fit@hcmus

DATA STRUCTURES & ALGORITHMS

Lecture 2: Sorting (part 2)

Lecturer: Dr. Nguyen Hai Minh

CONTENT

- Sorting Lower Bound
  - Decision trees
- Analysis of sorting algorithms using different algorithm design methods (cont)
  - Space and Time tradeoffs: Counting Sort, Radix Sort

SORTING LOWER BOUND

How fast can we sort?

- All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.
  - *E.g.*, insertion sort, merge sort, quicksort, heapsort.
- The best worst-case running time that we've seen for comparison sorting is $O(n\log_2 n)$.

Is $O(n\log_2 n)$ the best we can do?

- ***Decision trees*** can help us answer this question

Asymptotic lower bound – Ω-notation

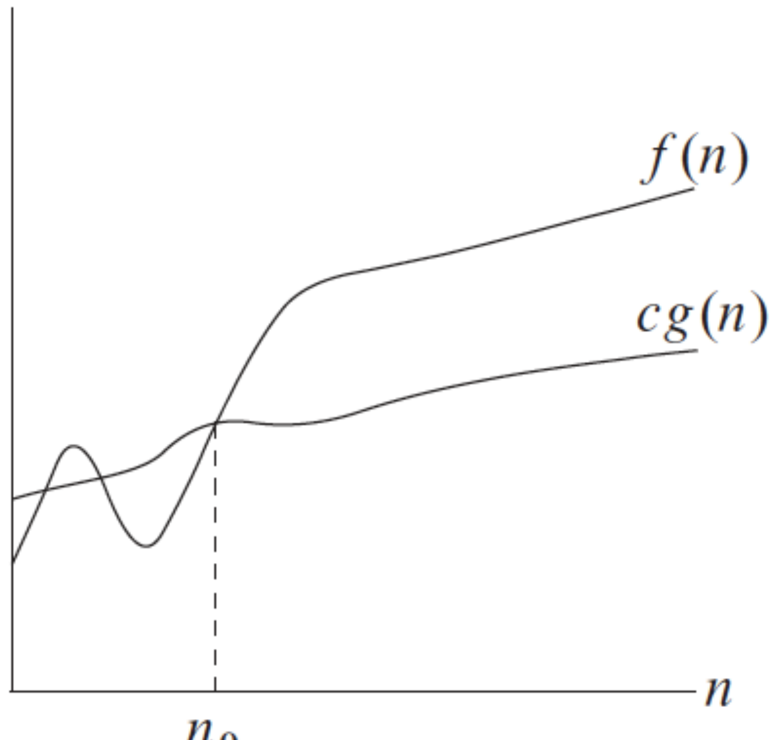
- Provides an asymptotic lower bound on a function
  - For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_o \text{ such that: } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

- Example:
  - Explain: f is big-omega of g if there is c so that f is on or above $c * g$ when n is large enough
  - $\sqrt{n} = \Omega(\log_2 n)$ $(c = 1, n = 16)$

Asymptotic lower bound – Ω notation

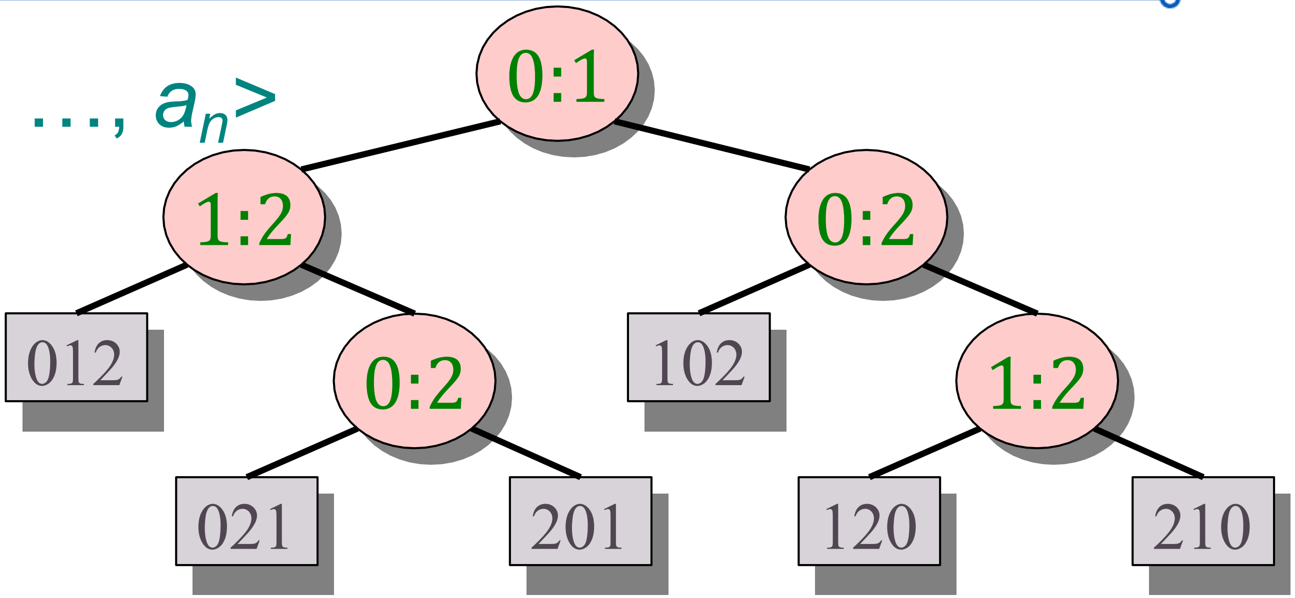
- Running time of an algorithm is $\Omega(g(n))$ means that the running time of that algorithm is at least a constant times $g(n)$, for sufficiently large n.



$f(n) = \Omega(g(n))$

Decision tree example

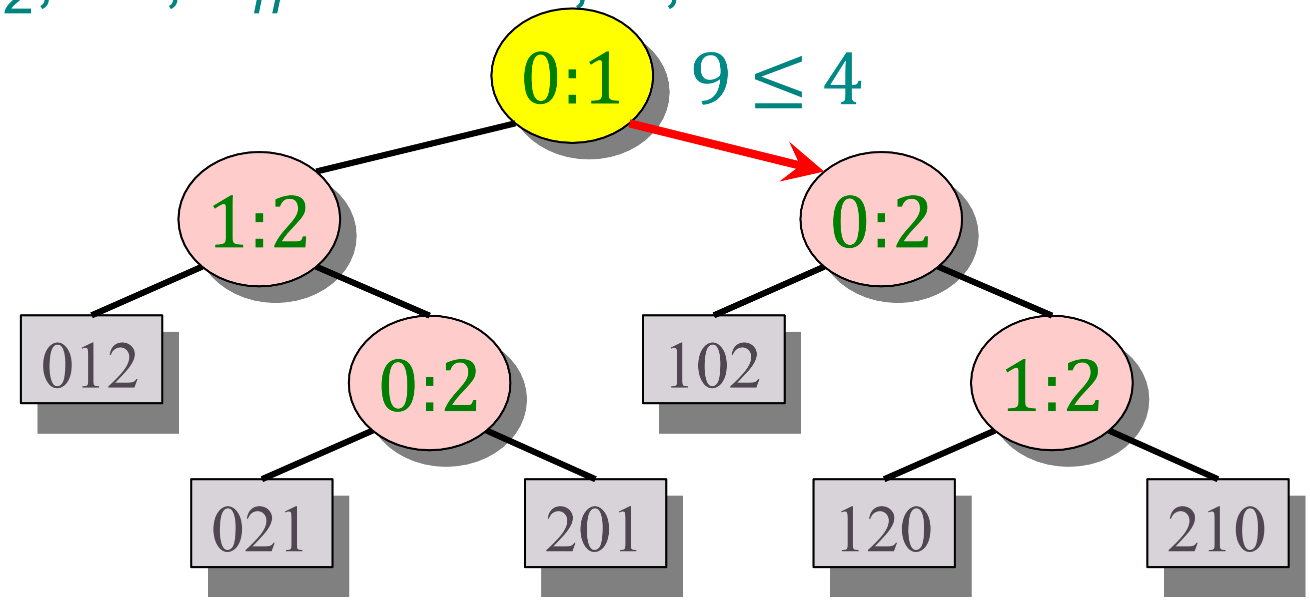
- Sort $\langle a_1, a_2, \ldots, a_n \rangle$



- Each internal node is labeled $i{:}j$ for $i, j \in \{0, 1, \ldots, n\text{-}1\}$.
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$.
  - The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision tree example

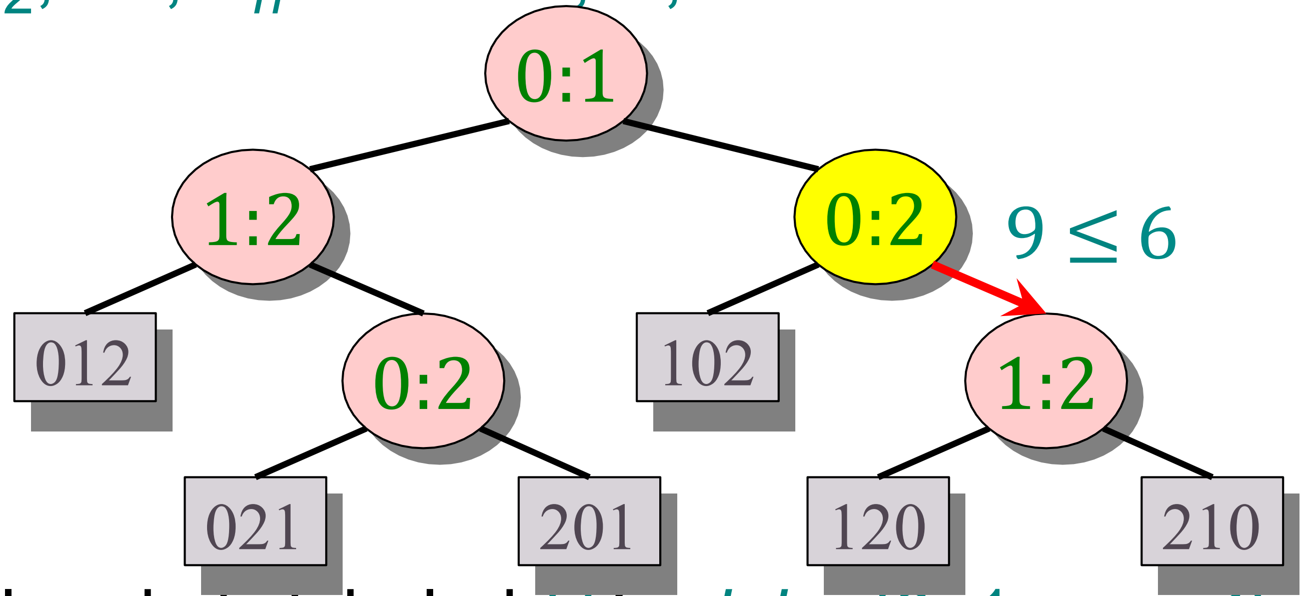
- Sort $<a_1, a_2, \ldots, a_n> = < 9, 4, 6 >$



- Each internal node is labeled $i{:}j$ for $i, j \in \{0, 1, \ldots, n\text{-}1\}$.
  - The left subtree shows subsequent comparisons if $a_i \le a_j$.
  - The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision tree example

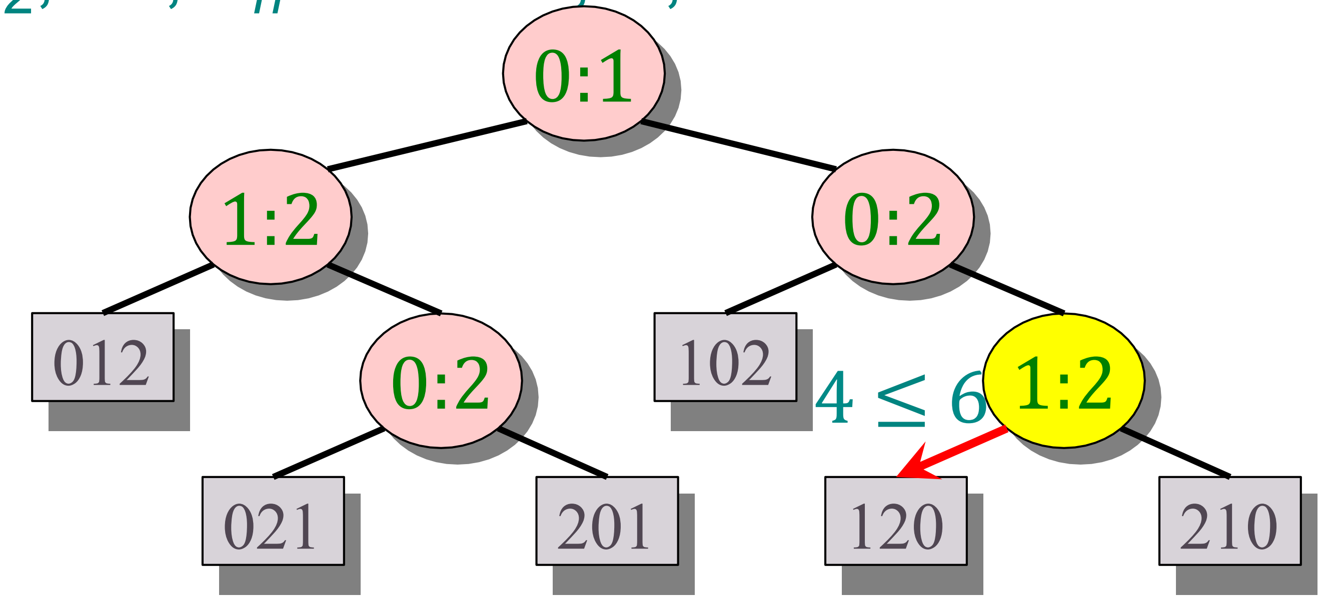
- Sort $<a_1, a_2, \ldots, a_n> = < 9, 4, 6 >$



- Each internal node is labeled $i{:}j$ for $i, j \in \{0, 1, \ldots, n-1\}$.
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$.
  - The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision tree example

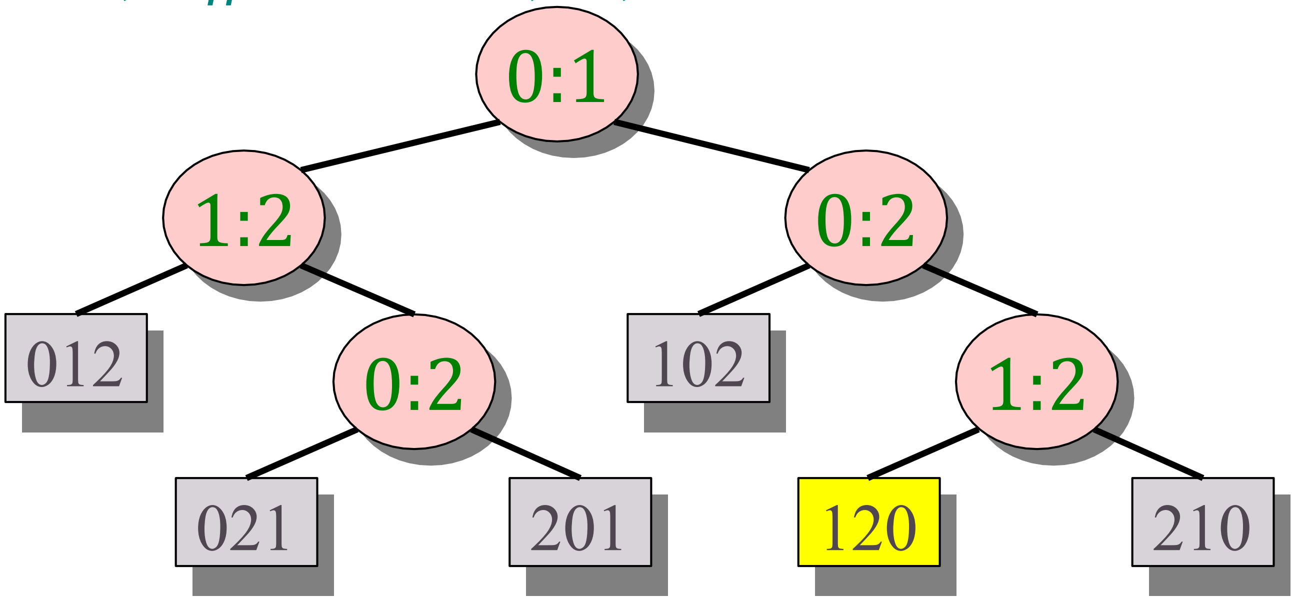
- Sort $\langle a_1, a_2, \ldots, a_n \rangle = \langle 9, 4, 6 \rangle$



- Each internal node is labeled $i{:}j$ for $i, j \in \{0, 1, \ldots, n\text{-}1\}$.
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$.
  - The right subtree shows subsequent comparisons if $a_i > a_j$.

Decision tree example

- Sort $<a_1, a_2, \ldots, a_n> = < 9, 4, 6 >$



- Each leaf contains a permutation $4 \leq 6 \leq 9$ $\langle \pi(0), \pi(1), \ldots, \pi(n-1) \rangle$ to indicate that the ordering $a_{\pi(0)} \leq a_{\pi(1)} \leq \cdots \leq a_{\pi(n-1)}$ has been established.

Decision tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size *n*.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \log_2 n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore \quad h &\geq \log_2(n!) && (\log_2 n \text{ is monotonically increasing}) \\
&\geq \log_2((n/e)^n) && (\text{Stirling's formula}) \\
&= n \log_2 n - n \log_2 e \\
&= \Omega(n \log_2 n)
\end{aligned}
$$

Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms.

SPACE-AND-TIME TRADEOFFS ALGORITHMS

Counting Sort

Radix Sort

Space and Time Trade-offs

- ***Space and time trade-offs*** are a well-known issue for both theoreticians and practitioners of computing.
- Consider the problem of computing values of a function at many points in its domain:
  - Precompute the function's values and store them in a table to speed up running time.
  - This idea is quite useful in the development of some important algorithms for other programs.

Space and Time Trade-offs

- **Input Enhancement:**
  - Preprocess the problem's input and store the additional information obtained to accelerate solving the problem
  - E.g., *Counting Sort, Boyer-Moore string matching*
- **Prestructuring:**
  - Use extra space to facilitate faster and/or more flexible access to the data
  - E.g., *Hashing, indexing with B-trees*
- **Dynamic Programming:**
  - Record solutions to overlapping subproblems of a given problem in a table
  - E.g., *the Knapsack problem*

Counting Sort Idea

- One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table.
- These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position
- Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.

Counting Sort

COUNTING-SORT(A[0…n-1],k)
//Input: An array A[0..n − 1] of integers between [0,k]
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
	Cost times

for j ← 0 **to** k **do**	
C[j] ← 0	$k+1$
for i ← 0 **to** n - 1 **do**	
C[A[i]] ← C[A[i]] + 1	n
for j ← 1 **to** k **do**	
C[j] ← C[j - 1] + C[j]	k
for i ← n-1 **to** 0 **do**	
S[C[A[i]] - 1] ← A[i]	n
C[A[i]] ← C[A[i]] – 1	n
return S	

Counting Sort Analysis

1. Input size: n, k
2. Basic operation: assignment & addition inside 4 loops
3. The number of key comparisons ***depends on the array size and the max value of the array.***
4. Sum of number of times the basic operations is:
   $$C(n,k) = k + 1 + n + k + n + n = 2k + 3n + 1$$
5. Order of growth: $\boldsymbol{O(n+k)}$

Counting sort – Illustration

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3

S:

Counting Sort – Loop 1

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
0	0	0	0

S:

for $j \leftarrow 0$ **to** k

  do $C[j] \leftarrow 0$

Counting Sort – Loop 2

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
0	0	0	1

S:

for $i \leftarrow 0$ **to** $n\text{-}1$

do $C[A[i]] \leftarrow C[A[i]] + 1$ $\quad \triangleleft\ C[i] = |\{\text{key} = i\}|$

Counting Sort – Loop 2

	0	1	2	3	4
A:	3	0	2	3	2

	0	1	2	3
C:	1	0	0	1

S:					

for $i \leftarrow 0$ **to** $n\text{-}1$

  do $C[A[i]] \leftarrow C[A[i]] + 1$ $\lhd$ $C[i] = |\{\text{key} = i\}|$

Counting Sort – Loop 2

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	1	1

S:

for $i \leftarrow 0$ **to** $n\text{-}1$

  do $C[A[i]] \leftarrow C[A[i]] + 1$ $\triangleleft$ $C[i] = |\{\text{key} = i\}|$

Counting Sort – Loop 2

	0	1	2	3	4
A:	3	0	2	3	2

	0	1	2	3
C:	1	0	1	2

S:					

for $i \leftarrow 0$ **to** $n\text{-}1$

do $C[A[i]] \leftarrow C[A[i]] + 1$ $\quad\triangleleft\ C[i] = |\{\text{key} = i\}|$

Counting Sort – Loop 2

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

for $i \leftarrow 0$ **to** $n\text{-}1$

  do $C[A[i]] \leftarrow C[A[i]] + 1$ $\triangleleft\ C[i] = |\{\text{key} = i\}|$

Counting Sort – Loop 3

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

C':

0	1	2	3
1	1	2	2

for $j \leftarrow 1$ **to** k
  do $C[j] \leftarrow C[j] + C[j-1]$ $\triangleleft$ $C[j] = |\{\text{key} \le j\}|$

Counting Sort – Loop 3

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

C':

0	1	2	3
1	1	3	2

for $j \leftarrow 1$ **to** k

do $C[j] \leftarrow C[j] + C[j-1]$ $\quad \triangleleft \ C[j] = |\{\text{key} \le j\}|$

Counting Sort – Loop 3

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

C':

0	1	2	3
1	1	3	5

for $j \leftarrow 1$ **to** k

  do $C[j] \leftarrow C[j] + C[j-1]$ $\triangleleft$ $C[j] = |\{\text{key} \le j\}|$

Counting Sort – Loop 4

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

		2		

C':

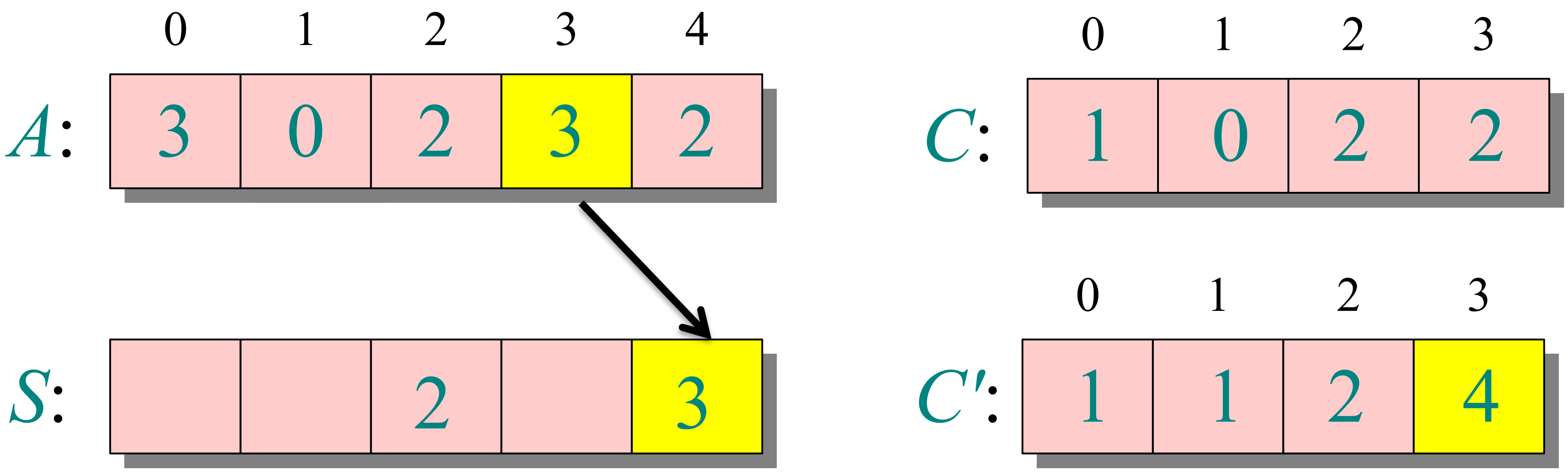
0	1	2	3
1	1	2	5

for $i \leftarrow n\text{-}1$ **down to** 0

  do $S[C[A[i]] - 1] \leftarrow A[i]$

    $C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort – Loop 4



for $i \leftarrow n\text{-}1$ **down to** 0

do $S[C[A[i]] - 1] \leftarrow A[i]$

$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort – Loop 4

A:

0	1	2	3	4
3	0	2	3	2

C:

0	1	2	3
1	0	2	2

S:

	2	2		3

C':

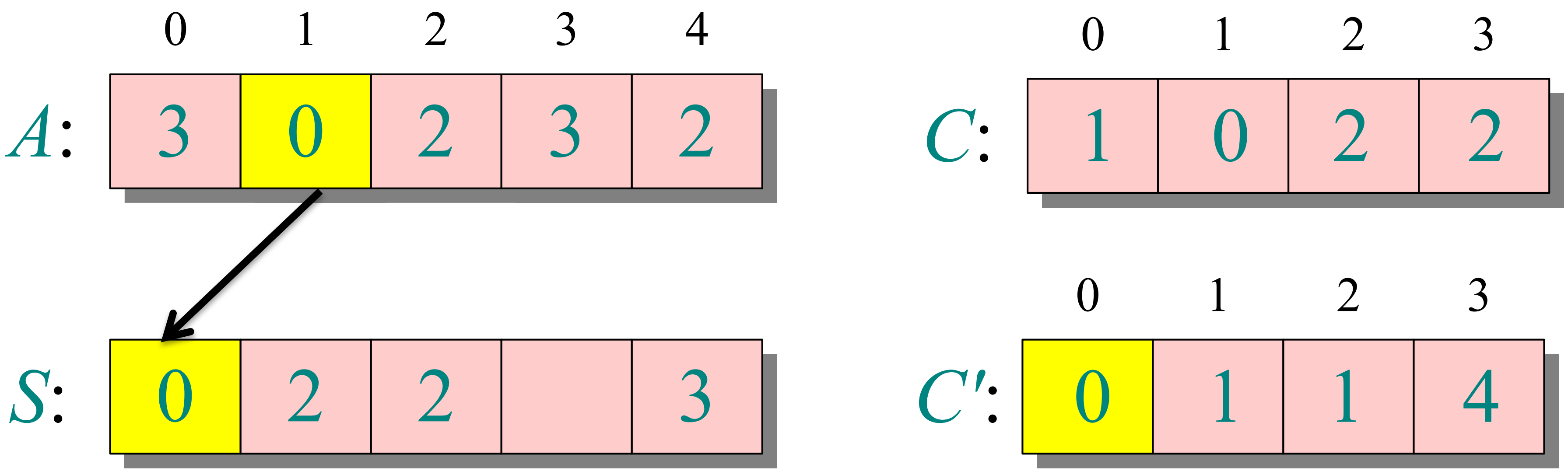
0	1	2	3
1	1	1	4

for $i \leftarrow n\text{-}1$ **down to** 0

do $S[C[A[i]] - 1] \leftarrow A[i]$

$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort – Loop 4



for $i \leftarrow n\text{-}1$ **down to** 0

do $S[C[A[i]] - 1] \leftarrow A[i]$

$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort – Loop 4

	0	1	2	3	4
A:	3	0	2	3	2

	0	1	2	3
C:	1	0	2	2

S:	0	2	2	3	3

	0	1	2	3
C':	0	1	1	3

for $i \leftarrow n\text{-}1$ **down to** 0

do $S[C[A[i]] - 1] \leftarrow A[i]$

$C[A[i]] \leftarrow C[A[i]] - 1$

Counting Sort – Running time

If $k = O(n)$, then counting sort takes $O(n)$ time.

- But, sorting takes $\Omega(n\log_2 n)$ time!
- Where's the fallacy?

Answer:

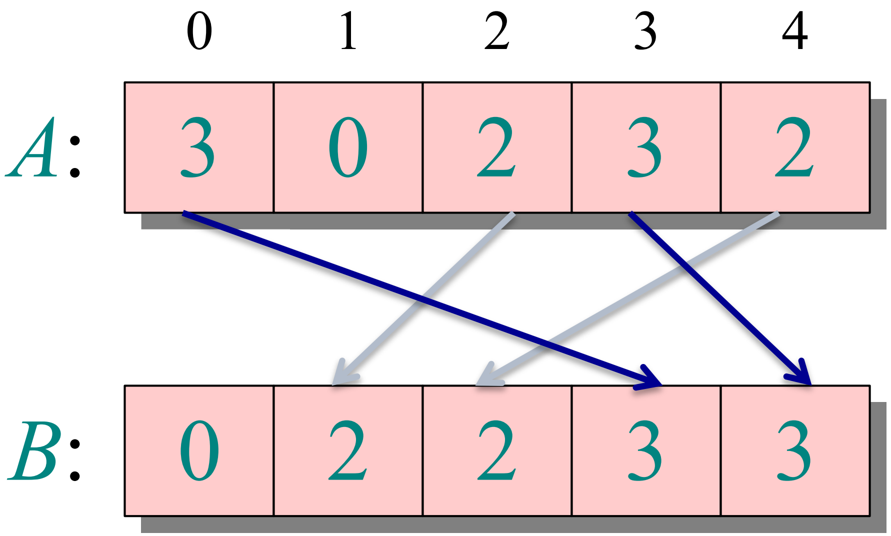
- ***Comparison sorting*** takes $\Omega(n\log_2 n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Counting Sort – Pros and Cons

- **Pros:**
  - It performs particularly well when the range of the input is small compared to the number of elements.
  - Stable sort
  - There is no comparison operation. Instead, it uses integer counting and index-based placement to sort the elements, resulting in faster execution.
- **Cons:**
  - Limited to sorting integers
  - Not in-place. It requires additional memory space proportional to the range of the input.
  - The input range must be known in advance.

Stable sorting

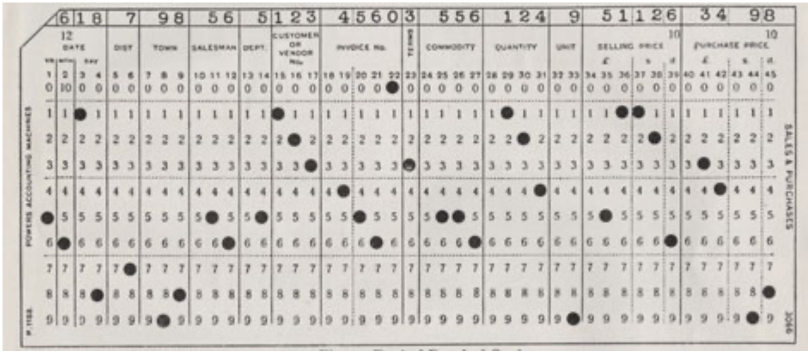
- Counting sort is a ***stable*** sort: it preserves the input order among equal elements.



- **Exercise:** What other sorts have this property?

Radix Sort

- Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
  - The cards have 80 columns, each has 12 places to punch by a machine.



  - The sorter can examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.

Radix Sort Idea

- For decimal digits, each column uses only 10 places.

  → A *d*-digit number occupies a field of *d* columns.
- Since the card sorter can look at only one column at a time, the problem of sorting *n* cards on a *d*-digit number requires a sorting algorithm:
  - **Intuitively:** Sort numbers on their **most significant** (leftmost) digit first.
  - **Better idea:** Sort numbers on their **least significant** (rightmost) digit first with auxiliary *stable* sort.
    - Then, it sorts the entire deck again on the second-least significant digit and recombines the deck.
    - Only d passes through the deck are required to sort.

Radix Sort – Algorithm & Analysis

RADIX-SORT(A[0…n-1],d)
//Input: An array A[0..n − 1] of n d-digit integers
//Output: Array A[0..n − 1] sorted in nondecreasing order
	Cost times

for i ← 0 **to** d-1 **do** Use a stable sort to sort array A on digit i	 $d \cdot C(n)$

n d-digits numbers

Counting Sort: O(n+k)

each has k possible values

O(d(n+k)) → if d is constant and $k = O(n)$ → **O(n)**

Operation of LSD Radix sort

- Radix sort on a “deck” of seven 3-digit numbers:

1st pass

3 2 **9**	7 2 **0**
4 5 **7**	3 5 **5**
6 5 **7**	4 3 **6**
8 3 **9**	4 5 **7**
4 3 **6**	6 5 **7**
7 2 **0**	3 2 **9**
3 5 **5**	8 3 **9**

Operation of LSD Radix sort

- Radix sort on a “deck” of seven 3-digit numbers.

2nd pass

3 2 9	7 2 0	7 2 0
4 5 7	3 5 5	3 2 9
6 5 7	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9
4 3 6	6 5 7	3 5 5
7 2 0	3 2 9	4 5 7
3 5 5	8 3 9	6 5 7

Operation of LSD Radix sort

- Radix sort on a “deck” of seven 3-digit numbers.

3[rd] pass

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Finish!

Radix Sort – Pros and Cons

- Pros:
  - In practice, radix sort is fast for large inputs, as well as simple to code and maintain.
  - Can be used to sort records of information that are keyed by multiple fields.
- Cons:
  - The digit sorts must be stable.
  - Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort far better on modern processors.

Radix Sort – Lemma

- **Lemma :** Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $((b/r)(n+2^r))$ time if the stable sort it uses takes $O(n + k)$ time for inputs in the range 0 to k.
- **Proof:**
  - See Textbook 1, page 295~

Most significant digit Radix sort

- Use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations.
- No need to preserve the order of duplicate keys
- Example:
  - car, bar, care, bare → bar, bare, car, care
  - 9, 8, 10, 1, 3 → 1, 10, 3, 8, 9

More Reading

- Stirling's approximation
  - Textbook 1 – Page 57

A weak upper bound on the factorial function is $n! \le n^n$, since each of the n terms in the factorial product is at most n. *Stirling's approximation*,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.18}$$

where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well. As Exercise 3.2-3 asks you to prove,

$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) , \end{aligned} \tag{3.19}$$

What's next?

- After today:
  - Read textbook 1 – Chapter 8
  - Read textbook 3 – 7.1
  - Do Homework 2

Q&A