Analysis and Design of Algorithms

CS3230

Week 4

Sorting lower bound
Linear-time sorting

Ken Sung

Diptarka Chakraborty

Any exponential function with base $a > 1$ grows faster than any polynomial

- Lemma: For any constants k>0 and a>1, $n^k = o(a^n)$.
- Proof: For any c>0, we need to find n_0 s.t. $n^k < c \cdot a^n$ for $n \geq n_0$.
  - $\frac{n}{\ln n}$ is an increasing function.
  - There exists n_0 such that $\frac{2k}{\ln a} < \frac{n}{\ln n}$ and $\frac{-2\log_a c}{\ln n} < \frac{n}{\ln n}$ for $n \geq n_0$.
  - We have $\frac{2k}{\ln a} + \frac{-2\log_a c}{\ln n} < \frac{2n}{\ln n} \rightarrow \frac{k}{\ln a} < \frac{n}{\ln n} + \frac{\log_a c}{\ln n}$
  - $k \log_a n = \frac{k \ln n}{\ln a} < n + \log_a c \rightarrow a^{k \log_a n} < c \cdot a^n \rightarrow n^k < c \cdot a^n$.
  - Hence, $n^k = o(a^n)$.

Recall previous question

- Let $F_{n+1} = F_n + F_{n-1} + F_{n-2}$. with $F_1=1$, $F_2=1$, $F_3=1$.
- Can you give an efficient algorithm to compute F_n?
- What is the running time?

- (1) $\Theta(n)$
- (2) $\Theta(\log n)$
- (3) $\Theta(\log^2 n)$
- (4) $\Theta(\log \log n)$

Better Answer

- The answer is (2).
- Theorem 1: $\begin{bmatrix} F_n \\ F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \\ F_{n-3} \end{bmatrix}$
- Theorem 2: $\begin{bmatrix} F_n \\ F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{n-3} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
- By the technique of powering a matrix, we can obtain an $\Theta(\log n)$-time algorithm.

Question 1

- A divide and conquer algorithm for solving problem has recurrence
  $T(n)=8T(n/2)+n^3$.
- Two improvements to the algorithm are found.

- **Improvement I**: The time for divide and combine is reduced from n^3 to n^2.

- **Improvement II**: The runtime for divide and combine remains the same, but the number of subproblems is reduced from 8 to 7.

- Which of the improvements is asymptotically better?
  1. Improvement 1
  2. Improvement 2
  3. The two improvements result in algorithms with the same asymptotic runtimes

Solution

- Original:
- T(n) = 8 T(n/2) + n^3
- a=8, b=2, $f(n) = n^3 = \Theta\left(n^{\log_2 8}\right)$
- It is case 2. Hence, $T(n) = \Theta(n^3 \lg n)$

- Improvement I:
- T(n) = 8 T(n/2) + n^2
- a=8, b=2, $f(n) = n^2 = O\left(n^{\log_2 8 - \epsilon}\right)$
- It is case 1. Hence, $T(n) = \Theta(n^3)$

- Improvement II:
- T(n) = 7 T(n/2) + n^3
- a=4, b=2, $f(n) = n^3 = \Omega\left(n^{\log_2 7 + \epsilon}\right)$
- It is case 3. Hence, $T(n) = \Theta(n^3)$

Improvement I and II gives the same improvement

Question 2

- Consider the following method of using a heap for merging two sorted lists in merge sort:
- **Method I:** Construct a min heap using both lists. Extract the minimum elements until the heap is empty.
- **Method II:** Put the first element in each list in a min heap. Extract the minimum element and replace the element by inserting the next element from the same list. Repeat until all elements have been inserted and extracted from the heap.

- Which of the methods give a better asymptotic running time when used in merge sort?
  1. Method 1
  2. Method 2
  3. The two methods give the same asymptotic runtime.

Solution

- Method 1: Build heap then extract all elements.
- $T(n) = 2T(n/2) + \Theta(n \lg n)$
- a=2, b=2, $f(n) = n \lg n = \Theta\left(n^{\lg_2 2} \lg n\right)$
- It is case 2. Hence, $T(n) = \Theta(n \lg^2 n)$

- Method 2: Heap size is always at most 2. So inserting and extracting all elements take f(n) = Θ(n) time.
- $T(n) = 2T(n/2) + \Theta(n)$
- a=2, b=2, $f(n) = n = \Theta\left(n^{\lg_2 2}\right)$
- It is case 2. Hence, $T(n) = \Theta(n \lg n)$

n(1 – cos n) = O(n)?

- Let f(n) = n(1 – cos n)
- Let g(n) = n
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{n(1-\cos n)}{n}\right) = \lim_{n\to\infty}(1-\cos n)$ is not converge.

- So, is n(1 – cos n) = O(n)?

- The answer is yes!
- Note that when c=2,
  - n(1 – cos n) ≤ c n for n≥0.
- Hence, by definition, n(1 – cos n) = O(n).

However, $n(1-\cos n) \neq \Theta(n)$

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \rightarrow f(n) = o(g(n))$

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \rightarrow f(n) = O(g(n))$

$0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \rightarrow f(n) = \Theta(g(n))$

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \rightarrow f(n) = \Omega(g(n))$

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \rightarrow f(n) = \omega(g(n))$

Conclusion

- We know that $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Longrightarrow f(n) = O\big(g(n)\big)$
- But $f(n) = O\big(g(n)\big) \nRightarrow \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty.$

- Out of syllabus,
  - $\lim_{n\to\infty} sup\left(\frac{f(n)}{g(n)}\right) < \infty \leftrightarrow f(n) = O\big(g(n)\big)$

Note:
lim sup is limit superior.

E.g.
$\lim_{n\to\infty} \cos n$ does not exist.

$\lim_{n\to\infty} sup \cos n = 1$

Admin

- Homework 1 can be found in LumiNUS Files
  - Submit to the box in undergrad office COM1-02-19 by Thurs 6 Feb (today) before 6pm (Week 4, next week).

- Mid-term test:
  - Time: 7 Mar 2019 (Sat) 11:00am – 01:00pm.
  - Venue: MPSH2C

The sorting problem

- ***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.
- ***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Example:
  - ***Input:*** 8 2 4 9 3 6
  - ***Output:*** 2 3 4 6 8 9

- We learned a number of sorting algorithms:
  - Bubble sort, Insertion sort, Merge sort, Heap sort, Quick sort, etc.

Classification of the sorting algorithm

- The sorting algorithms are classified by
  - Running time (i.e. $O(n^2)$, $O(n \log n)$)

- We can also classify the sorting algorithms by
  1. In-Place
  2. Stable
  3. Comparison sort or not

In-Place

- A sorting algorithm is in-place if it uses very little additional memory beyond that used for the data, usually $O(1)$ or $O(\lg n)$.

- Useful when memory is limited.

Examples:

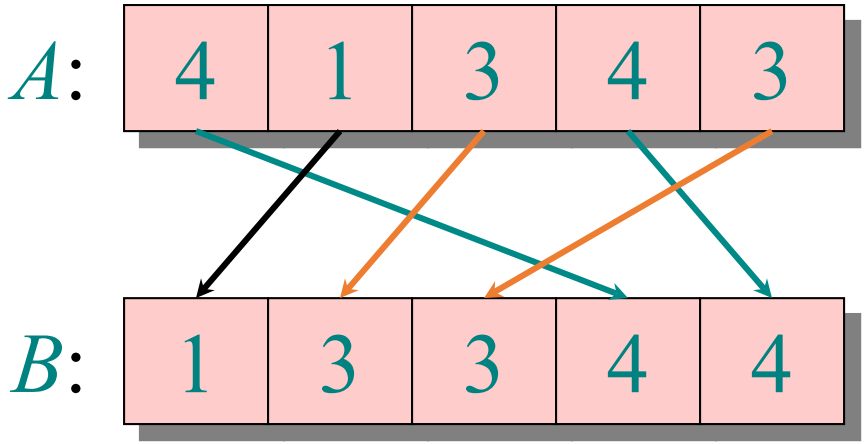
- Insertion sort
- Quicksort ($O(\lg n)$ additional memory with proper implementations)

Stable

- A sorting algorithm is stable if the original order of equal elements is preserved in the sorted output.

Examples:

- Insertion sort
- Merge sort



Comparison-based algorithm

- Comparison-based algorithm is an algorithm that only compares elements.

- A sorting algorithm that sort the elements by comparing them only is called a comparison 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.

What we want to study today?

1. What is the best time complexity for a comparison sort?

2. Is there any sorting algorithm which is better than a comparison sort?

What is the best time complexity for a comparison sort?

- So far, all sorting algorithms we learned are comparison sorts.

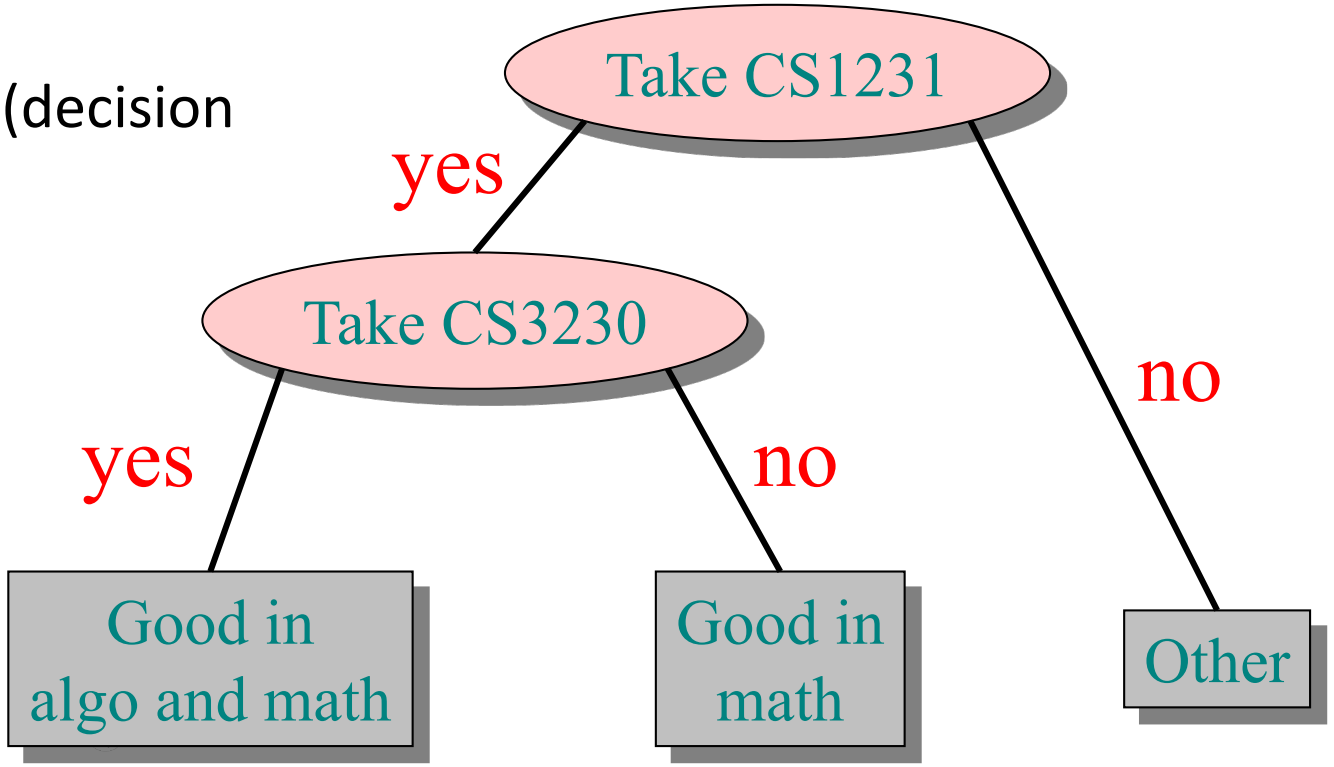
- The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

- ***Is $O(n \lg n)$ the best we can do?***
  - ***Decision trees*** can help us answer this question.

What is a decision tree?

```
if (take CS1231) then
    if  (take CS3230) then
        return "Good in Algo & Math"
    else
        return "Good in Math"
else
    return "other"
```

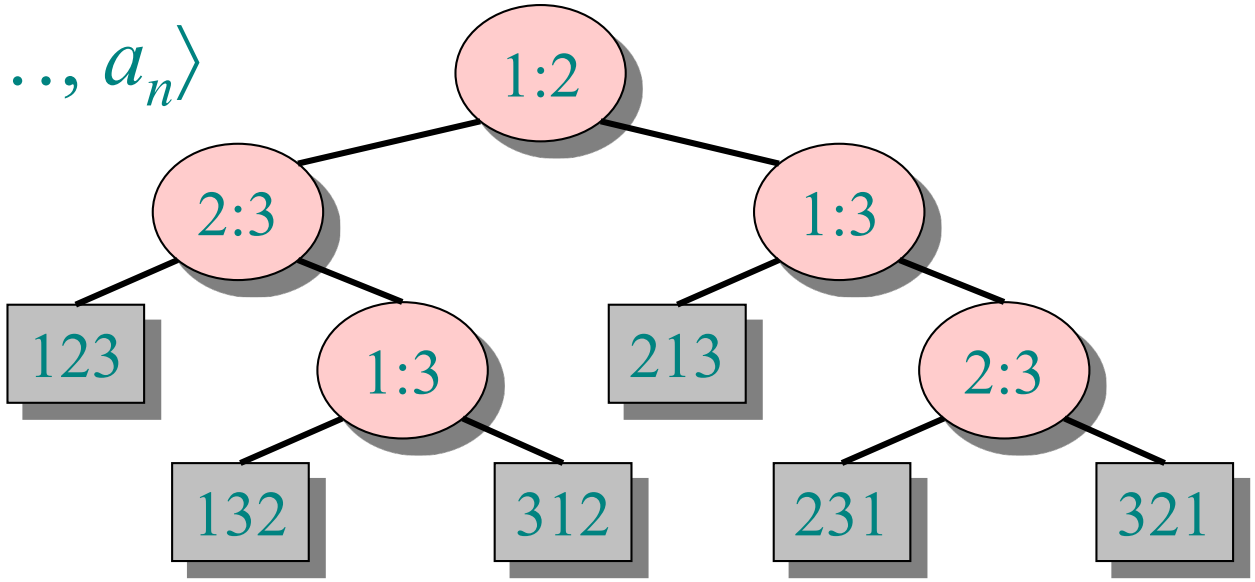
- A decision tree is a tree-like model.
  - Every node is a comparison.
  - Every branch represents the outcome of the comparison
  - Every leaf represents a class label (decision after all comparisons)
- We can use decision tree to model any comparison-based algorithm.



Decision-tree example

A decision tree models the execution of any comparison sort.

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

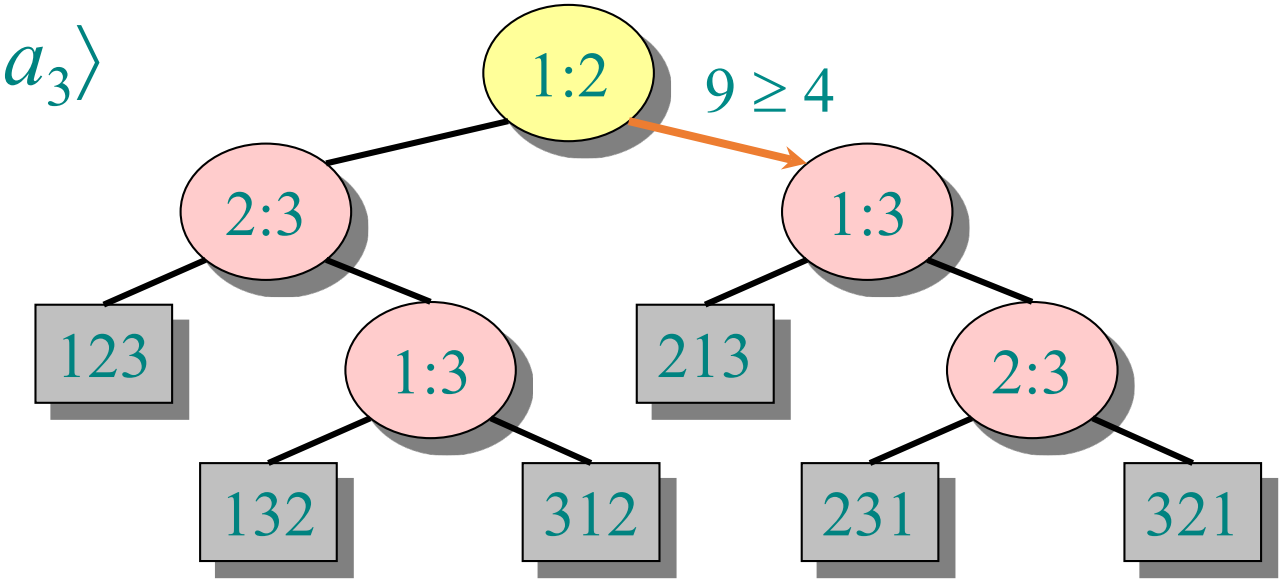


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:

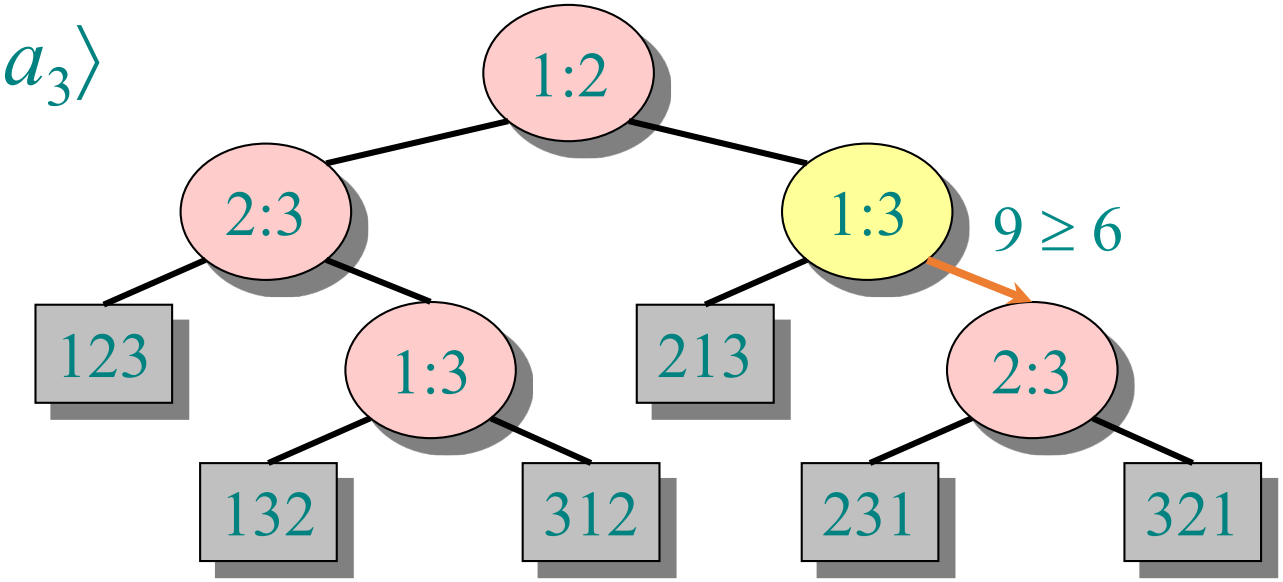


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

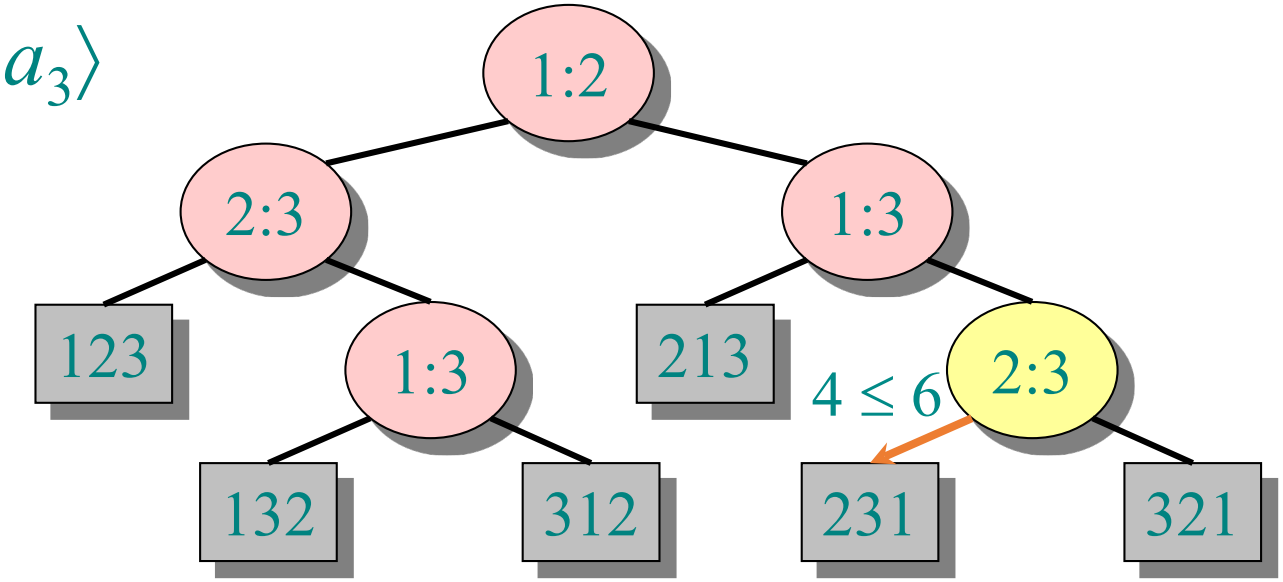


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

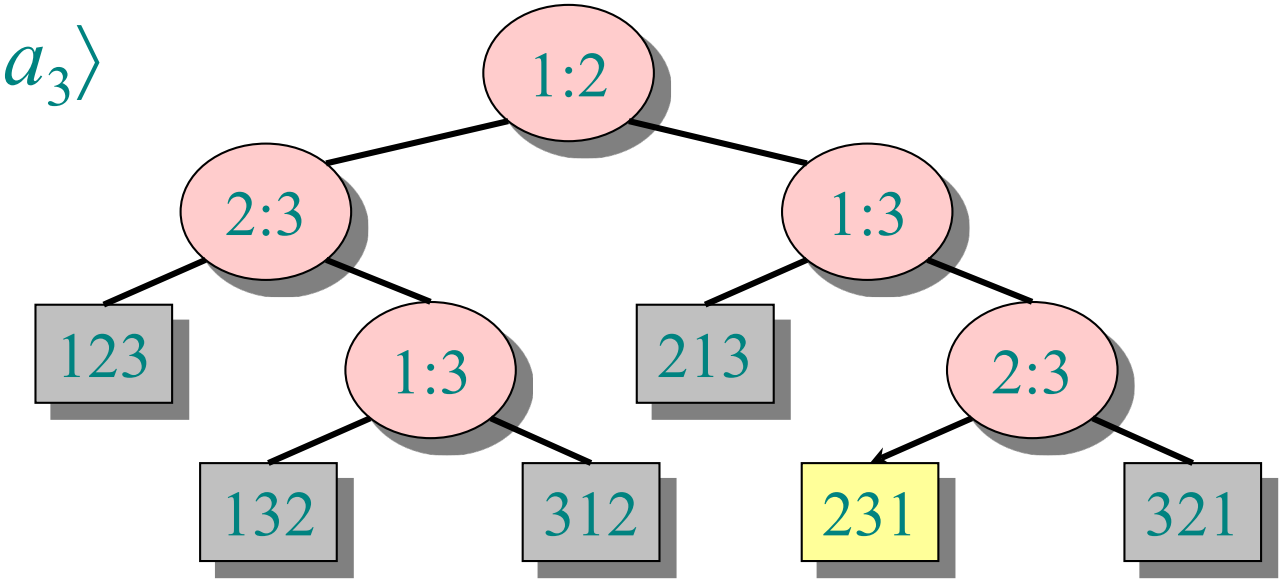


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i < a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

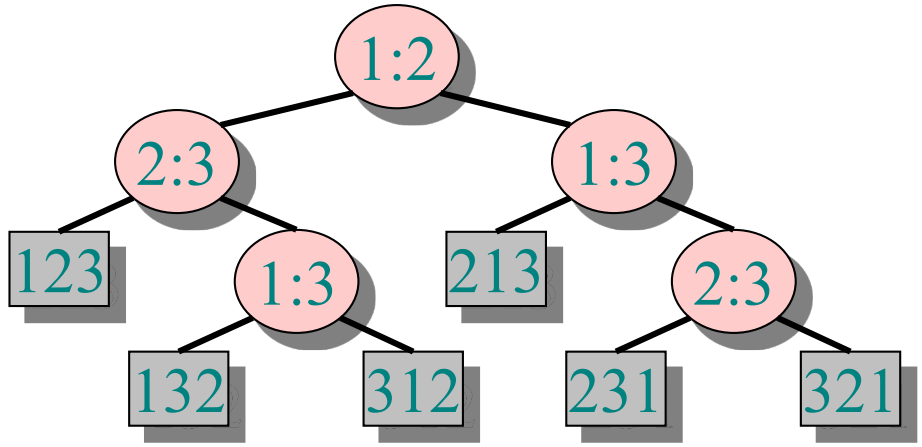
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \Lambda \le a_{\pi(n)}$ 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 $\Omega(n \lg n)$.
- *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, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore \quad h &\geq \lg(n!) && \text{(lg is mono. increasing)} \\
&\geq \lg\left((n/e)^n\right) && \text{(Stirling's formula)} \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n)\,. && \blacksquare
\end{aligned}
$$

Lower bound for comparison sorting

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

Question 3

- There exists a comparison-based sorting algorithm that can sort any 6-element array using at most 9 comparisons.

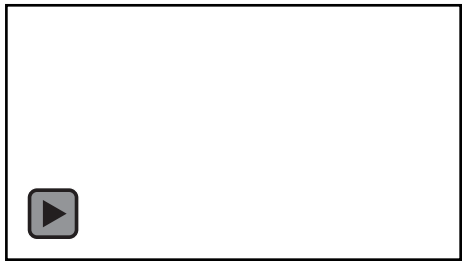
1. True
2. False

Solution

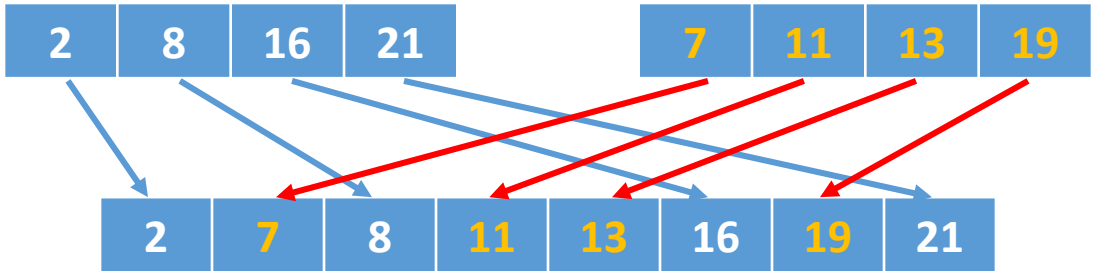
- **False**.
- There are 6! = 6 * 5 * 4 * 3 * 2 = 720 permutations of 6 elements.
- A complete binary tree of height 9 has only 512 leaves which is not enough to describe an algorithm that can output 720 permutations.
- Any comparison sort algorithm requires at least 10 comparisons for some input.

Question 4

- Given two sorted arrays A[1..n] and B[1..n] of real numbers (assuming all numbers are distinct).
- In the comparison model, we want to construct the sorted sequence of $A \cup B$.
- What is the lower bound of the running time?



Solution

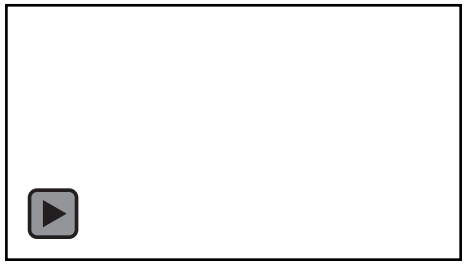


- **The lower bound is $\boldsymbol{\Omega(n)}$.**
- As A and B are sorted, number of possible ways to form sorted sequences of A∪B is not $(2n)!$.
- Number of ways to form sorted sequences of A∪B is $\frac{(2n)!}{n!n!}$.
- To sort A∪B, the lower bound is $\Omega\left(\log\frac{(2n)!}{n!n!}\right)$.
- Recall that $\lg(n!) = \Theta(n \lg n)$.
- Hence, $\lg\left(\frac{(2n)!}{n!n!}\right) = \Theta\big(\lg\big((2n)!\big) - 2\lg(n!)\big)$

$$= \Theta(2n\lg(2n) - 2n\lg n) = \Theta(n).$$

Question 5

- Ali has 81 coconuts, all of which have the same weight, except for one which is heavier. He does not know which is the heavier coconut. Ali's friend has a balance scale, but will charge Ali one dollar for each use of the scale.
- What is the maximum amount of money that Ali has to pay to guarantee that he can find the heaviest coconut, assuming that Ali uses the optimal algorithm?



Answer

Solution: Divide the coconuts into three equal sized piles and put two of the piles on the two sides of the scale. If they are equally heavy, repeat with the third pile, otherwise repeat with the heavier pile, until the heavy coconut is found.

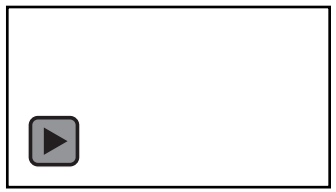
For 81 coconuts, we get the sequence of 27, 9, 3, and 1 coconuts, allowing us to identify the heavy coconut in 4 weightings.

Answer

- To see that this is optimal, note that the scale can divide the coconuts into at most 3 groups with each weighting.
- Any algorithm using only the scale can be described as a ternary tree with the label of the heavy coconut at each leaf.
- A full ternary tree of height 3 has only 27 leaves, hence there is no algorithm with the worst case number of weightings of 3 or fewer.

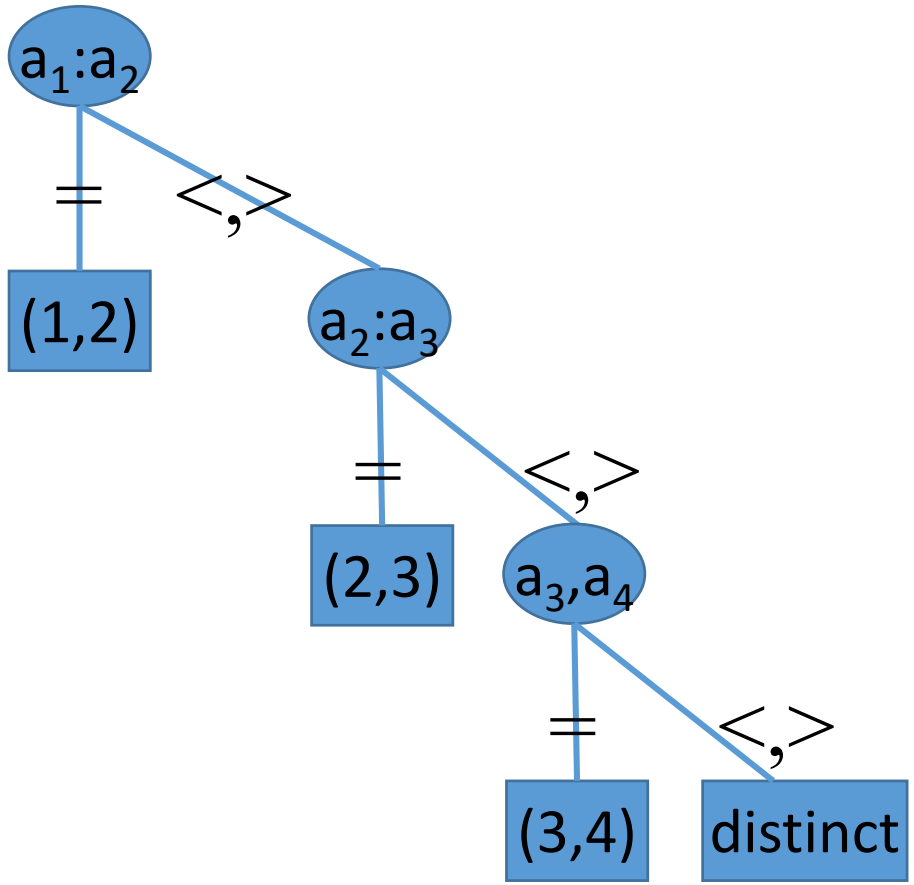
Question 6

- Given a sorted array of n numbers A[1..n]. You are asked to check if there are any duplicates in A[1..n]. You need to report either
  - (a) distinct (all elements are different)
  - (b) (i, j) such that A[i]=A[j].
- Assume comparison model, i.e., you have the procedure compare(i,j) which returns <, =, or >.
- What is a good lower bound on the number of comparisons to solve this problem?



Answer

Example: the best decision tree for this problem for
$A=[a_1, a_2, a_3, a_4]$.
The lower bound is 3.



Answer

Answer: the lower bound is n-1.

- By contrary, assume there exists a decision tree with depth $\leq$ n-2.
  - i.e. there exists an algorithm that runs in n-2 comparisons.
- This implies that there exists at least one adjacent pair (A[i], A[i+1]) which is not compared.
- We set A[i] = A[i+1].
- The algorithm fails to identify this pair of duplicates. We arrived at contradiction.

Question 7

Who is the Master of Algorithms pictured below?

- Donald Knuth
- Robert Tarjan
- Andrew Yao
- Tony Hoare

Andrew Yao

- Turing Award Winner
- Professor and Dean at Tsinghua University
- Well known for work in theoretical computer science, cryptography, pseudo-random number generation, and also Yao's minimax principle that is used for proving lower bounds on performance of randomized algorithms

Can we sort faster?

- Ω(n log n) is the lower bound for sorting.
- Can we break the lower bound?

- The sorting lower bound assumes we only compare elements.
- If we do more than comparison, we may be able to break the lower bound!

Sorting in linear time

- Assume there are k different elements. We can perform counting sort.

Counting sort: No comparisons between elements.

- ***Input***: $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- ***Output***: $B[1 \ldots n]$, sorted.
- ***Auxiliary storage***: $C[1 \ldots k]$.

3 steps in counting sort

1. Set C[i] be the number of elements equal i for i=1, 2, …, k.

2. Set C[i] be the number of elements smaller than or equal i for i=1, 2, …, k.

3. Move elements equal i to B[C[i-1]+1..C[i]] for i=1, 2, …, k.

Counting sort

Set C[i] be the number of elements equal i.

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1        ▷ C[i] = |{key = i}|
```

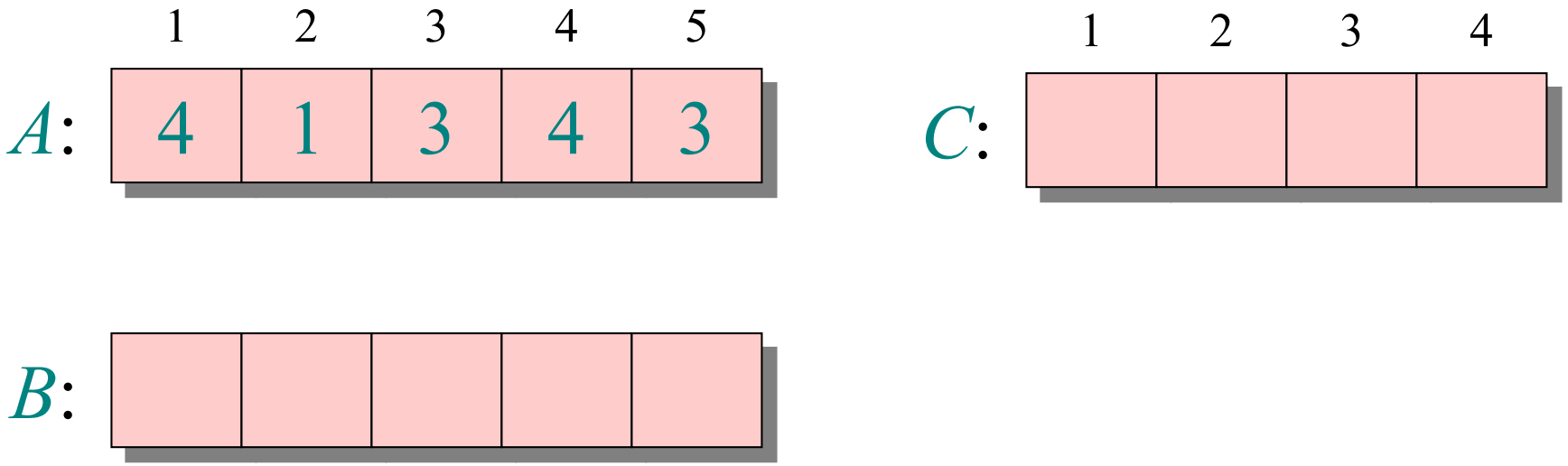
Set C[i] be the number of elements smaller than or equal i.

```
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]         ▷ C[i] = |{key ≤ i}|
```

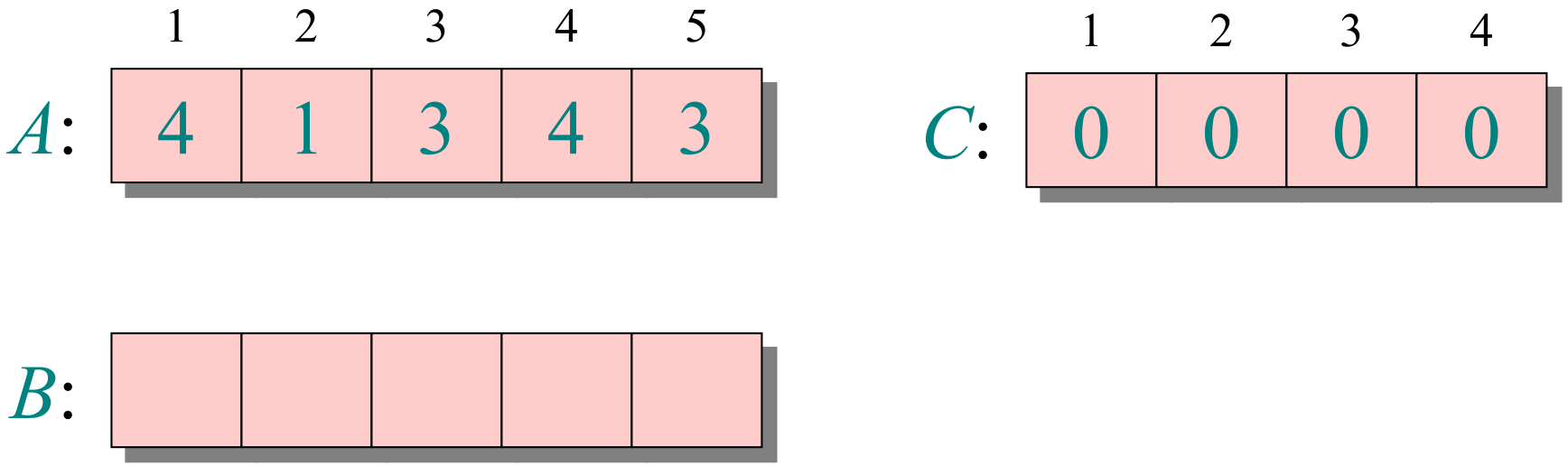
Move elements equal i to B[C[i-1]+1..C[i]].

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

Counting-sort example

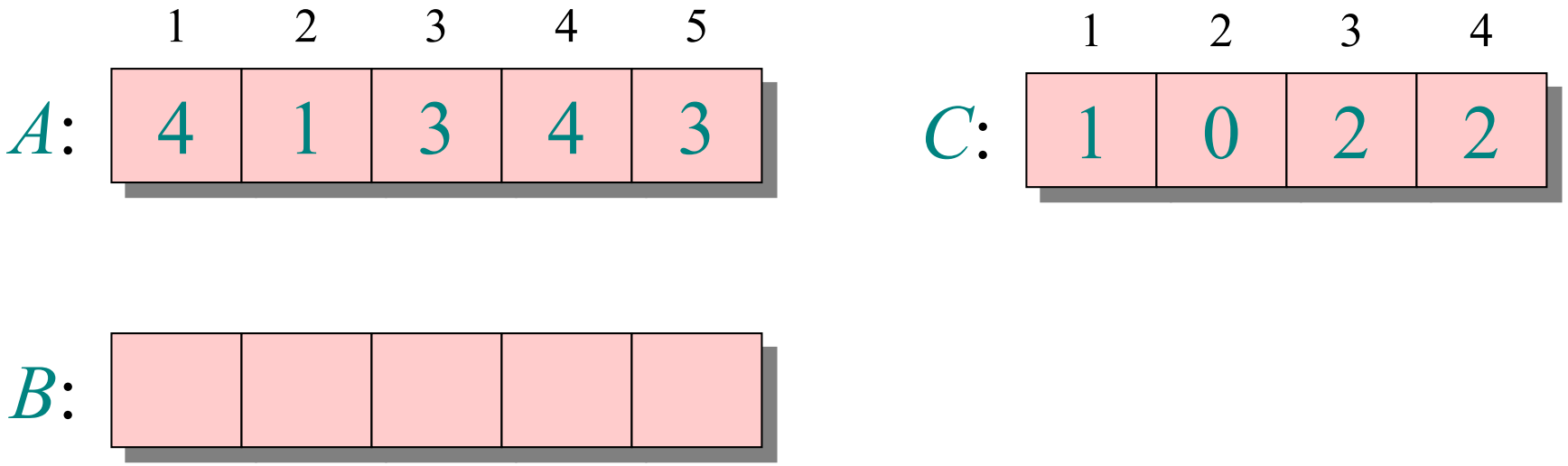


Phase 1 (I)



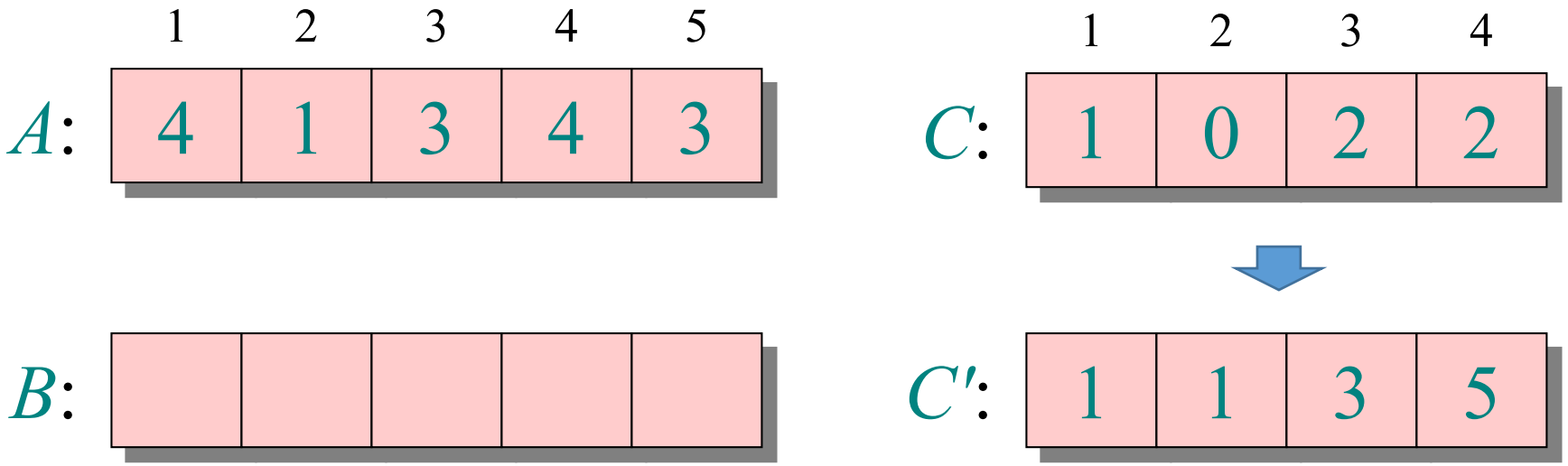
for $i \leftarrow 1$ **to** k
do $C[i] \leftarrow 0$

Phase 1 (II)



for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

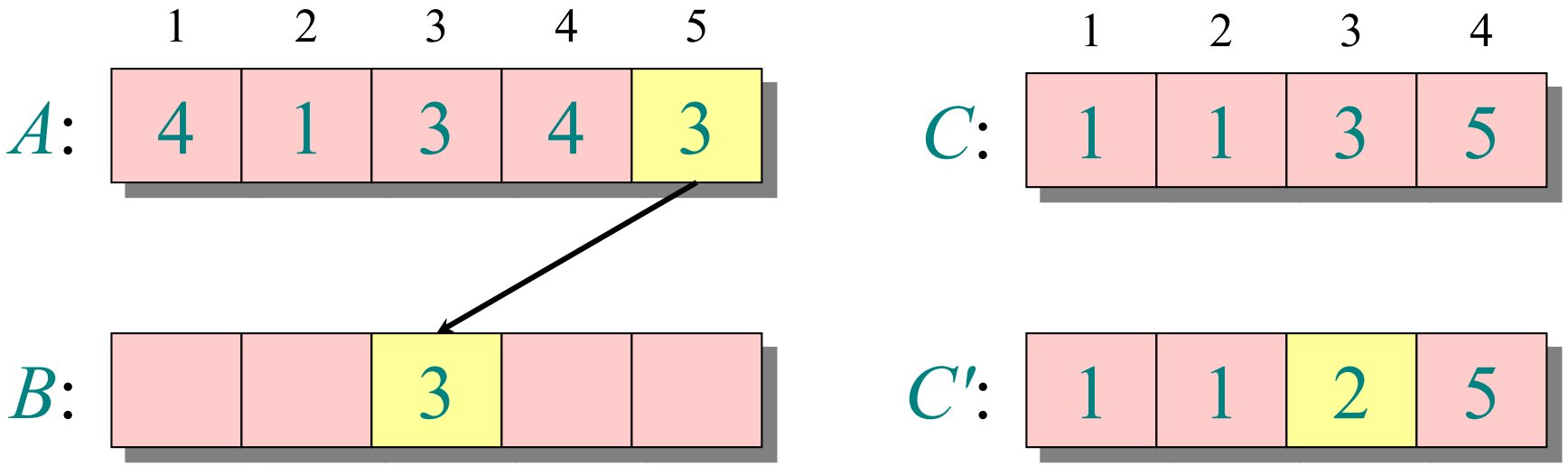
Phase 2



for $i \leftarrow 2$ **to** k

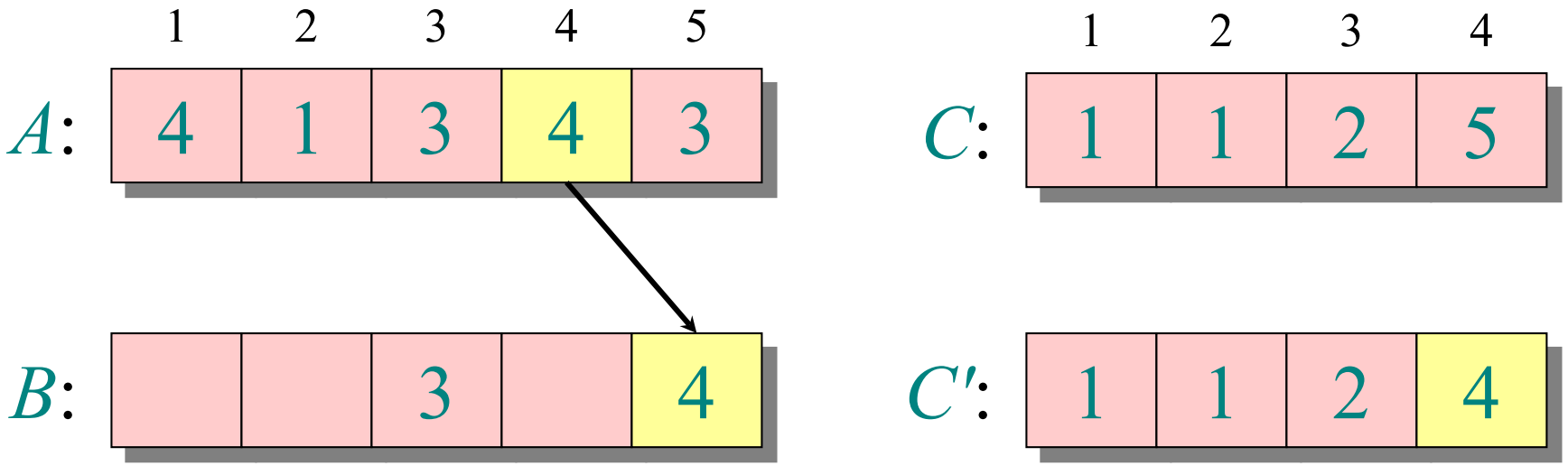
do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

Phase 3



for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Phase 3

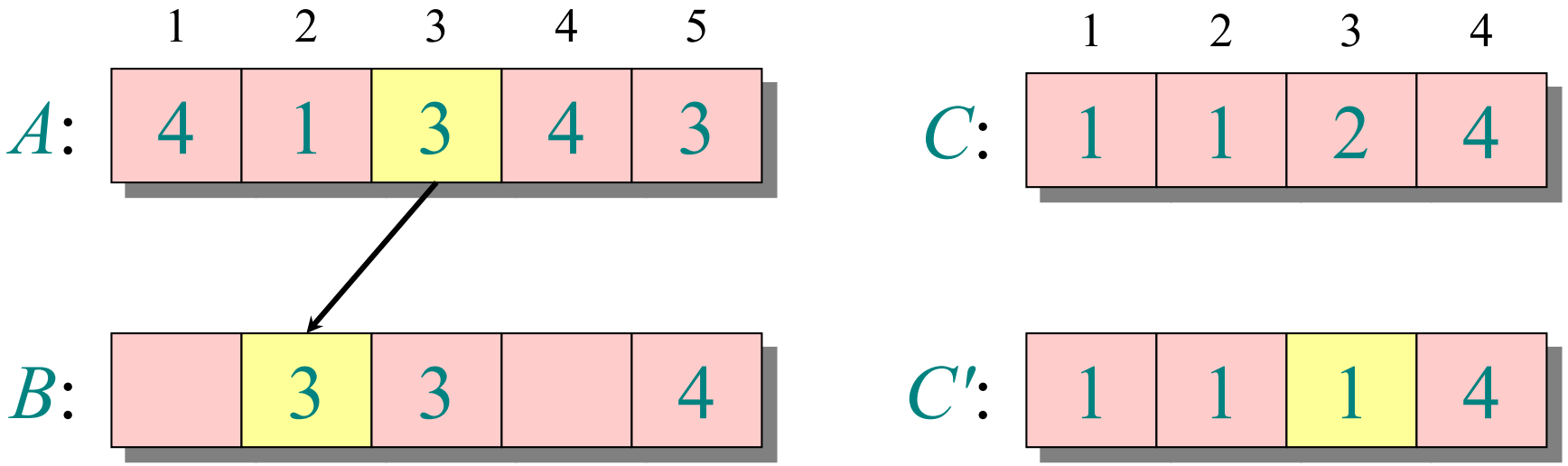


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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

Phase 3

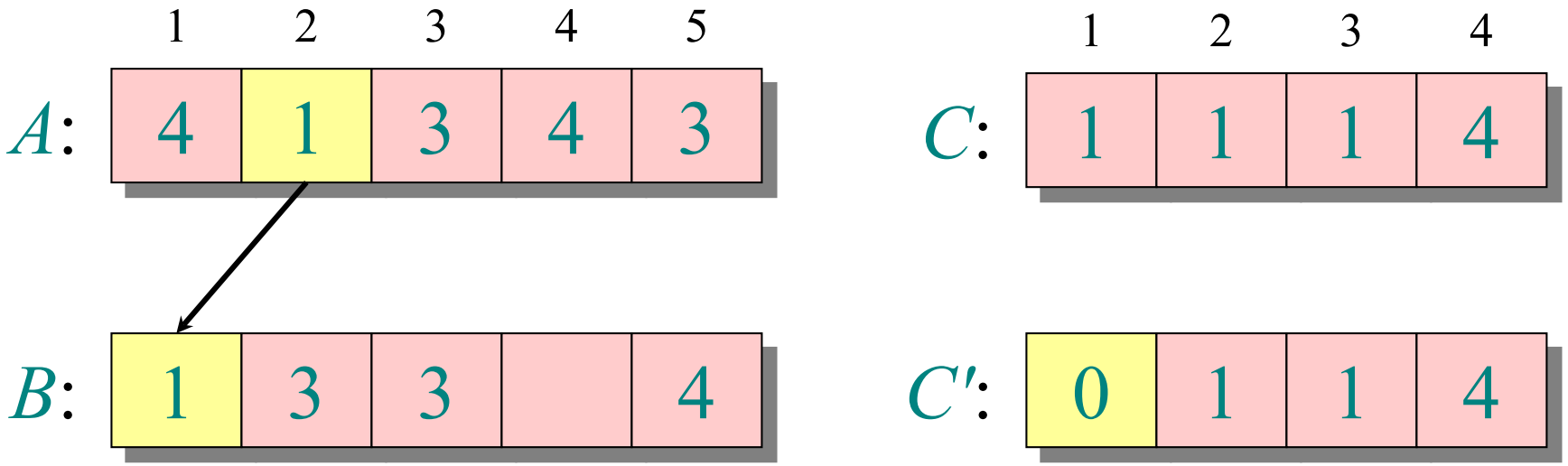


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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

Phase 3

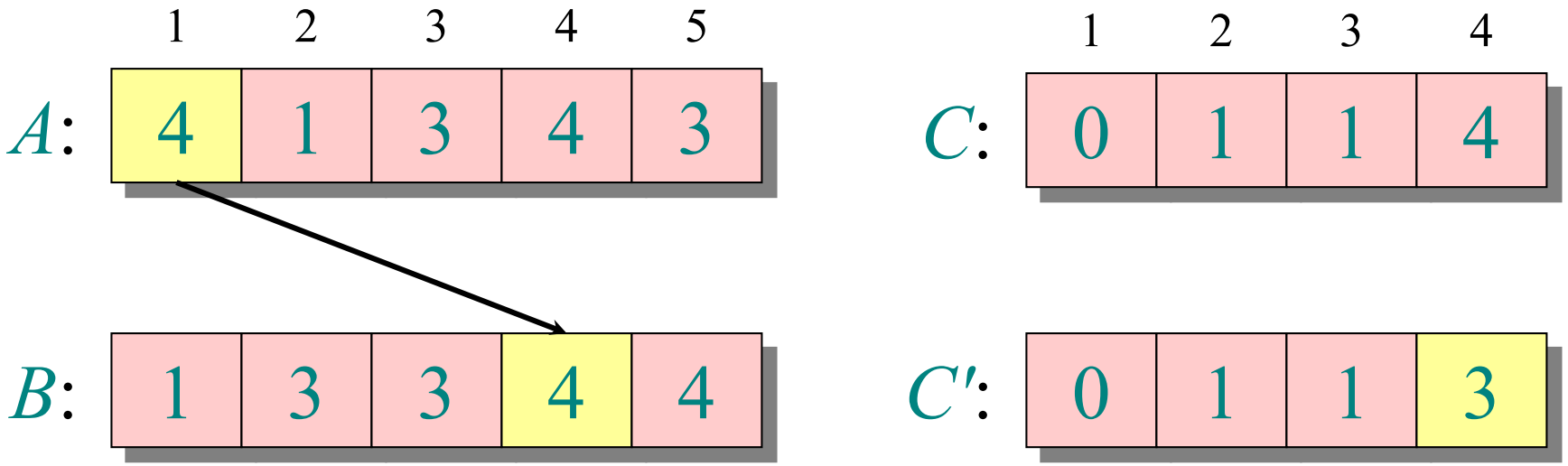


for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$

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

Phase 3



for $j \leftarrow n$ **downto** 1

do $B[C[A[j]]] \leftarrow A[j]$

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

Analysis

$\Theta(k)$ **for** $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

$\Theta(n)$ **for** $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ **for** $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ **for** $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Running time

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

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

Answer:

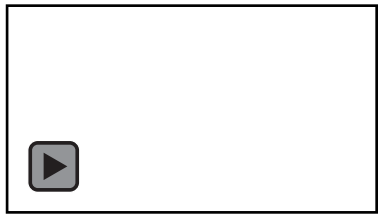
- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Is counting sort a stable sorting?

- Yes.

Question 8

- Consider an array containing n numbers.
- Suppose each number is in the range [1..n^3].
- We want to sort the array using counting sort.
- What is the time complexity?



Answer

- $k=n^3$.
- Counting sort runs in $O(n + k)$ time.
- Hence, the time complexity is $O(n + n^3) = O(n^3)$.

Radix sort

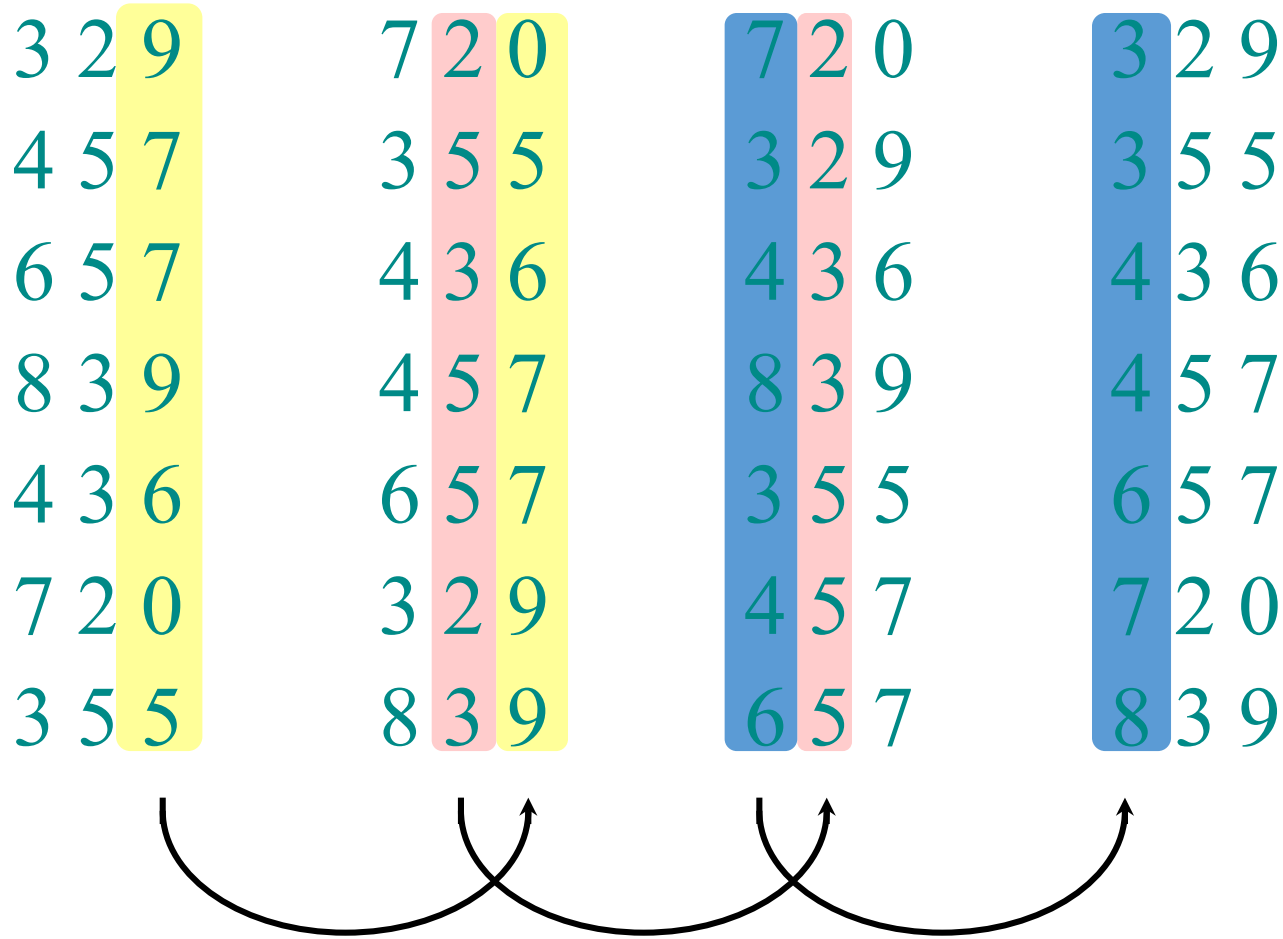
- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

Operation of radix sort

Suppose there are T digits.

for $t \leftarrow 1$ **to** T

do sort the t^{th} least significant digit by a **stable** sorting algorithm;

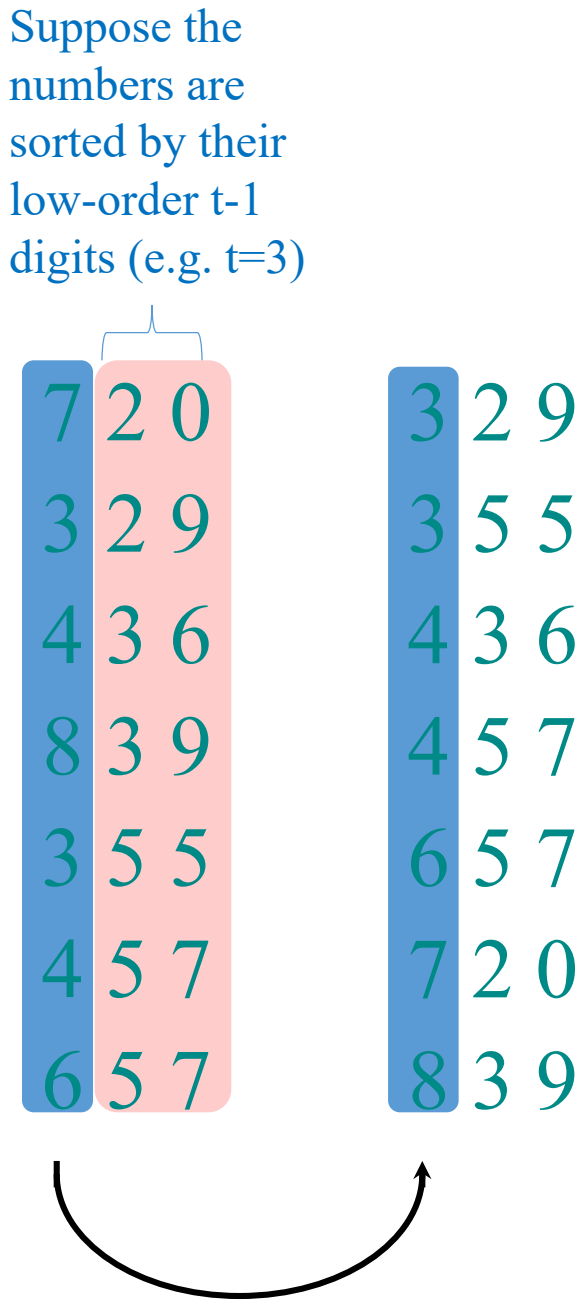


Example: We can use counting sort as the stable sorting algorithm.

Correctness of radix sort

- Proof by induction.
- P(t): The numbers are sorted by their low-order t digits.

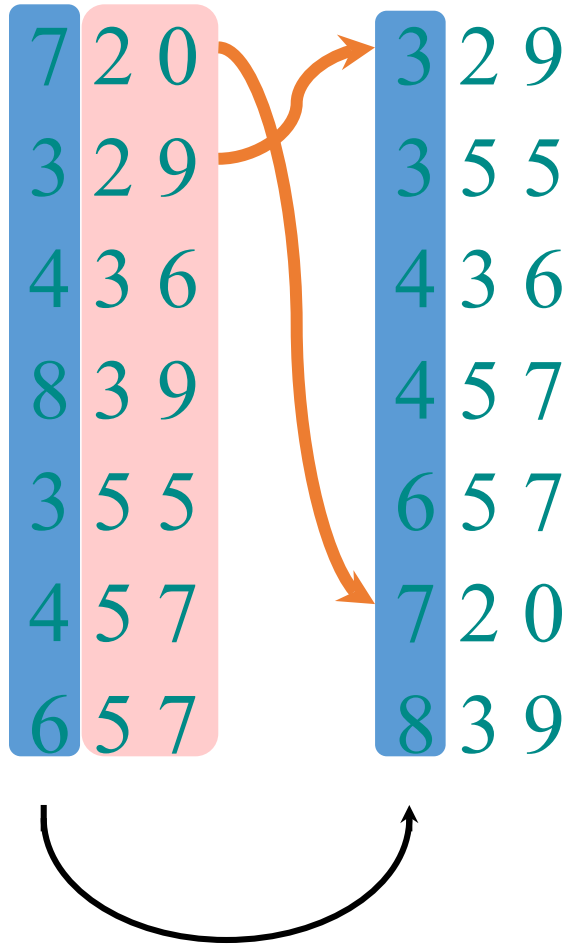
- Base case:
  - The radix sort algorithm sorts the 1st digit by stable sorting algorithm.
  - Hence, P(1) is true.
- Inductive case:
  - Assume P(t-1) is true.
  - We need to show P(t) is true.



Correctness of radix sort

Induction on digit position

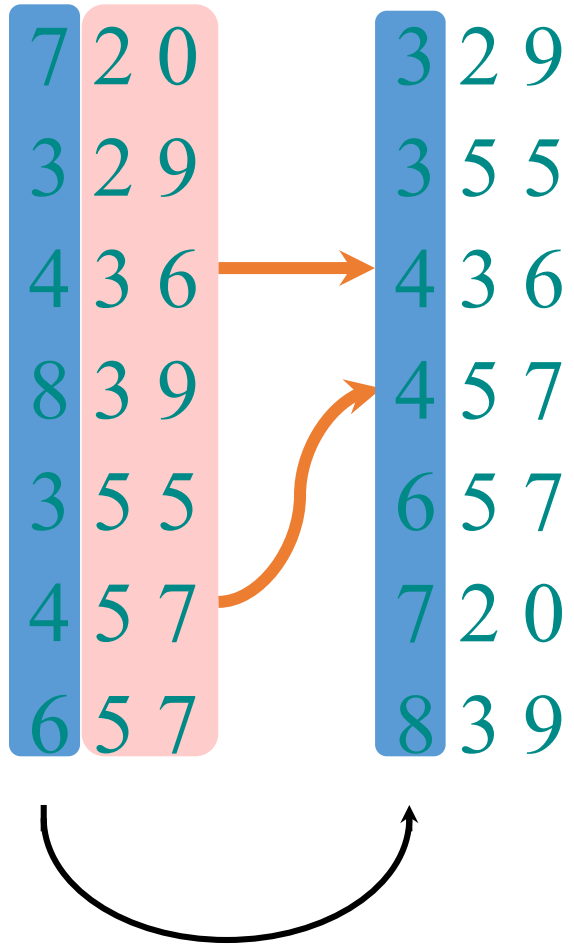
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



Question 9

Which of the following sorting algorithms (under typical implementations) will **not** work correctly as the auxiliary sorting algorithm (for sorting a digit) in radix sort?

- Insertion sort
- Merge sort
- Quicksort
- Counting sort
- Bubble sort

Solution

For use as an auxiliary sort to sort the digits in radix sort, the sorting algorithm needs to be stable.

Quicksort, under typical implementations, is not stable.

Analysis of radix sort

- Suppose counting sort is used as the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.
- Example: 32-bit word

- $r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

- To optimize the running time, how many passes (b/r) should we make?

Analysis (continued)

- **Recall:** Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 1 to k.
- If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have:
  - $T(n, b) = \Theta(\frac{b}{r}(n + 2^r))$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r > \lg n$, the time grows exponentially.

Choosing r

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

Choose r that minimizes $T(n,b)$ (by differentiation).

The optimal r is close to $\lg n$.
(Optimal r is slightly smaller than $\lg n$.)

Choosing $r = \lg n$ implies $T(n,b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.

When should we use radix sort?

- Radix sort is simple to code and maintain.
- In practice, radix sort is fast when the number of passes is small.

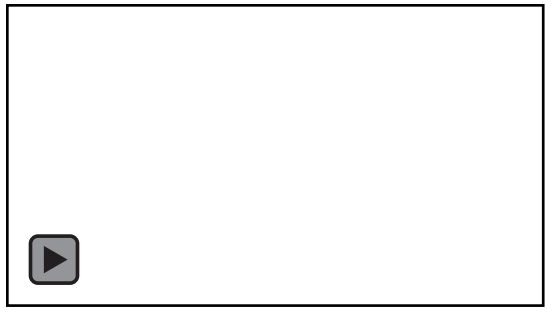
Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

- **Downside:** Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Question 10

True, False, Justify: You need to sort n numbers. You find out that the range of those numbers is in $[1,2^n]$. You should use radix sort in preference to merge sort.



Answer

Since the range is $[1,2^n]$, each number is of b-bits where b=n.

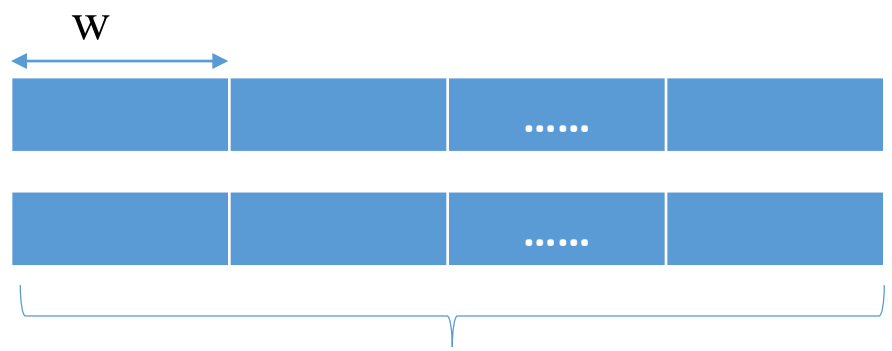
The runtime of radix sort is $\Theta\left(\frac{bn}{\lg n}\right)$. Hence, the runtime is $\Theta(n^2/\lg n)$.

Merge sort requires $\Theta(n \lg n)$ comparisons.

- If comparison requires constant time, the runtime is $\Theta(n \lg n)$.

In this case, the statement is **false**.

Answer



When n is big, say, $n=2^w$ where w is the word size.

(For today computer, w=64.)

Then, we cannot assume comparison of two n-bit numbers takes O(1) time.

- Each comparison can only compare w bits. It requires $\Theta(n/w)$ time to compare two n-bit numbers. Then, the runtime is $\Theta((n/w)\cdot n \lg n) = \Theta(n^2 \lg n / w)$.
- As $w=\lg n$, merge sort takes $\Theta(n^2)$ time.

In this case, the statement is **true**.

Conclusion

- This lecture covers the analysis of sorting.
- We describe the lower bound for comparison sort, which is $\Omega(n \lg n)$.
- We also discuss linear time sorting algorithms:
  - Counting sort and radix sort.

- When you design an algorithm to sort elements,
  - Select suitable sorting algorithms depends on the properties of the elements.
  - A number of issues:
    - Memory issue: InPlace sort
    - Need to preserve the original order of equal elements: Stable sort
    - Average case fast: QuickSort
    - Worst case fast (comparison model): worst case O(n log n)-time sort
    - Worst case fast (non-comparison model): radix sort

Acknowledgement

- The slides are modified from
  - the slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - the slides from Prof. Lee Wee Sun