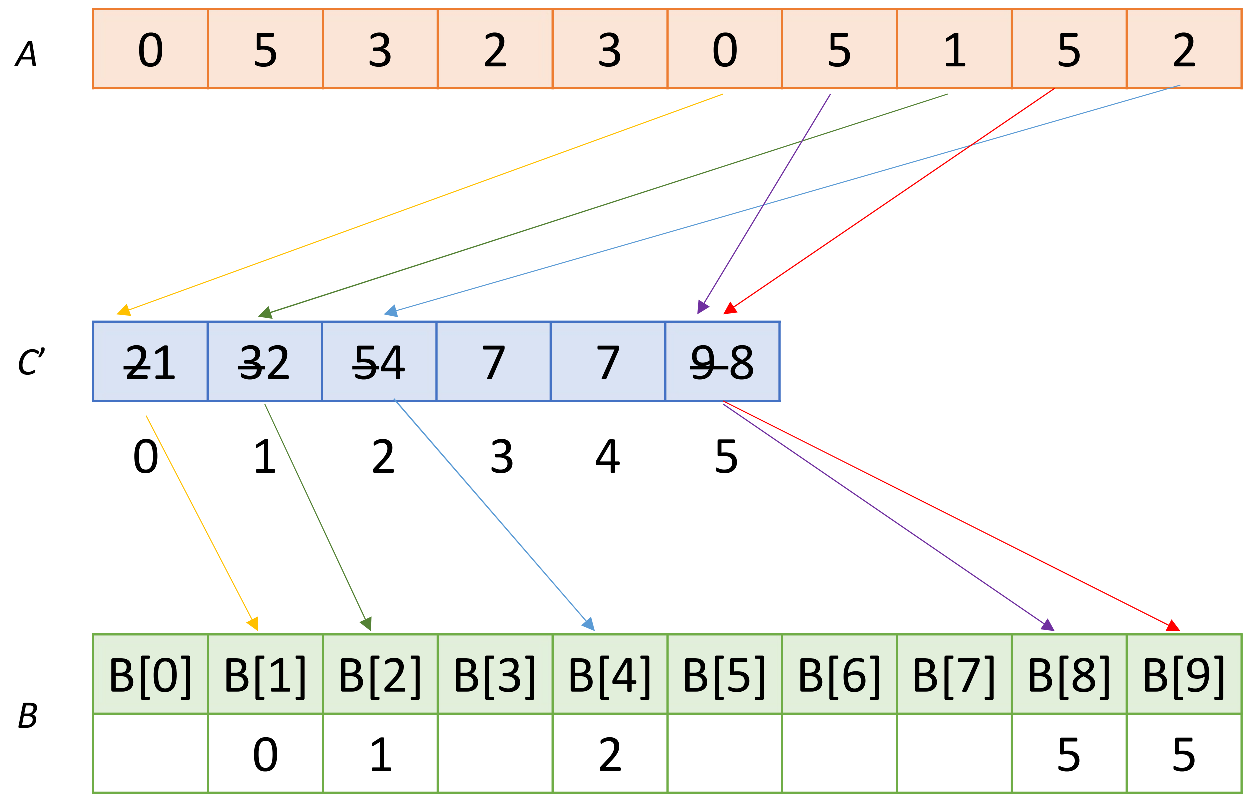
Step 3: Place Elements



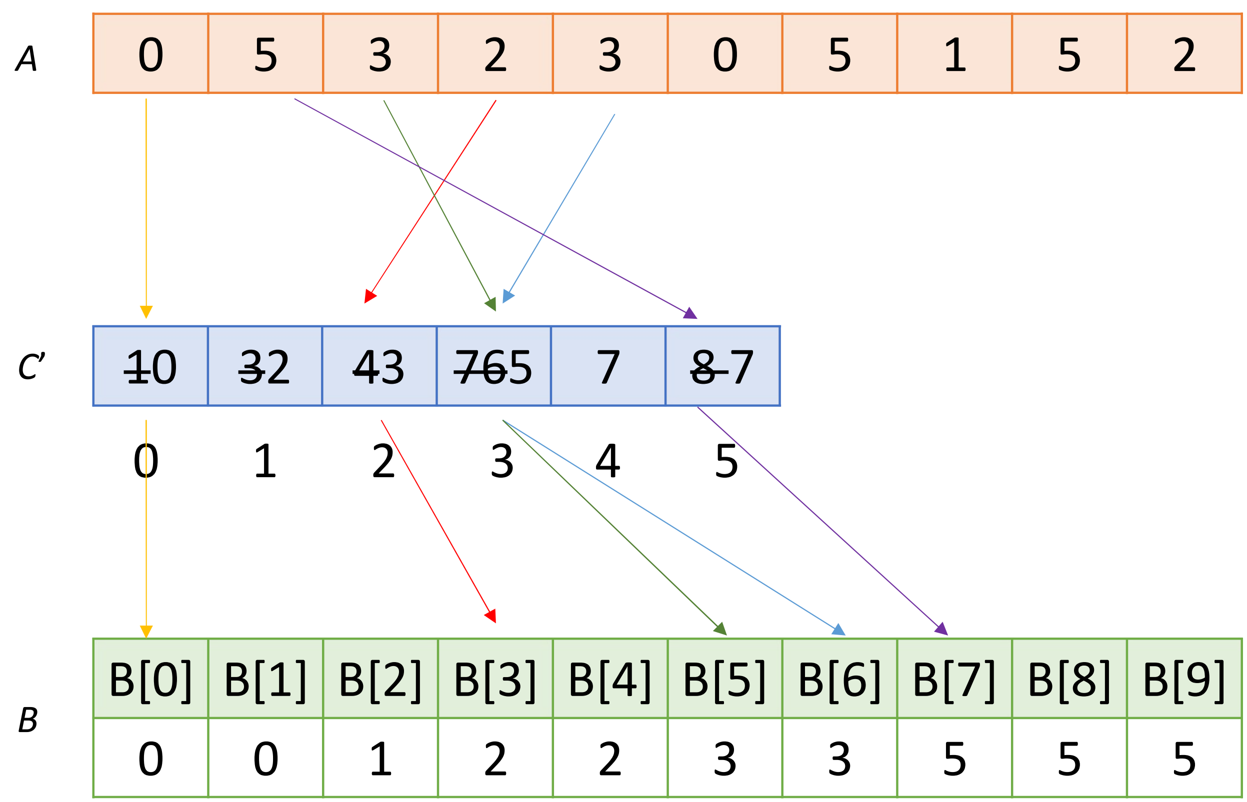
Step 3: Place Elements



Step 3: Place Elements



Step 3: Place Elements



Counting Sort: Code

- Predict the value of *K* or find it by checking the elements in *A*.
- Line 7-8: counting *O*(*N*); line 9-10: accumulation *O*(*K*)
- Line 11-14: place and sort *O*(*N*)

```
1 #define k 10
2
3 void csort(int n, int *a, int *b, int k){
4     int i;
5     int *c = (int *)calloc(k, sizeof(int));
6
7     for (i = 0; i < n; i++)
8         c[a[i]]++;
9     for (i = 1; i < k; i++)
10        c[i] = c[i] + c[i - 1];
11    for (i = n - 1; i >= 0; i--)
12        b[--c[a[i]]] = a[i];
13    free(c);
14 }
```

Step 1

Step 2

Step 3

Counting Sort: Code (2)

- We need a **driver** function to prepare *B* and copy *B* back to *A*.
- If you need to calculate *K*, you can put your code here.
- The **complexity** of counting sort is $O(N + K)$.
  - If $K = O(N)$, then $T(N) = O(N)$.

```c
void countingsort(int n, int *a){
    int i;
    int *b = (int *)malloc(n * sizeof(int));

    csort(n, a, b, K);

    for (i = 0; i < n; i++)
        a[i] = b[i];

    free(b);
}
```

Stability

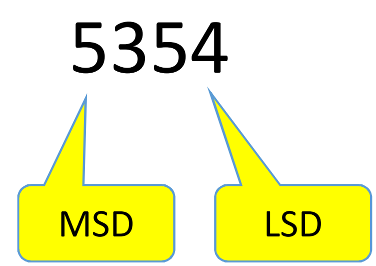
- Elements with same keys appear in the output array in the same (**relative**) order.
  - This is why we process the array *A* in reversed order.
- This is an **important** property when we use counting sort as subroutine in radix sort.

A	0_1	5_1	3_1	2_1	3_2	0_2	5_2	1	5_3	2_2
B	0_1	0_2	1	2_1	2_2	3_1	3_2	5_1	5_2	5_3

stable

```
for (i = n - 1; i >= 0; i--)
     b[--c[a[i]]] = a[i];
```

Radix Sort



- **Human** sorts numbers digit by digit and words alphabet by alphabet.
- **Idea**: most significant digits (MSD) **dominate** less significant digits (LSD).
- Don't forget that you still have to sort against every digit to get the sorting job done.
- **Question**: Sort MSD first or sort LSD first?

Imagine: you have a set of number cards, when you sort by the more significant digits first, you need to put aside the sorted piles. This creates problem when you program in a similar fashion.

Radix Sort (2)

- LSD radix sort solves the problem of card sorting on the least significant digit first.
- The cards in <u>bin 0</u> (smaller digits) **precedes** the cards in the <u>bin 1</u>.
- The entire deck is sorted again on the second-least significant digit.
- The process continues until the cards have been sorted on **all *D* digits**.
- **Important**: The digit sort must be **stable**.

Radix Sort: Example

Original	Digit 0	Digit 1	Digit 2
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort: Code

- For illustration (**ONLY**), we code radix sort on **base 10**.
- In practice, we would use a **group of bits** instead, so that the digit extraction can be done using bitwise operators.

```
#define R 10
#define D 3
...
void radixsort(int n, int *a){
    int i;

    for (i = 0; i < D; i++){
        csort(n, a, i);
        printf("i=%d\n", i);
        print_array(n, a);
    }
}
```

Radix Sort: Code (2)

- We modify the original counting sort so that we can consider a specified digit to be the key.

```c
void csort(int n, int *a, int d){
    int i, r = 1;
    int *b = (int *)malloc(n * sizeof(int));
    int *c = (int *)calloc(R, sizeof(int));

    for (i = 0; i < d; i++)
        r *= R; // calculate R^d
    for (i = 0; i < n; i++)
        c[(a[i] / r) % R]++; // counting
    for (i = 1; i < R; i++)
        c[i] = c[i] + c[i - 1];
    for (i = n - 1; i >= 0; i--)
        b[--c[(a[i] / r) % R]] = a[i];
    for (i = 0; i < n; i++)
        a[i] = b[i];

    free(b); free(c);
}
```

Step 1

Step 2

Step 3

```c
#define k 10

void csort(int n, int *a, int *b, int k){
    int i;
    int *c = (int *)calloc(k, sizeof(int));

    for (i = 0; i < n; i++)
        c[a[i]]++;
    for (i = 1; i < k; i++)
        c[i] = c[i] + c[i - 1];
    for (i = n - 1; i >= 0; i--)
        b[--c[a[i]]] = a[i];
    free(c);
}
```

Original

Radix Sort: Code (2)

- We modify the original counting sort so that we can consider a specified digit to be the key.

```
void csort(int n, int *a, int d){
    int i, r = 1;
    int *b = (int *)malloc(n * sizeof(int));
    int *c = (int *)calloc(R, sizeof(int));

    for (i = 0; i < d; i++)
        r *= R; // calculate R^d
    for (i = 0; i < n; i++)
        c[(a[i] / r) % R]++; // counting
    for (i = 1; i < R; i++)
        c[i] = c[i] + c[i - 1];
    for (i = n - 1; i >= 0; i--)
        b[--c[(a[i] / r) % R]] = a[i];
    for (i = 0; i < n; i++)
        a[i] = b[i];

    free(b); free(c);
}
```

Step 1

Step 2

Step 3

Sorting using the i^{th} digit

d = 2, R = 10, r = 100

a[i]= 720, r = 100, R = 10
→ (a[i] / r) % R = 7

PROOF

Radix Sort: Correctness

Suppose the list of numbers L_D are sorted on digits D, D - 1, ..., 1, then we sort L_D using a stable sorting algorithm on digit D + 1.

Consider two numbers a and b on L_D with a precedes b, then we know the last d digits of a is less than or equal to that of b (by assumption).

Consider the (D + 1)-digit of a and b,
If $a_{D+1} > b_{D+1}$, after this round, a will be placed after b.
If $a_{D+1} = b_{D+1}$, after this round, a precedes b (by stability).
If $a_{D+1} < b_{D+1}$, after this round, a precedes b (by D + 1 digit)

Therefore, L_{D+1} is sorted on digit D + 1, D, ..., 1.

By induction, radix sort is correct.

Radix Sort: Analysis

- Given N D-digits number in which each digit can take on up to K possible values, radix sort correctly sorts these numbers in $O(D(N + K))$.
  - There are D counting sorts which each takes $O(N + K)$.
- However, in general, it is hard to **bound** the numbers by the number of digits.
- It is easier to express in terms of **bits** as we use binary representation for numbers in computers.

Given N B-bits numbers and any positive integer $R \leq B$, radix sort sorts in $O((B/R)(N + 2^R))$.
Example: 32-bit word has 4 8-bit digits.
$B = 32$, $R = 8$, $K = 2^R - 1 = 255$, and $D = B/R = 4$.

11010110	00101001	00111000	11110101

Summary

- **Counting** sort : sort by counting frequencies of the keys. Stable but does not sort in-place. $\boldsymbol{\Theta(N + K)}$
- **Radix** sort: execute stable sort digit by digit. Begin with least significant digit requires little memory overheads. $\boldsymbol{\Theta((B/R)(N + 2^R))}$