CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

Sorting algorithms

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
  - bubble sort, selection sort, insertion sort, quick sort, heap sort, …
- Sorting is usually done with multiple rounds
  - Simple sorting algorithms run in O(N^2) time. Some uses O(nlog(n)) time. Best algorithms use O(n) time.
- Conventional sorting algorithms: https://www.toptal.com/developers/sorting-algorithms
- We discuss sorting values in "ascending" order in the class.
  - it is not difficult to figure out how to change the order to "descending".

Bucket sort

To sort N integer values within a range of (L, H)

- If we use *H-L+1 buckets*, one for each possible value, sorting is done by simply putting each integer into the corresponding bucket.
  - Fast, especially when N is large and range is small.
  - Too expensive, especially when N is small and range is large. E.g., sorting 100 unsigned integers needs 2^{32} buckets.
- Solution: use fewer buckets to find a good trade-off between N and range.
  - Each bucket serves the values within a smaller range.
    - E.g., two buckets, one for range (L, M) and one for range (M, H), M=(H+L)/2.
  - Put values into buckets.
  - Sort the values in each bucket.
    - Apply bucket sort recursively and/or apply other sorting algorithm when the values in a bucket are not many.

Radix Sort: bucket sort on every bit

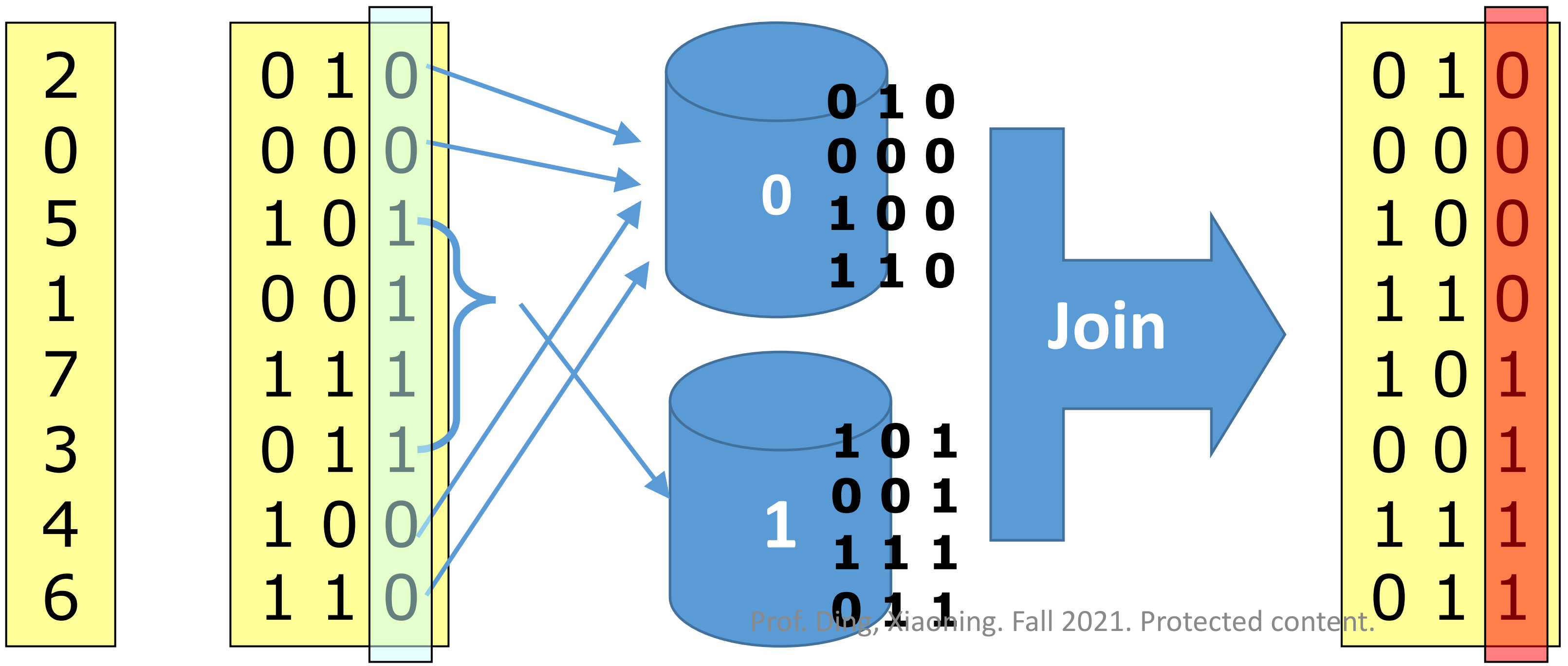
- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7x10^{308} , 1.7x10^{308})
- Solution(radix sort):
    - Sort the binary raw data
    - apply bucket sort on every bit, from least significant bit to most significant bit.

2	0 1 0
0	0 0 0
5	1 0 1
1	0 0 1
7	1 1 1
3	0 1 1
4	1 0 0
6	1 1 0

Use two buckets.

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7x10^{308} , 1.7x10^{308})
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



2	0 1 0	0 1 0
0	0 0 0	0 0 0
5	1 0 1	1 0 0
1	0 0 1	1 1 0
7	1 1 1	1 0 1
3	0 1 1	0 0 1
4	1 0 0	1 1 1
6	1 1 0	0 1 1

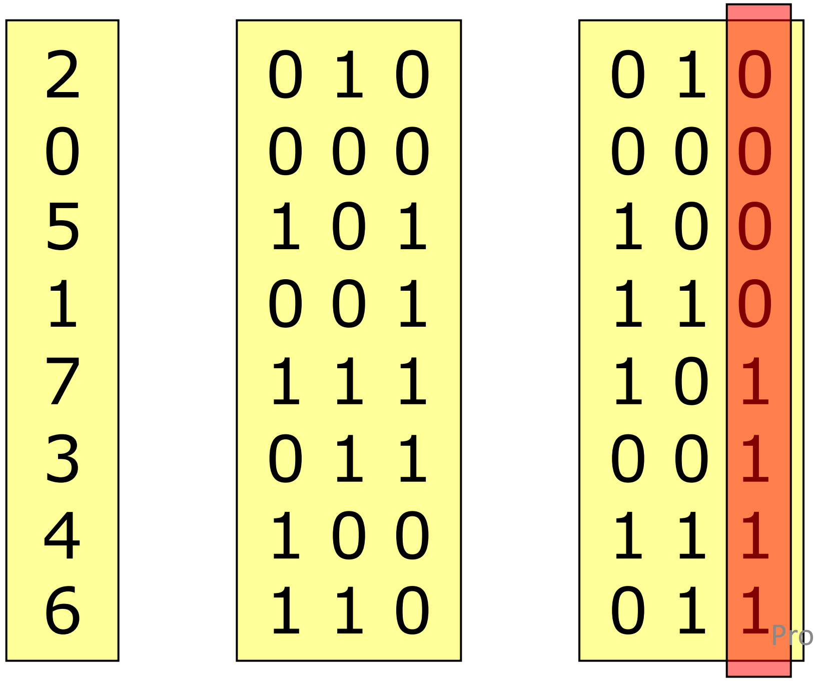
0 1 0
0 0 0
1 0 0
1 1 0

1 0 1
0 0 1
1 1 1
0 1 1

Last bits are sorted

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7×10^{308} , 1.7×10^{308})
- Solution(radix sort):
    - Sort the binary raw data
    - apply bucket sort on every bit, from least significant bit to most significant bit.



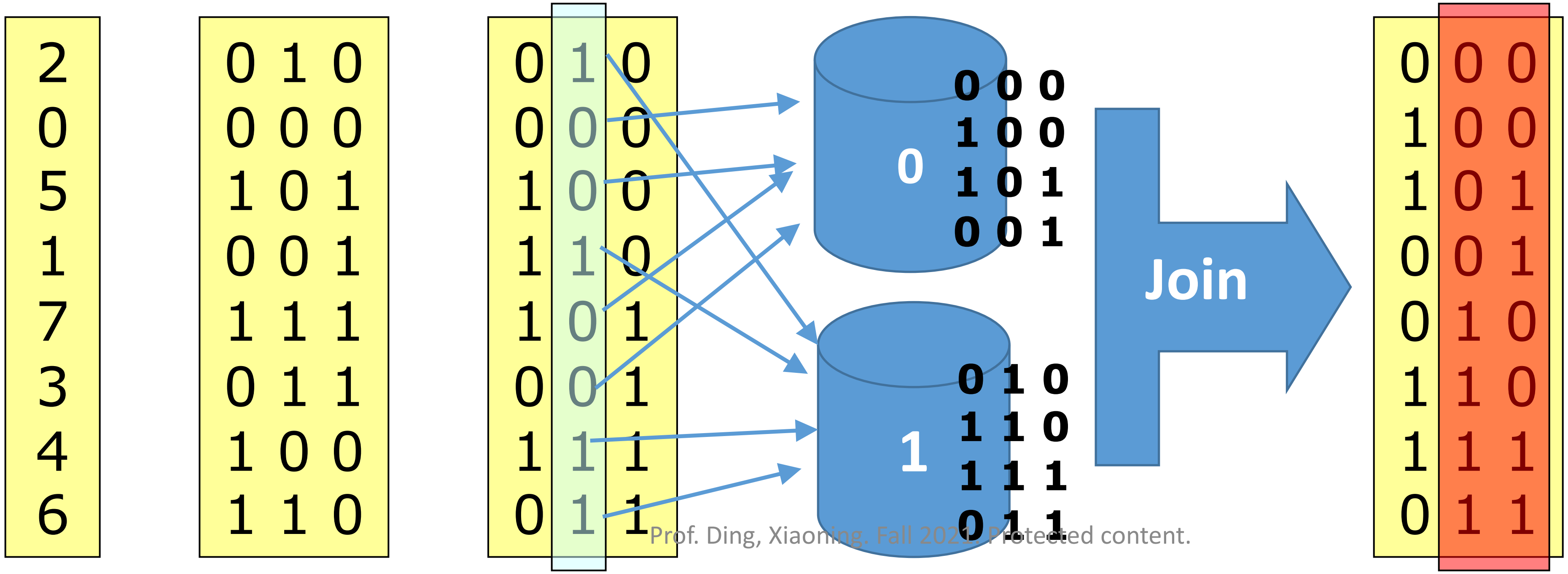
2	0 1 0	0 1 0
0	0 0 0	0 0 0
5	1 0 1	1 0 0
1	0 0 1	1 1 0
7	1 1 1	1 0 1
3	0 1 1	0 0 1
4	1 0 0	1 1 1
6	1 1 0	0 1 1

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is $(-1.7\text{x}10^{308}, 1.7\text{x}10^{308})$
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

Values with last bit =0 enter buckets first.

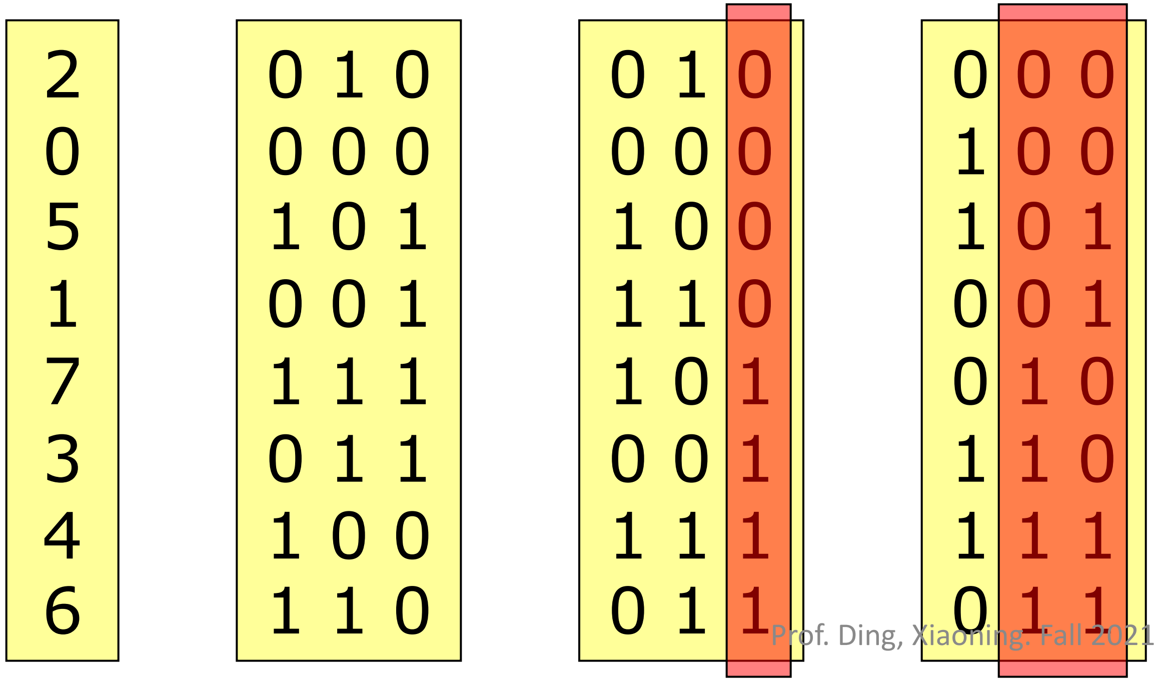
Last 2 bits are sorted in each bucket.



Last 2 bits are sorted after the join.

Radix Sort: bucket sort on every digit/bit

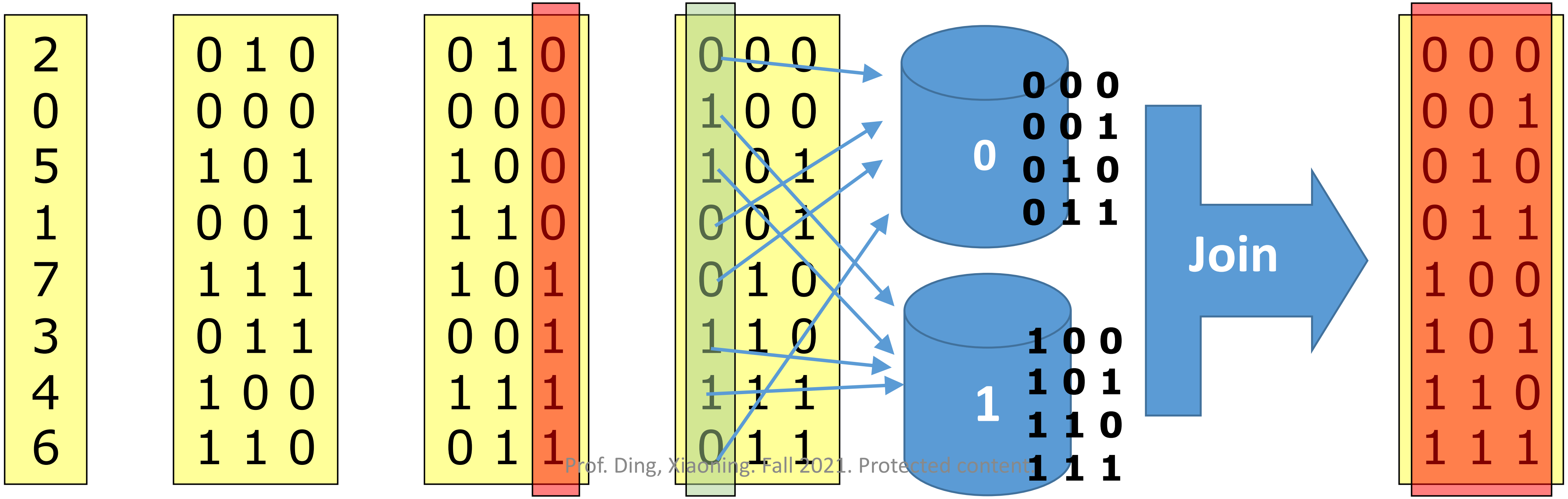
- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7x10^{308} , 1.7x10^{308})
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



2	0 1 0	0 1 0	0 0 0
0	0 0 0	0 0 0	1 0 0
5	1 0 1	1 0 0	1 0 1
1	0 0 1	1 1 0	0 0 1
7	1 1 1	1 0 1	0 1 0
3	0 1 1	0 0 1	1 1 0
4	1 0 0	1 1 1	1 1 1
6	1 1 0	0 1 1	0 1 1

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7×10^{308} , 1.7×10^{308})
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.



2	0 1 0	0 1 0	0 0 0	0 0 0
0	0 0 0	0 0 0	1 0 0	0 0 1
5	1 0 1	1 0 0	1 0 1	0 1 0
1	0 0 1	1 1 0	0 0 1	0 1 1
7	1 1 1	1 0 1	0 1 0	1 0 0
3	0 1 1	0 0 1	1 1 0	1 0 1
4	1 0 0	1 1 1	1 1 1	1 1 0
6	1 1 0	0 1 1	0 1 1	1 1 1

0

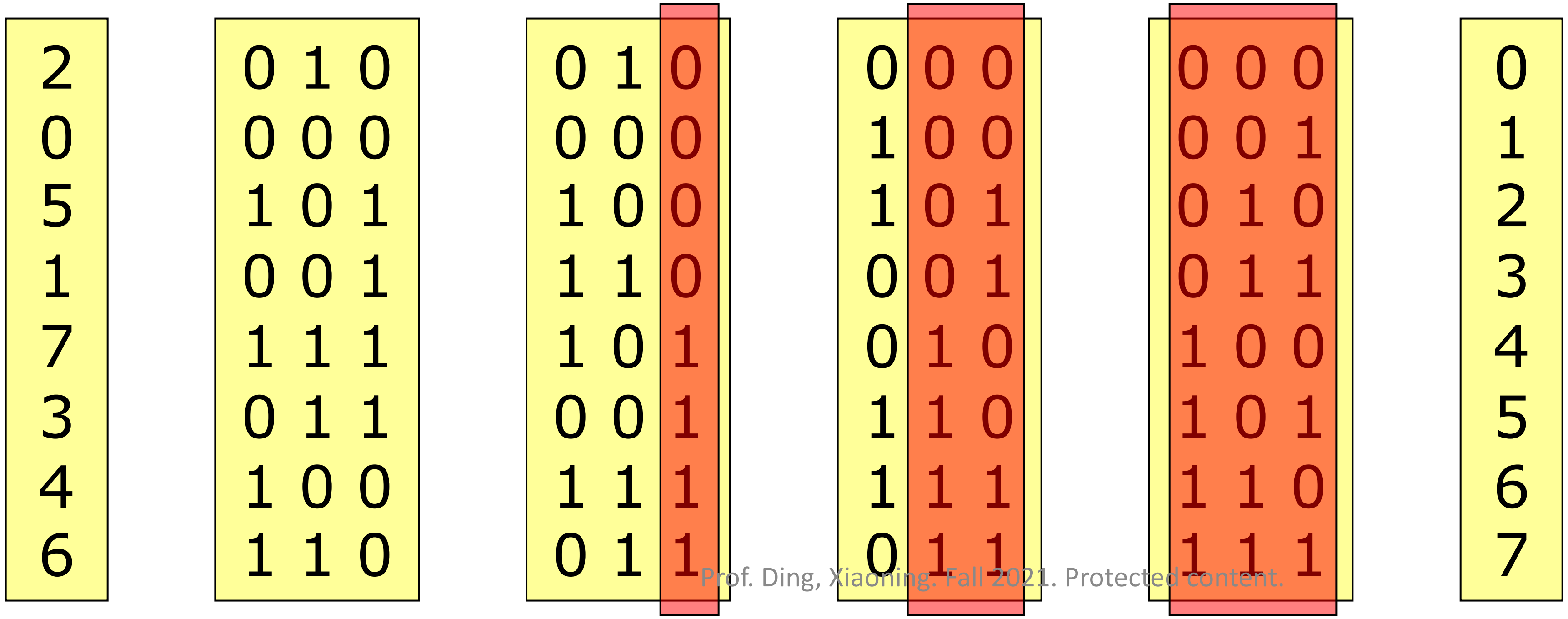
0 0 0
0 0 1
0 1 0
0 1 1

1

1 0 0
1 0 1
1 1 0
1 1 1

Radix Sort: bucket sort on every digit/bit

- Problem with bucket sort: It is difficult to choose number of buckets, especially the range can be huge. E.g., when sorting double precision values, the range is (-1.7x10^{308} , 1.7x10^{308})
- Solution(radix sort):
  - Sort the binary raw data
  - apply bucket sort on every bit, from least significant bit to most significant bit.

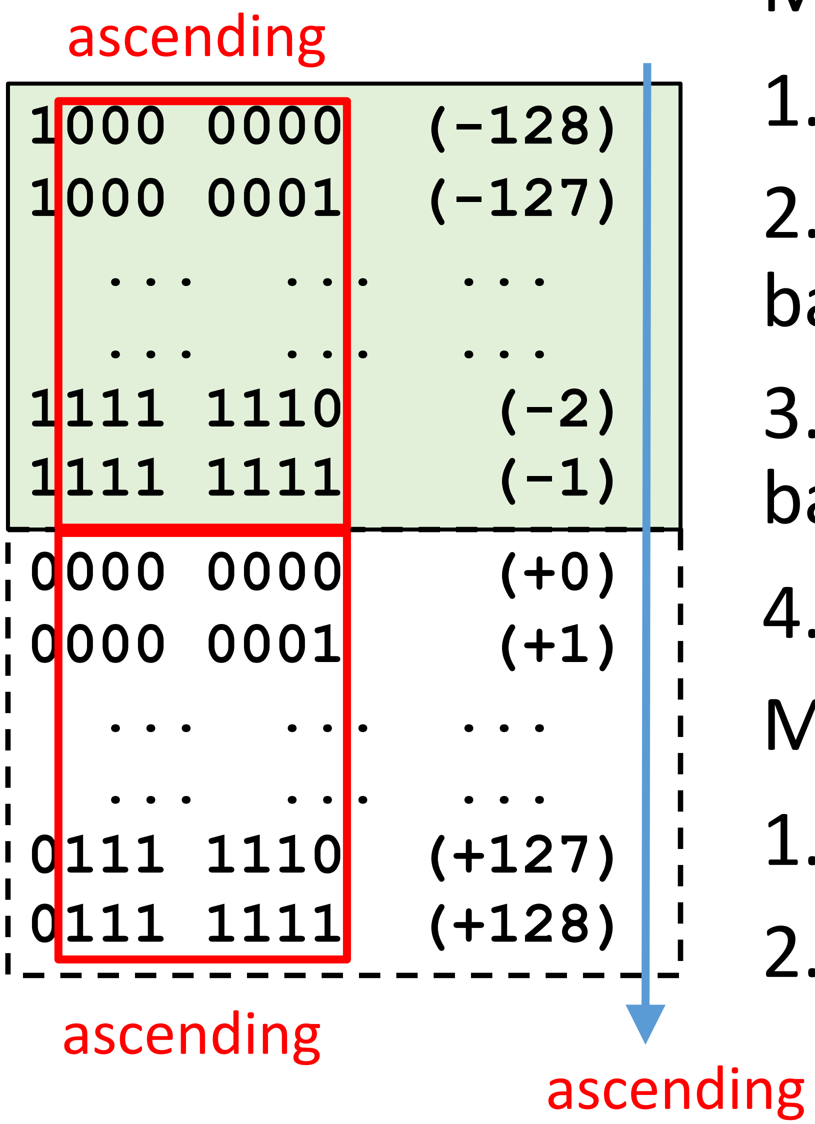


2	0 1 0	0 1 0	0 0 0	0 0 0	0
0	0 0 0	0 0 0	1 0 0	0 0 1	1
5	1 0 1	1 0 0	1 0 1	0 1 0	2
1	0 0 1	1 1 0	0 0 1	0 1 1	3
7	1 1 1	1 0 1	0 1 0	1 0 0	4
3	0 1 1	0 0 1	1 1 0	1 0 1	5
4	1 0 0	1 1 1	1 1 1	1 1 0	6
6	1 1 0	0 1 1	0 1 1	1 1 1	7

Radix-sort unsigned integers

```
radix_sort(A, n, k) {   /* A: array; n: number of items; */
    /* k: number of bits in each item (32 for unsigned int) */
        create two buckets  (buckets can be arrays or lists)
        for (d = 0; d <k; d++) {
            /* sort A using d-th bit as the key. */
            for (i = 0; i<n; i++) {
                if the d-th bit (from right) of A[i] is 0
                    add A[i] to bucket #0
                else
                    add A[i] to bucket #1
            }
            A = Join the buckets
      }
}
```

Radix-sort **integers with signs**



ascending

```
1000 0000   (-128)
1000 0001   (-127)
  ...   ...   ...
  ...   ...   ...
1111 1110     (-2)
1111 1111     (-1)
0000 0000     (+0)
0000 0001     (+1)
  ...   ...   ...
  ...   ...   ...
0111 1110   (+127)
0111 1111   (+128)
```

ascending

ascending

Method1

1. Separate positive numbers and negative numbers.

2. Radix-sort positive numbers in ascending order based on low 31 bits

3. Radix-sort negative numbers in ascending order based on low 31 bits

4. Join positive numbers and negative numbers.

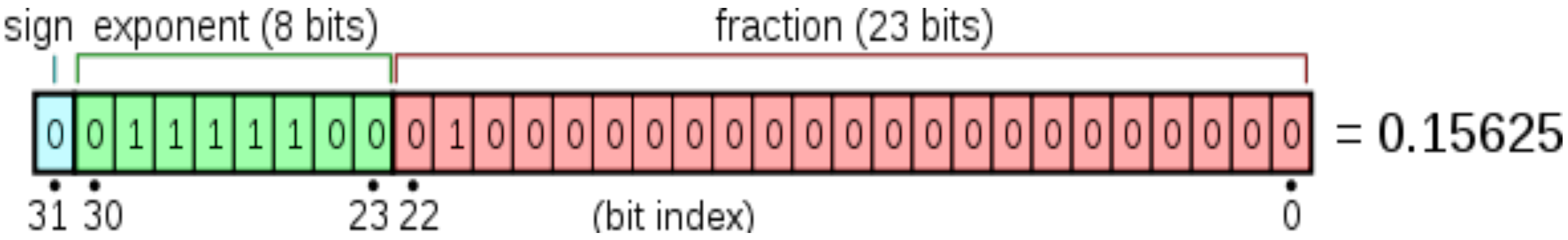
Method2:

1. Sort all values as if they were unsigned
2. In the last round joining the two bucket, put values from bucket 1 before values from bucket 0.

Radix Sort IEEE Floats/Doubles

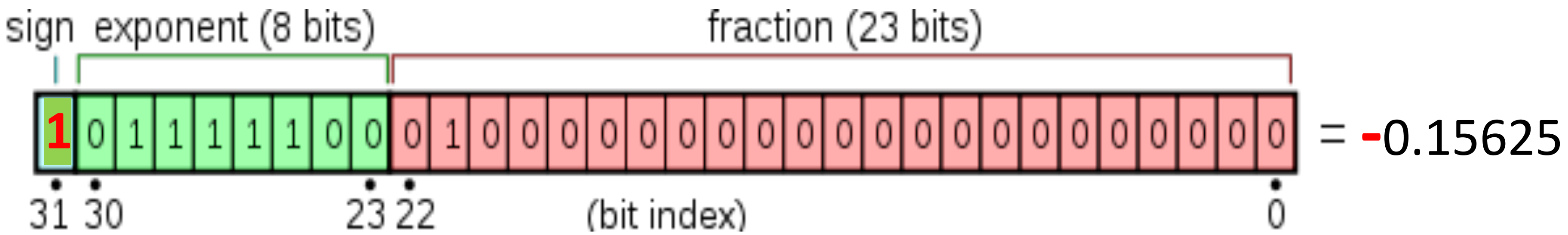
- It is straightforward to use radix sort on integers.
- Some people say you can’t Radix Sort real numbers.
- You can Radix Sort real numbers, in most representations
- We do IEEE floats/doubles, which are used in C/C++.

Observations



- Non-negative float point numbers
  - Larger value in a digit means larger number
    - e.g., 0 01111100 **0**10000… =0.15625 (the value above);
      0 01111100 **1**10000… =0.21875
    - When joining buckets, bucket with a **smaller** digit value comes first to achieve “ascending” order
  - Values are more determined by a higher digit than any lower digits
    - e.g., 0 01111100 **01**0000… =0.15625 (the value above);
      0 01111100 **10**0000… =0.18750
    - Exponent always more significant than significand
    - e.g., 0 0111110**0** 0**1**0000… =0.15625 (the value above);
      0 0111110**1** 0**0**0000… =0.25
    - Repeat the rounds from the least significant bit to most significant bit

Observations



- Negative float point numbers
  - Larger value in a digit means **smaller** number
    - e.g., 1 01111100 010000… = -0.15625 (the value above);
      1 01111100 110000… = -0.21875 (smaller)
    - When joining buckets, the bucket with a **larger** digit value comes first to achieve “ascending” order
  - Values are more determined by a higher digit than any lower digits
    - e.g., 1 01111100 010000… =-0.15625 (the value above);
      1 01111100 100000… =-0.18750 (smaller)
    - Exponent always more significant than significand
    - e.g., 0 01111100 010000… = -0.15625 (the value above);
      0 01111101 000000… = -0.25 (smaller)
    - Repeat the rounds from the least significant bit to most significant bit

What if there are non-negative numbers and negative numbers?

- Method 1: sort non-negative numbers and negative numbers separately
  - Pay attention to the way of joining the buckets
  - Put all non-negative numbers after negative numbers
- Method 2: what if we sort non-negative and negative numbers together in the same way?
  - Step 1: sort all the numbers as if they were all **unsigned integers**.
    - Join the buckets in the same way (smaller digits first) for all the numbers
  - As illustrated later, when step 1 is finished,
    - all the negative numbers come after non-negative numbers
    - non-negative numbers are **ascending**
    - negative numbers are **descending**
  - Fix the order by re-organizing the numbers.
    - Flip the order of negative #s, and move negative #s before non-negative #s.

1056964608	0.50
1069547520	1.50
1075838976	2.50
1080033280	3.50
1083179008	4.50
1085276160	5.50
1087373312	6.50
1089470464	7.50
1091043328	8.50
3204448256	-0.50
3217031168	-1.50
3223322624	-2.50
3227516928	-3.50
3230662656	-4.50
3232759808	-5.50
3234856960	-6.50
3236954112	-7.50
3238526976	-8.50
3239575552	-9.50
3240624128	-10.50

If you sort -10.5, -9.5, …, 7.5, 8.5 as you do for unsigned int

a

```
#include <stdlib.h>

main(){
    int i;
    float value, f[20];
 /* typecasting w/ a pointer */
    unsigned int *p =
                (unsigned int *) f;

    value = -10.5;
    for( i = 0; i < 20; i++) {
        f[i] = value;
        value = value + 1;
    } /*-10.5 … 8.5 -> f */

    for( i = 0; i < 20; i++)
        printf("%.2f\t%u\n",
                f[i], p[i]);
}
```

-10.50	3240624128
-9.50	3239575552
-8.50	3238526976
-7.50	3236954112
-6.50	3234856960
-5.50	3232759808
-4.50	3230662656
-3.50	3227516928
-2.50	3223322624
-1.50	3217031168
-0.50	3204448256
0.50	1056964608
1.50	1069547520
2.50	1075838976
3.50	1080033280
4.50	1083179008
5.50	1085276160
6.50	1087373312
7.50	1089470464
8.50	1091043328

Numbers sorted properly

Radix-sort float point numbers (method 1)

You need to know how to extract bits correctly.

1. Radix-sort all numbers based on all 32 bits

2. Reverse the order of negative numbers

3. Put all negative numbers before positive numbers.

Radix-sort float point numbers (method 2)

Similar to method 1 of radix sorting a mixture of positive and negative integers. But you need to know how to extract bits correctly.

1. Separate positive numbers and negative numbers.
2. Radix-sort positive numbers in ascending order based on low 31 bits
3. Radix-sort negative numbers in descending order based on low 31 bits
4. Join positive numbers and negative numbers.

Other info about radix sort

- Radix sort was first used in 1890 U.S. census by Hollerith
- Used to sort numbers or texts
- Very efficient when sorting a large number of elements
  - O(M*N). M: length of each elements; N: number of elements
- Fixed size buckets make it consume more space than other sorting algorithms
  - E.g., bubble sort is in-place soring.

Radix sort for any radix values

- Radix = “The base of a number system” (Webster’s dictionary)
- Radix is another term of “base” : number of unique digits, including the digit zero, used to represent numbers
- Radix of numbers:
  - Binary numbers have a radix of 2
  - decimals have a radix of 10
  - hexadecimals have a radix of 16.
- Radix of texts:
  - 26 if only capital letters are considered
  - 36 if capital letters and decimal digits are considered
  - 62 for capital letters + small letters + decimal digits

Radix sort of decimal numbers

Values to be sorted 126, 328, 636, 341, 416, 131, 328

- Sort based on on lower digit:

341, 131, 126, 636, 416, 328, 328

- Sort the result based on next-higher digit:

416, 126, 328, 328, 131, 636, 341

- Sort the result based on highest digit:

126, 131, 328, 328, 341, 416, 636

RadixSorting Strings

- Single characters can be Bucket-Sorted
- Break strings into characters
- Append NULLs to short strings
- Start from the last character, end with the first character.

	5th pass	4th pass	3rd pass	2nd pass	1st pass
String 1	z	i	p	p	y
String 2	z	a	p		
String 3	a	n	t	s	
String 4	f	l	a	p	s

NULLs are treated as character with ASCII code equal to 0

Radix and bit masks when sorting binary data

- Values to be sorted: 126, 328, 636, 341, 416, 131, 328
    - Binary numbers: (0 001 111 110, 0 101 001 000, 1 001 111 100, 0 101 010 101,
      0 110 100 000, 0 010 000 011, 0 101 001 000)
    - Octal numbers: (0176, 0510, 1174, 0525, 0640, 0203, 0510)
    - Hexadecimal numbers: (07E, 148, 27C, 1A0, 083, 148)
- Selection is a trade-off between time complexity and space complexity.
- For the above examples
    - how many buckets are needed?
    - how many passes must be made?
    - How to generate masks and how to determine bucket index?

Mask=0xF<<(pass*4)

Bucket_index = (Value & Mask)>>(pass*4)

Radix sort algorithm in a general form

```
radix_sort(A, n, k) {
    /* A: array; n: number of items; k: number of digits */
    create buckets  (buckets can be arrays or lists)
     for (d = 0; d <k; d++) {
         /* sort A using digit position d as the key. */
         for (i = 0; i<n; i++) {
                 p = the d-th digit  (from right) of A[i]
                 Add A[i] to bucket p
         }
        A = Join the buckets
    }
}
```

Order is important

1. Control the passes to move from the least significant part to the most significant part.
2. Enforce the same order when selecting items to move buckets, organizing the items in buckets, and joining the buckets.

Not magic. It provably works.

- Elements: N-digit numbers, base B
- Claim: after i^{th} sorting, least significant i digits are sorted.
    - e.g. B=2, i=2, elements are 101 and 011. 101 comes before 011 for last 2 bits.
- Proof using induction:
    - base case: i=1. 1 digit is sorted (that wasn't hard!)
    - Induction step
        - assume for i, prove for $i+1$.
        - consider two numbers: X, Y. Say X_i is i^{th} digit of X (from the right)
            - Values are more determined by a higher digit than any lower digits, i.e.,
            - $X_{i+1} > Y_{i+1}$ then $i+1^{th}$ sorting will put them in order
            - $X_{i+1} < Y_{i+1}$, same thing
            - $X_{i+1} = Y_{i+1}$, order depends on last i digits. Induction hypothesis says already sorted for these digits.