Lecture 14

Radix Sort

Jianchen Shan

Department of Computer Science

Hofstra University

Radix and Radix Sort

- 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 and 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
- May use more space than other sorting algorithms
  - E.g., bubble sort is in-place soring.
- Basic idea: Bucket sort on each digit, from least significant digit to most significant digit.

Bucket Sort in Radix Sort

- Use a bucket array of size R for a radix of R
- Put elements into the correct bucket in the array
- R = 5; unique digits (0,1,2,3,4); list = (0,1,3,4,3,2,1,1,0,4,0)

Buckets	
= 0	0,0,0
= 1	1,1,1
= 2	2
= 3	3,3
= 4	4,4

Sorted list:
0,0,0,1,1,1,2,3,3,4,4

Radix Sort: bucket sort on every digit/bit

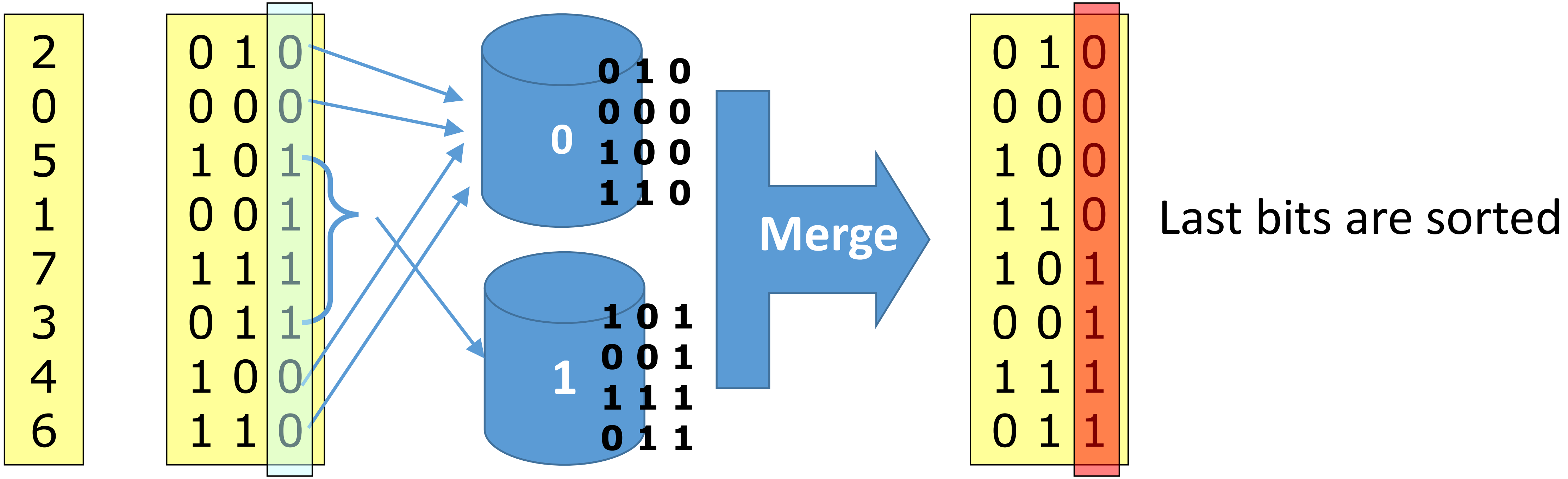
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit

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

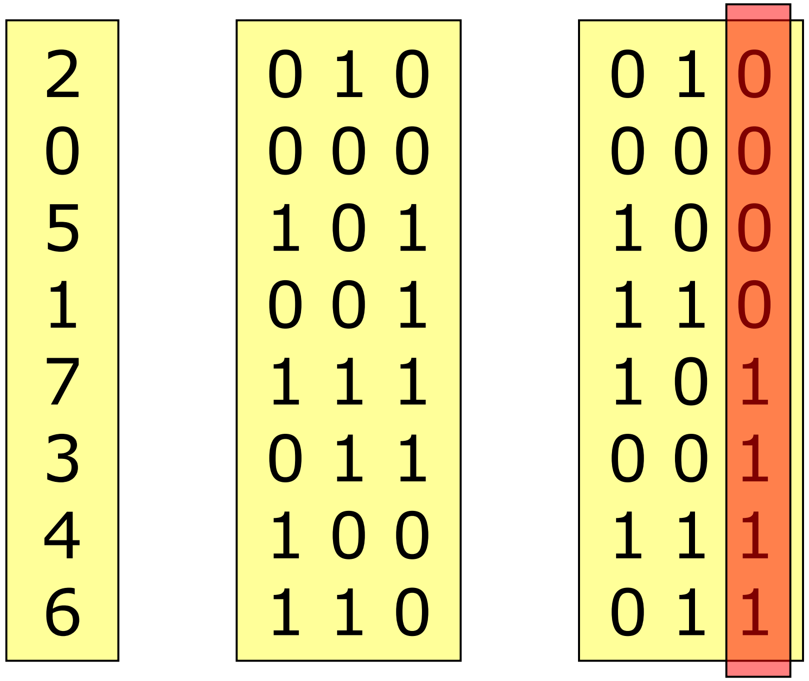
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
    - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit



Last bits are sorted

Radix Sort: bucket sort on every digit/bit

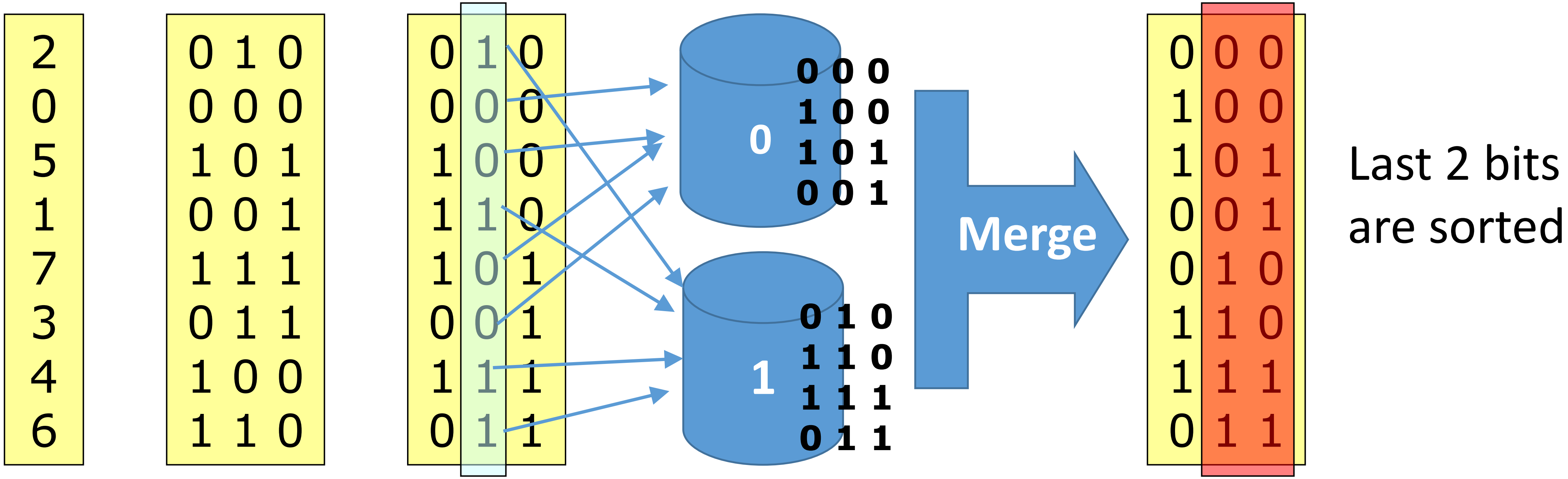
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit



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

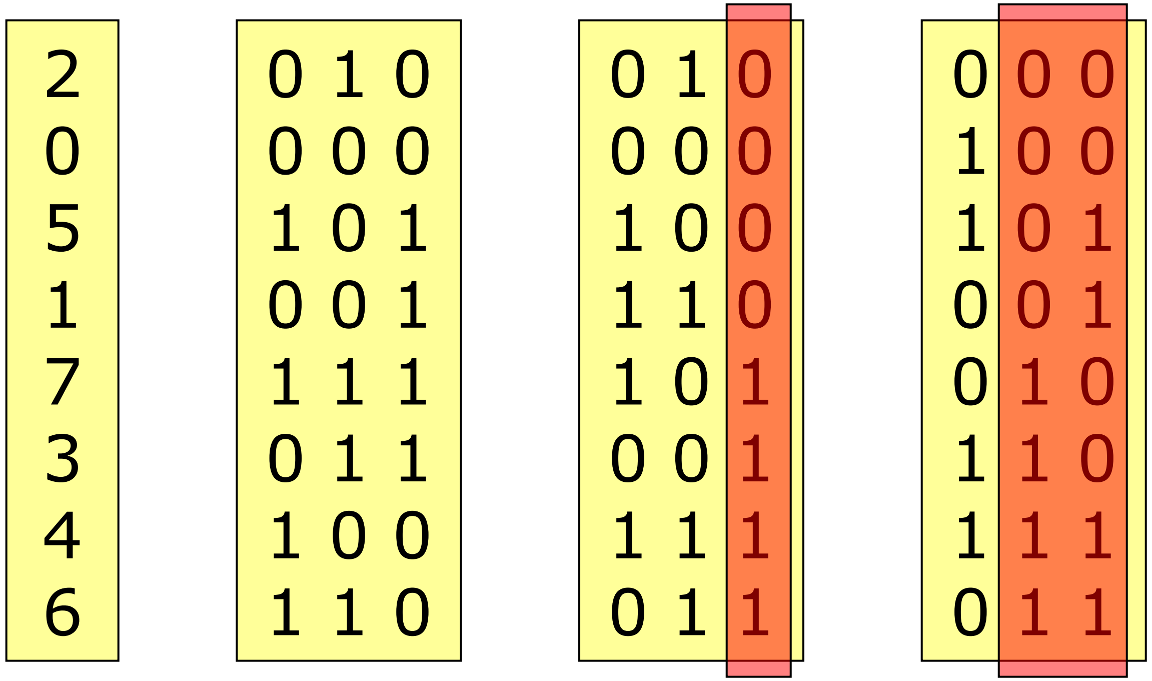
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit



Last 2 bits are sorted

Radix Sort: bucket sort on every digit/bit

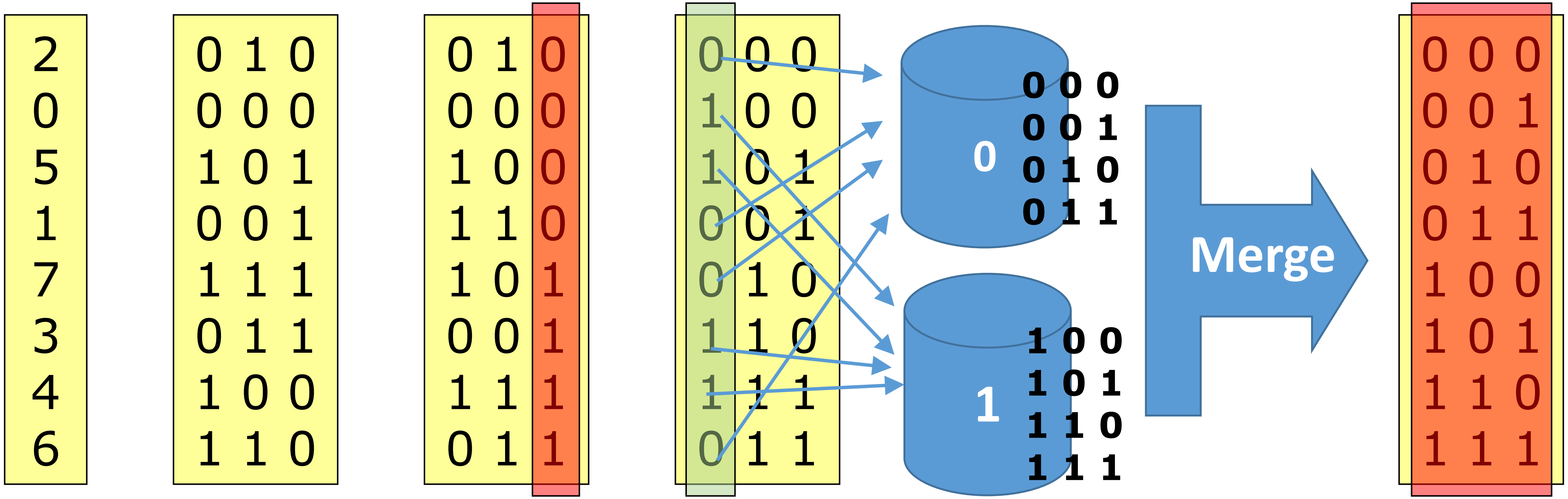
- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit



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

- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/bit



2 0 5 1 7 3 4 6

010 000 101 001 111 011 100 110

010 000 100 110 101 001 111 011

000 100 101 001 010 110 111 011

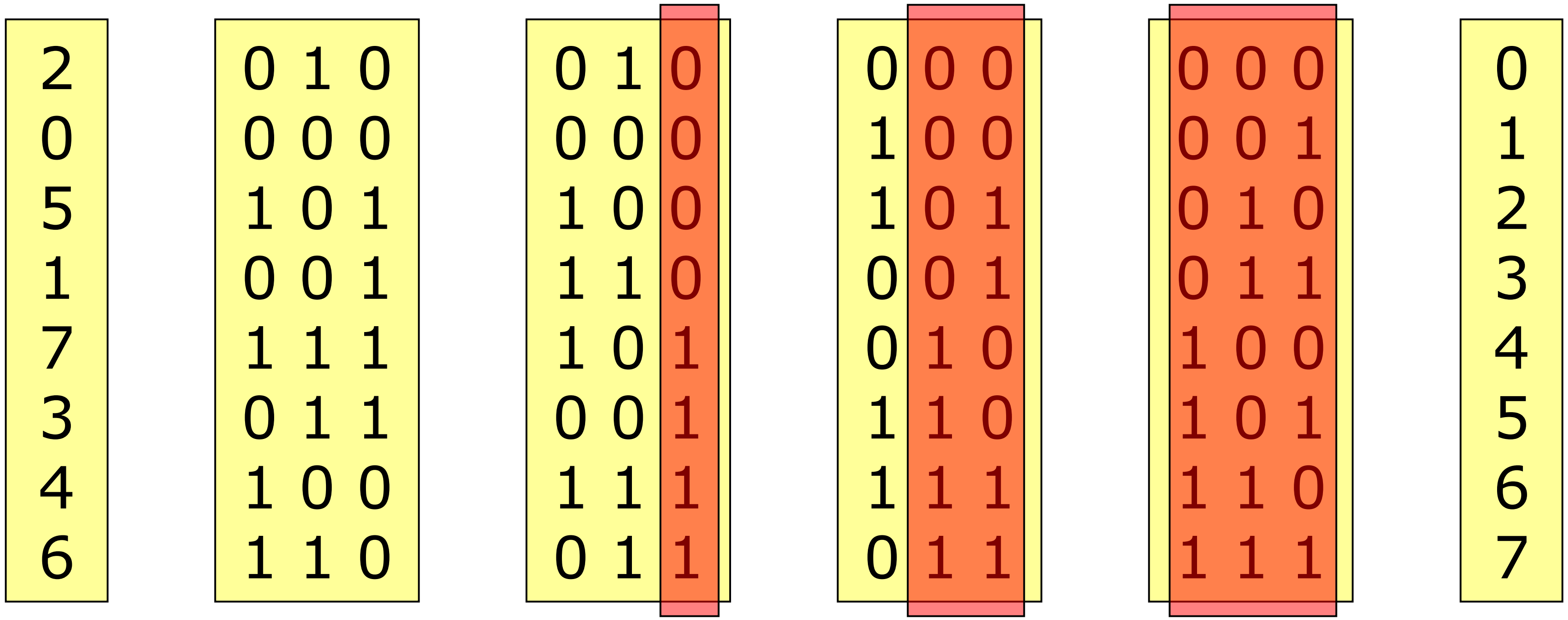
000 001 010 011

100 101 110 111

000 001 010 011 100 101 110 111

Radix Sort: bucket sort on every digit/bit

- For N elements between (L, H), using H-L+1 buckets can sort the elements in one round
- Problem: the range (L, H), if there is one, may be too large.
  - Sorting 4-byte unsigned integers, range is [0, 2^32-1]
- Solution(radix sort): apply bucket sort on every digit/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

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. 1<u>01</u> comes before 0<u>11</u> 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.

You can choose an appropriate radix value

- Numbers in different formats
  - decimal whole numbers: (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)
- Radix sort of decimal numbers
  - 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
- Selection is a trade-off between time complexity and space complexity.
  - For the above examples, how many buckets are needed? And how many passes must be made?

Radix Sort Algorithm

```
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
    }
}
```

RadixSorting Strings

- Single characters can be Bucket-Sorted
- Break strings into characters
- Need to know length of biggest string (or calculate this on the fly).

	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 Sort IEEE Floats/Doubles

- It is straightforward to use radix sort on integers or strings with fixed lengths.
- 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++.
- The key:
  1. Always move from the least significant part to the most significant part.
  2. Join the lists in correct orders.

Additional Slides

- Read by yourself!

IEEE floating point

- Shifting the binary point one position left
  - Divides the number by 2
  - Compare 101.11 base 2 with 10.111 base 2
- Shifting the binary point one position right
  - Multiplies the number by 2
  - Compare 101.11 base 2 with 1011.1 base 2
- Numbers 0.111…11 base 2 represent numbers just below 1 -> 0.111111 base 2 = 63/64
- Only finite-length encodings
  - 1/3 and 5/7 cannot be represented exactly
- Fractional binary notation can only represent numbers that can be written x * 2^y i.e. 63/64 = 63*2^{-6}
  - Otherwise, approximated
  - Increasing accuracy = lengthening the binary representation but still have finite space

IEEE floating point

- IEEE Standard 754 floating point is the most common representation today for real numbers on computers
- Limited range and precision (finite space)
- “real numbers” having a decimal portion != 0
- Example: 12.34 base 10 Meaning: $1*10^{1} + 2*10^{0} + 3*10^{-1} + 4*10^{-2}$
- Digit format: $d_{m}d_{m-1}$…$d_{1}d_{0}.d_{-1}d_{-2}$…d_{-n}
- dnum -> summation_of(i = -n to m) $d_{i} * 10^{i}$
- Example: 110.11 base 2 Meaning: $1*2^{2} + 1*2^{1} + 0*2^{0} + 1*2^{-1} + 1*2^{-2}$
- Digit format: b_{mbm-1}…$b_{1}b_{0}.b_{-1}b_{-2}$…b_{-n}
- bnum -> summation_of(i = -n to m) $b_{i} * 2^{i}$
- In both cases, digits on the left of the “point” are weighted by positive power and those on the right are weighted by negative powers

Floating-Point Representation in Computer

Sign(positive or negative)

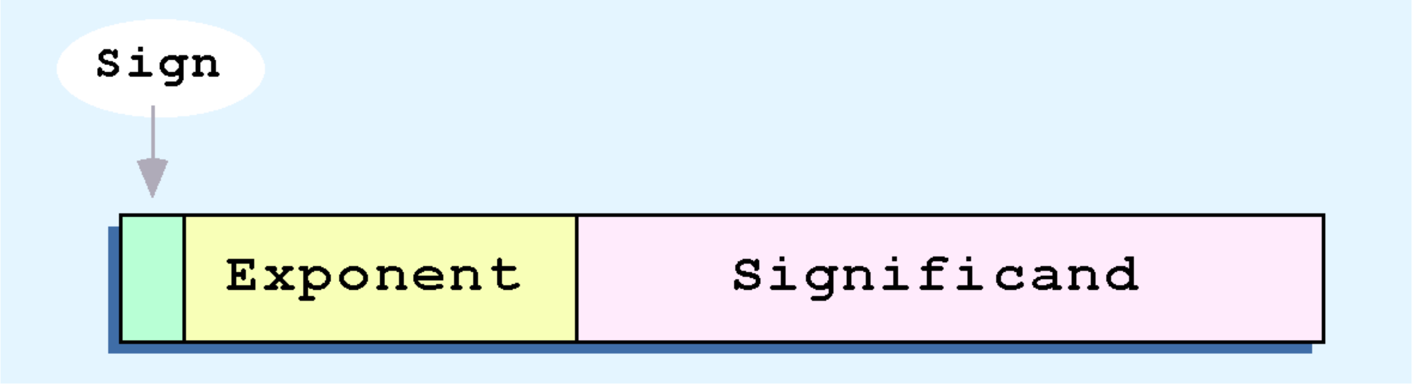
$-1.01*2^{101}$

$+1.11*2^{-11}$

Exponent

Significand (a.k.a. mantissa)

Computer representation of a floating-point number consists of three fixed-size fields:



Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

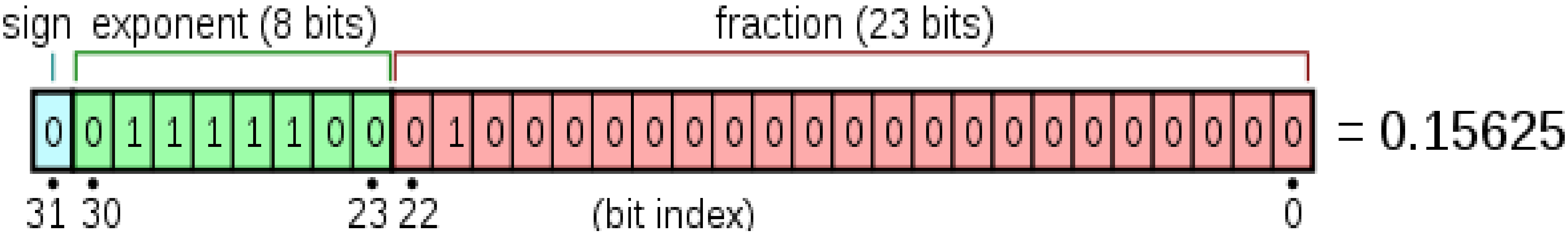
The sign bit and the exponent

- The sign bit is as simple as it gets.
  - 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.
- The exponent field needs to represent both positive and negative exponents.
  - A bias is added to the actual exponent in order to get the stored exponent.
  - For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.
  - For double precision, the exponent field is 11 bits, and has a bias of 1023.

More on the “bias”

- In IEEE 754 floating point numbers, the exponent is biased in the engineering sense of the word – the value stored is offset from the actual value by the exponent bias.
- **Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**
- To solve this problem the exponent is biased before being stored, by adjusting its value to put it within an unsigned range suitable for comparison.
- By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.
- When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.
- For a single-precision number, an exponent in the range −126 .. +127 is biased by adding 127 to get a value in the range 1 .. 254 (0 and 255 have special meanings). For a double-precision number, an exponent in the range −1022 .. +1023 is biased by adding 1023 to get a value in the range 1 .. 2046 (0 and 2047 have special meanings).

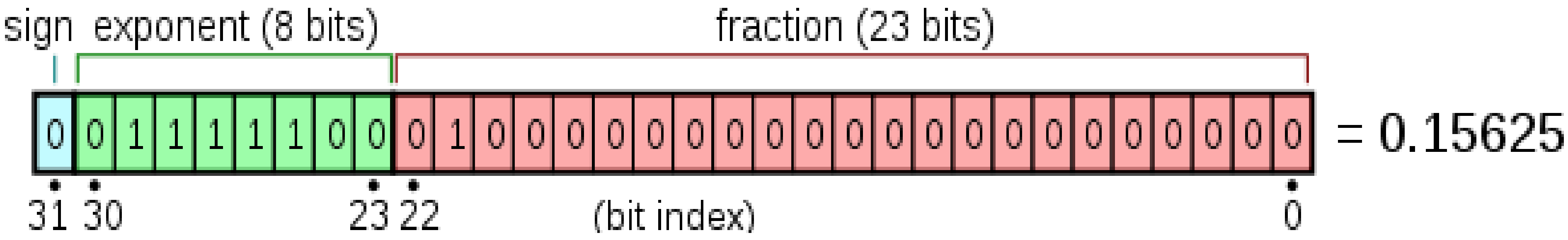
- Sign: 1 bit; 0 --- non-positive, 1 --- negative
- Significand: always 1.*fraction*. *fraction* uses 23 bits
- Biased exponent: 8 bits.
  - Bias: represent –126 to +127 by adding 127 (so range is 0-254)



$$+1.01*2^{(01111100-01111111)}$$

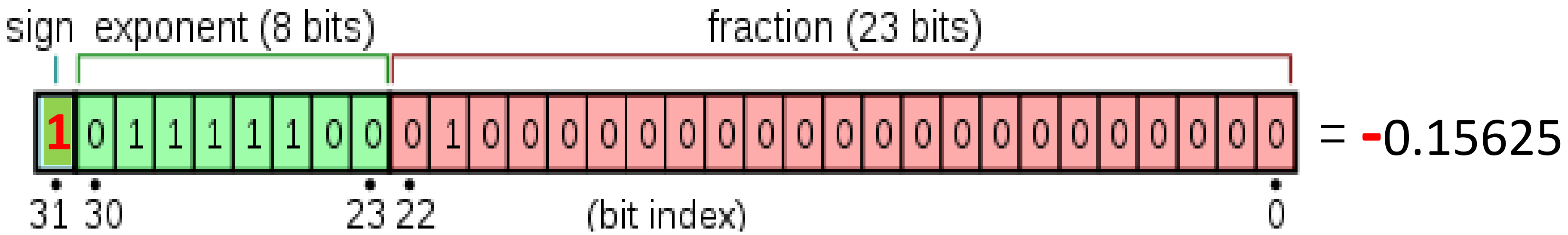
$$1.01*2^{-11} \rightarrow 101*2^{-101} \rightarrow 5/32$$

Observations



- Non-negative float point numbers
  - Larger value in a bit/octal digit/hexadecimal digit means larger number
    - e.g., 0 01111100 010000… =0.15625 (the value above);
      0 01111100 110000… =0.21875
    - Join the buckets with smaller bits/digits first.
  - Values are more determined by a higher digit than any lower digits
    - e.g., 0 01111100 010000… =0.15625 (the value above);
      0 01111100 100000… =0.18750
    - Exponent always more significant than significand
    - e.g., 0 01111100 010000… =0.15625 (the value above);
      0 01111101 000000… =0.25
  - Same proof is still valid for non-negative float point numbers

Observations



- Negative float point numbers
  - Larger value in a bit/octal digit/hexadecimal digit means **smaller** number
    - e.g., 1 01111100 **0**10000… = -0.15625 (the value above);
      1 01111100 **1**10000… = -0.21875 (smaller)
    - Join the buckets with **larger** bits/digits first.
  - Values are more determined by a higher digit than any lower digits
    - e.g., 1 01111100 **01**0000… =-0.15625 (the value above);
      1 01111100 **10**0000… =-0.18750 (smaller)
    - 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 (smaller)
  - Same proof is still valid for float point numbers

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 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?
  - Sort all the numbers as if they were all **unsigned integers**.
    - Join the buckets in the same way (smaller bit/digits first) for all the numbers
  - What does the sorted list look like?
    - All the negative numbers are after non-negative numbers
    - In the part of non-negative numbers, all the numbers are in **ascending** order
    - In the part of negative numbers, all the numbers are in **descending** order
  - Fix the order by re-organizing the numbers.
    - Flip the order of negative #s, and move negative #s before non-negative #s.