# CS 240 – Data Structures and Data Management

## Module 3: Sorting and Randomized Algorithms

T. Biedl   E. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

**David R. Cheriton School of Computer Science, University of Waterloo**

Winter 2021

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# Selection Problem

- Given array $A$ of $n$ numbers, and $0 \leq k < n$, find the element that would be at position $k$ if $A$ was sorted
  - 'select $k$'
  - $k$ elements are smaller or equal, $n - 1 - k$ elements are larger or equal

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 80 | 90 | 20 | 40 | 70 |

*select*$(2)$

- Special case: *median finding* $(k = \lfloor \frac{n}{2} \rfloor)$
- Heap-based selection can be done in $\Theta(n + k \log n)$
  - this is $\Theta(n \log n)$ for median finding
    - the same cost as our best sorting algorithms
- **Question**: can we do selection in linear time?
  - yes, with quick-select (average case analysis)
  - subroutines for quick-select also useful for sorting algorithms

# Crucial Subroutines

| 0 | 1 | 2 | 3 | $p = 4$ | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | $v$ =**50** | 80 | 90 | 20 | 40 | 70 |

- *quick-select* and related algorithm *quick-sort* rely on two subroutines
  - *choose-pivot*$(A)$
    - return an index $p$ in $A$
    - use *pivot-value* $v \leftarrow A[p]$ to rearrange the array

| 0 | 1 | 2 | 3 | 4 | $i = 5$ | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 10 | 0 | 20 | 40 | $v$ =**50** | 60 | 80 | 90 | 70 |

- *partition* $(A, p)$ rearranges $A$ so that
  - all items in $A\ [0, \ldots, i-1]$ are $\leq v$
  - pivot-value $v$ is in $A[i]$
  - all items in $A\ [i+1, \ldots, n-1]$ are $\geq v$
  - index $i$ is called *pivot-index* $i$
  - *partition*$(A, p)$ returns *pivot-index* $i$
    - $i$ is a correct location of $v$ in sorted $A$
    - if we were interested in select$(i)$, then $v$ would be the answer

# Choosing Pivot

- Simplest idea for *choose-pivot*
  - always select rightmost element in array

*choose-pivot1*$(A)$

**return** $A$.size() − 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $p = 9$ |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 80 | 90 | 20 | 40 | $v =$ **70** |

- Will consider more sophisticated ideas later

# Partition Algorithm

*partition*$(A, p)$

$A$: array of size $n$, $p$: integer s.t. $0 \leq p < n$

create empty lists $small, equal$ and $large$

$v \leftarrow A[p]$

**for** each element $x$ **in** $A$

**if** $x < v$ **then** $small.append(x)$

**else if** $x > v$ **then** $large.append(x)$

**else** $equal.append(x)$

$i \leftarrow small.size$

$j \leftarrow equal.size$

overwrite $A[0 \ldots i-1]$ by elements in $small$

overwrite $A[i \ldots i+j-1]$ by elements in $equal$

overwrite $A[i+j \ldots n-1]$ by elements in $large$

**return** *i*

- Easy linear-time implementation using extra (auxiliary) $\Theta(n)$ space
- More challenging: partition *in-place*, i.e. $O(1)$ auxiliary space

# Efficient In-Place partition (Hoare)

$i = -1$ $j = 9$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 80 | 90 | 20 | 40 | $v$=70 |

$i = 5$ $j = 8$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | **80** | **90** | **20** | **40** | $v$=70 |

$i = 5$ $j = 8$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | **40** | 90 | 20 | **80** | $v$=70 |

$i = 6$ $j = 7$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 40 | **90** | **20** | 80 | $v$=70 |

$i = 6$ $j = 7$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 40 | **20** | **90** | 80 | $v$=70 |

$j = 6$ $i = 7$

almost done, just swap with pivot $v$

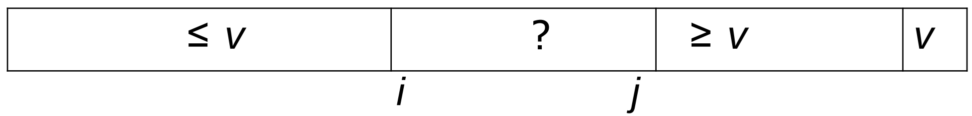
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 40 | 20 | 90 | 80 | $v$=70 |

$j = 6$ $i = 7$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30 | 60 | 10 | 0 | 50 | 40 | 20 | $v$=70 | 80 | 90 |

# Efficient In-Place partition (Hoare)

- **Idea Summary**: Keep swapping the outer-most wrongly-positioned pairs




- One possible implementation

$$\textbf{do } i \leftarrow i+1 \textbf{ while } i < n \textbf{ and } A[i] \leq v$$
$$\textbf{do } j \leftarrow j-1 \textbf{ while } j > 0 \textbf{ and } A[j] \geq v$$

- More efficient (for quickselect and quicksort) when many repeating elements

$$\textbf{do } i \leftarrow i+1 \textbf{ while } i < n \textbf{ and } A[i] < v$$
$$\textbf{do } j \leftarrow j-1 \textbf{ while } j > 0 \textbf{ and } A[j] > v$$

- Can simplify the loop bounds

$$\textbf{do } i \leftarrow i+1 \textbf{ while } A[i] < v$$
$$\textbf{do } j \leftarrow j-1 \textbf{ while } j \geq i \textbf{ and } A[j] > v$$

# Efficient In-Place partition (Hoare)

*partition* $(A, p)$

$A$: array of size $n$

$p$: integer s.t. $0 \le p < n$

*swap*$(A[n-1],\ A[p])$
$i \leftarrow -1, \qquad j \leftarrow n-1, \qquad v \leftarrow A[n-1]$
**loop**
**do** $i \leftarrow i+1$ **while** $A[i] < v$
**do** $j \leftarrow j-1$ **while** $j \ge i$ **and** $A[j] > v$
**if** $i \ge j$ **then break**
**else** *swap*$(A[i],\ A[j])$
**end loop**
*swap*$(A[n-1],\ A[i])$
**return** $i$

- Running time is $\Theta(n)$

# Efficient In-Place partition (Hoare)

*partition* $(A, p)$

$A$: array of size $n$

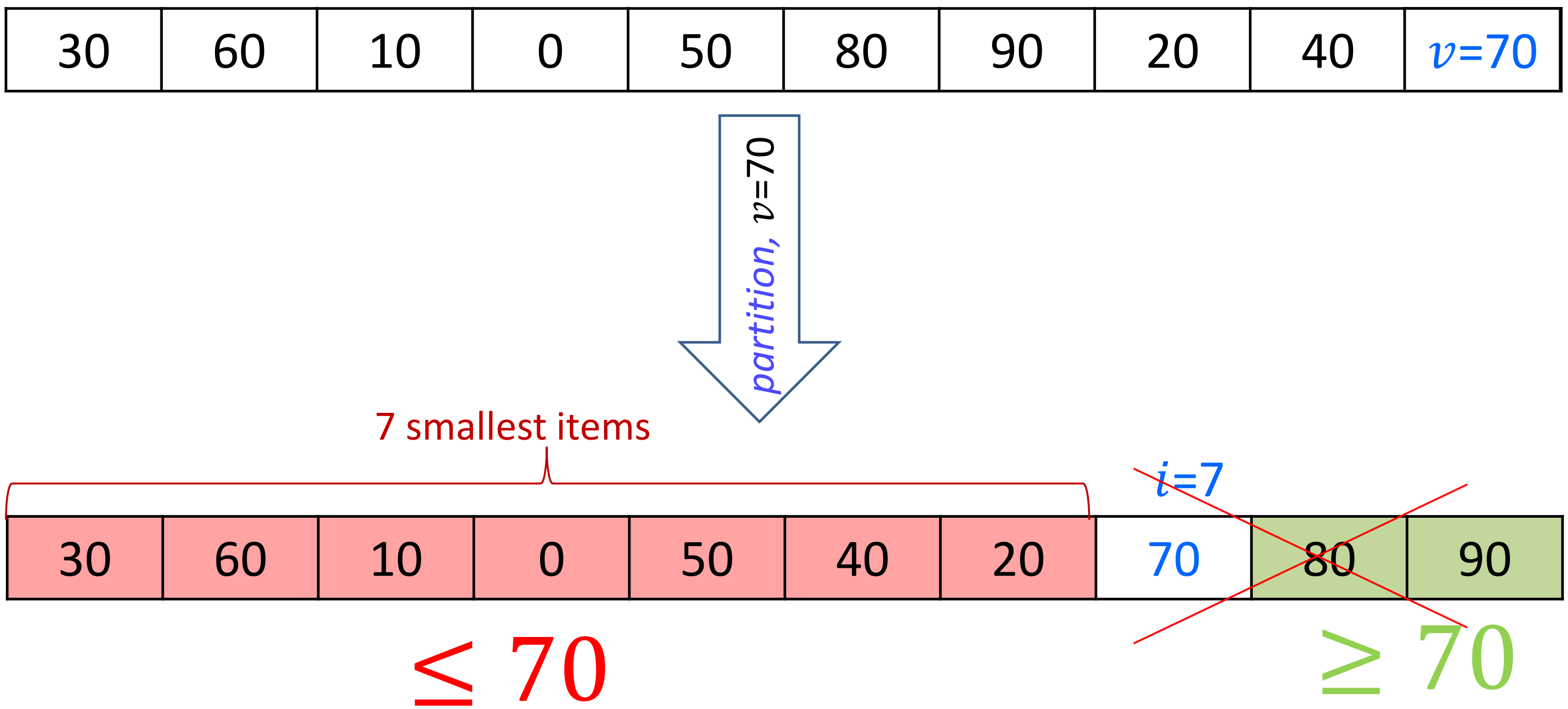
$p$: integer s.t. $0 \le p < n$

*swap*$(A[n-1],\ A[p])$
$i \leftarrow -1, \qquad j \leftarrow n-1, \qquad v \leftarrow A[n-1]$
**loop**
  **do** $i \leftarrow i + 1$ **while** $A[i] < v$
  **do** $j \leftarrow j - 1$ **while** $j \ge i$ **and** $A[j] > v$
  **if** $i \ge j$ **then break**
  **else** *swap*$(A[i],\ A[j])$
**end loop**
*swap*$(A[n-1],\ A[i])$
**return** $i$

- Running time is $\Theta(n)$

# Quick Select Algorithm

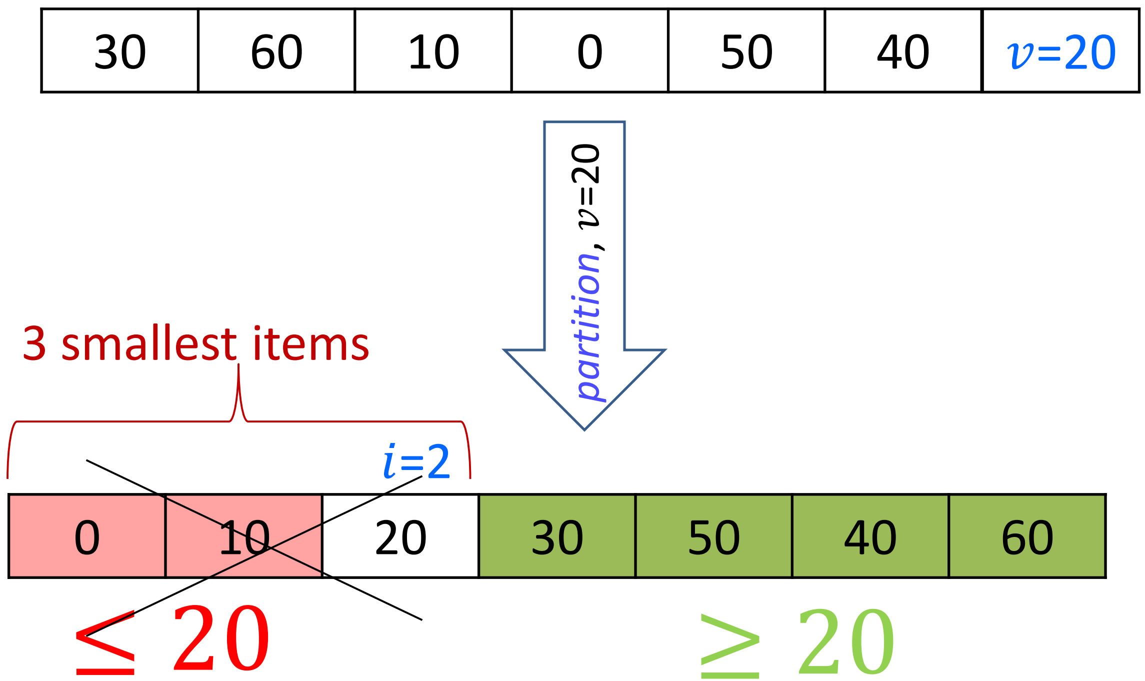
- Find item that would be in $A[k]$ if $A$ was sorted
- Similar to quick-sort, but recurse only on one side ("quick-sort with pruning")
- Example: select($k = 4$)
  - [the correct answer is 40  in this case]




- $i > k$, search recursively in the left side to select $k$

# Quick Select Algorithm

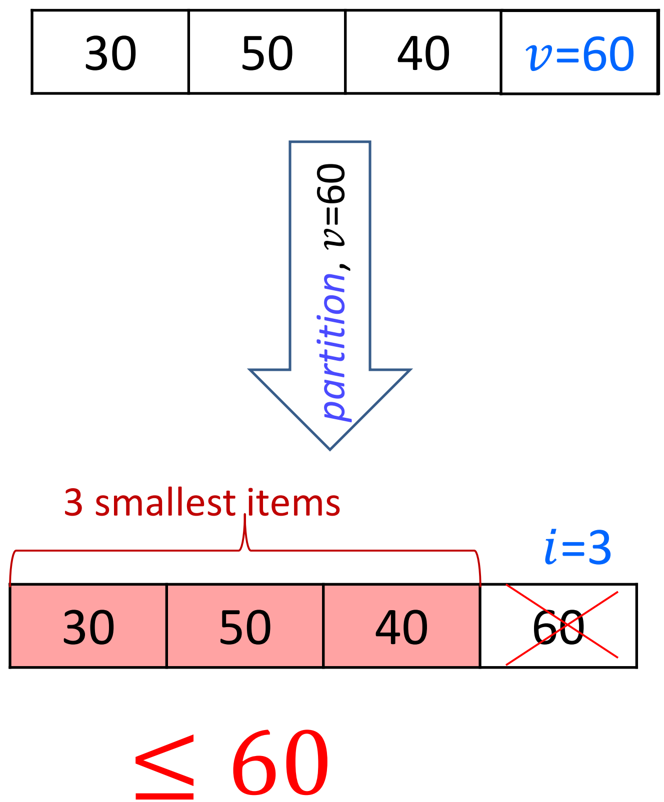
- Example continued: select($k = 4$)




- $i < k$, search recursively on the right, select $\boldsymbol{k - (i+1)}$
  - $k = 1$ in our example

# Quick Select Algorithm

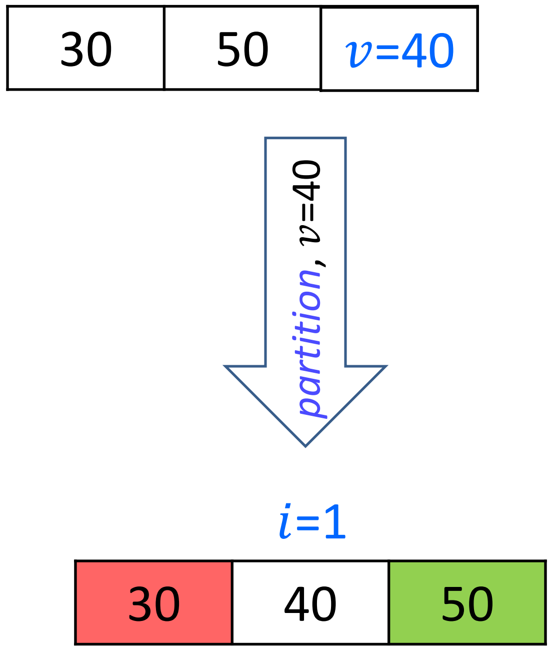
- Example continued: select($k = 1$)




- $i > k$, search on the left to select $k$

# Quick Select Algorithm

- Example continued: select($k = 1$)




- $i = k$, found our item, done!
- In our example, we got to subarray of size 3
- Often stop much sooner than that
    - running time?

# QuickSelect Algorithm

*quick-select1*$(A, k)$

$A$: array of size $n$, $k$: integer s.t. $0 \leq k < n$

$p \leftarrow$ *choose-pivot1*$(A)$

$i \leftarrow$ *partition*$(A, p)$

**if** $i = k$ **then**

  **return** $A[i]$

**else if** $i > k$ **then**

  **return** *quick-select1*$(A[0, 1, \ldots, i-1],\ k)$

**else if** $i < k$ **then**

  **return** *quick-select1*$(A[i+1, \ldots, n-1],\ k-(i+1))$

- **Best case**
  - first chosen pivot could have pivot-index $k$
  - no recursive calls, total cost $\Theta(n)$
- **Worst case**: recurrence equation $T(n) = \begin{cases} cn + T(n-1) & n > 1 \\ c & n = 1 \end{cases}$

# QuickSelect Algorithm

- **Worst case**: recurrence equation $T(n) = \begin{cases} cn + T(n-1) & n > 1 \\ c & n = 1 \end{cases}$
- Solution: repeatedly expand until we see a pattern forming

$$T(n) = cn + T(n-1)$$

$$T(n-1) = \underbrace{c(n-1) + T(n-2)}$$

$$T(n) = cn + c(n-1) + T(n-2)$$

after 1 expansion

$$T(n-2) = \underbrace{c(n-2) + T(n-3)}$$

$$T(n) = cn + c(n-1) + c(n-2) + T(n-3)$$

after 2 expansions

- After $i$ expansions

$$T(n) = cn + c(n-1) + c(n-2) + \cdots + c(n-i) + T(n-(i+1))$$

- Stop expanding when get to base case $T(n-(i+1)) = T(1)$
- Happens when $n-(i+1) = 1$, or, rewriting, $i = n-2$
- Thus $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + T(1)$

$$= cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + c$$

$$= c(n + (n-1) + \cdots + 2 + 1) \in \Theta(n^2)$$

# Average-Case Analysis of *quick-select1*

$$T^{avr}(n) = \frac{1}{\text{\# instances of size } n} \sum_{I:size(I)=n} T(I)$$

**infinitely many**

- Need to make some assumptions
- First assumption
  - all input numbers are distinct
  - this assumption is just for simpler analysis, can prove the same thing without this assumption

# Average-Case Analysis of *quick-select1*

- QuickSelect is *comparison-based*
  - only cares if $A[i] < A[j]$ for $i, j$
  - does not care what the actual values of $A[i]$, $A[j]$ are

$I_1$

| | | | |
|---|---|---|---|
| 30 | 60 | 0 | 10 |

$I_2$

| | | | |
|---|---|---|---|
| 20 | 50 | 10 | 15 |

- QuickSelect makes exactly the same sequences of steps on $I_1$ and $I_2$
  - therefore $T(I_1) = T(I_2)$
- Any comparison based algorithm has exactly the same running time for arrays that have the same relative order of elements, regardless of actual array values
- Second assumption: we are sorting integers $0, \ldots, n-1$
  - now there are $n!$ possible input instances $I$
  - more formal proof uses *sorting permutations*
    - permutation $\pi$ for which $A[\pi(0)] \leq A[\pi(1)] \leq \ldots \leq A[\pi(n-1)]$
    - for $I_1$ (and $I_2$) sorting permutation is $\pi = (2, 3, 0, 1)$
    - assume *each sorting permutation is equally likely*
    - $n!$ possible permutations

# Average-Case Analysis of *quick-select1*

$$T^{avr}(n) = \frac{1}{\#\text{ instances of size } n} \sum_{I:size(I)=n} T(I)$$

- Example for $n = 3$, using all the assumptions

$$T^{avr}(3) = \frac{1}{3!}(T(\{0,1,2\}) + T(\{0,2,1\}) + T(\{1,0,2\}) + T(\{1,2,0\}) + T(\{2,0,1\}) + T(\{2,1,0\}))$$

# Average-Case Analysis of *quick-select1*

- Recall that pivot is last array element
- Pivot index is equal to pivot value due to assuming we sort $0, \dots, n-1$

$A$

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 2 | 3 | 0 | $v$=1 |

for $v$=1, pivot index $i = 1$

- Partition sum over different pivot indexes

$$T^{avr}(n) = \frac{1}{n!} \sum_{I:Size(I)=n} T(I) \qquad = \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\substack{I:size(I)=n, \\ pivot\ is\ i}} T(I)$$

- Example for $n = 3$

$$T^{avr}(3) = \frac{1}{3!}(T(\{0,1,2\}) + T(\{0,2,1\}) + T(\{1,0,2\}) + T(\{1,2,0\}) + T(\{2,0,1\}) + T(\{2,1,0\}))$$

$$\begin{aligned} T^{avr}(3) = \tfrac{1}{3!}\ & (T(\{1,2,\mathbf{0}\}) + T(\{2,1,\mathbf{0}\})) + \\ & (T(\{0,2,\mathbf{1}\}) + T(\{2,0,\mathbf{1}\})) + \\ & (T(\{0,1,\mathbf{2}\}) + T(\{1,0,\mathbf{2}\})) \end{aligned}$$

# Average-Case Analysis of *quick-select1*

- Partition sum over different pivots $T^{avr}(n) = \frac{1}{n!}\sum_{i=0}^{n-1} \sum_{\substack{I:size(I)=n,\\ pivot\ is\ i}} T(I)$

- There are $(n-1)!$ input instances $I$ with pivot index $i$

| $A$ | | | | | $\boldsymbol{i}$ |
|---|---|---|---|---|---|
| | choice of $n-1$ items: anything but $i$ | choice of $n-2$ items: anything but $i$ and $A[0]$ | … | 'choice' of 1 items | no choice |

- One can show (will only hint at the proof with example for $n = 4, i = 1$)

$$\sum_{\substack{I:size(I)=n,\\ pivot\ is\ \boldsymbol{i}}} T(I) \le (n-1)!\,cn + (n-1)!\,max\{T^{avr}(i)\,, T^{avr}(n-i-1)\,\}$$

- Therefore $T^{avr}(n) \le cn + \frac{1}{n}\sum_{i=0}^{n-1} max\{T^{avr}(i)\,, T^{avr}(n-i-1)\,\}$

# Average-Case Analysis of *quick-select1*

- Let $n = 4, i = 1$

$$\sum_{\substack{I:size(I)=4,\\ pivot\ is\ 1}} T(I) = T(\{0,2,3,\mathbf{1}\}) + T(\{0,3,2,\mathbf{1}\}) + T(\{2,0,3,\mathbf{1}\}) + T(\{2,3,0,\mathbf{1}\}) + T(\{3,0,2,\mathbf{1}\}) + T(\{3,2,0,\mathbf{1}\})$$

- Total work is proportional to comparisons, will count comparisons

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| comparisons to partition: | $3$ | $3$ | $3$ | $3$ | $3$ | $3$ | Total: $3(3)!$ |
| instances | $\{0,2,3,\mathbf{1}\}$ | $\{0,3,2,\mathbf{1}\}$ | $\{2,0,3,\mathbf{1}\}$ | $\{2,3,0,\mathbf{1}\}$ | $\{3,0,2,\mathbf{1}\}$ | $\{3,2,0,\mathbf{1}\}$ | |
| partitions (assume stable order) | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{3,2\}$ | |

# Average-Case Analysis of *quick-select1*

- Let $n = 4, i = 1$

$$\sum_{\substack{I:size(I)=4,\\ pivot\ is\ 1}} T(I) = \begin{array}{l} T(\{0,2,3,\mathbf{1}\}) + T(\{0,3,2,\mathbf{1}\}) \\ + T(\{2,0,3,\mathbf{1}\}) + T(\{2,3,0,\mathbf{1}\}) \\ + T(\{3,0,2,\mathbf{1}\}) + T(\{3,2,0,\mathbf{1}\}) \end{array}$$

- Total work is proportional to comparisons, will count comparisons

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| comparisons to partition: | 3 | 3 | 3 | 3 | 3 | 3 | Total: $3(3)!$ |
| instances | $\{0,2,3,\mathbf{1}\}$ | $\{0,3,2,\mathbf{1}\}$ | $\{2,0,3,\mathbf{1}\}$ | $\{2,3,0,\mathbf{1}\}$ | $\{3,0,2,\mathbf{1}\}$ | $\{3,2,0,\mathbf{1}\}$ | |
| partitions | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{3,2\}$ | |

Case 1: $k > i$

$$T(\{2,3\}) + T(\{3,2\}) + T(\{2,3\}) + T(\{2,3\}) + T(\{3,2\}) + T(\{3,2\})$$

$$= T(\{0,1\}) + T(\{1,0\}) + T(\{0,1\}) + T(\{0,1\}) + T(\{1,0\}) + T(\{1,0\})$$

since only relative order matters

swap

# Average-Case Analysis of *quick-select1*

- Let $n = 4, i = 1$

$$\sum_{\substack{I:size(I)=\mathbf{4},\\ pivot\ is\ \mathbf{1}}} T(I) = \begin{aligned} &T(\{0,2,3,\mathbf{1}\}) + T(\{0,3,2,\mathbf{1}\})\\ &+ T(\{2,0,3,\mathbf{1}\}) + T(\{2,3,0,\mathbf{1}\})\\ &+ T(\{3,0,2,\mathbf{1}\}) + T(\{3,2,0,\mathbf{1}\})\end{aligned}$$

- Total work is proportional to comparisons, will count comparisons

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| comparisons to partition: | $3$ | $3$ | $3$ | $3$ | $3$ | $3$ | Total: $3(3)!$ |
| instances | $\{0,2,3,\mathbf{1}\}$ | $\{0,3,2,\mathbf{1}\}$ | $\{2,0,3,\mathbf{1}\}$ | $\{2,3,0,\mathbf{1}\}$ | $\{3,0,2,\mathbf{1}\}$ | $\{3,2,0,\mathbf{1}\}$ | |
| partitions | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{3,2\}$ | |

Case 1: $k > i$

$$T(\{2,3\}) + T(\{3,2\}) + T(\{2,3\}) + T(\{2,3\}) + T(\{3,2\}) + T(\{3,2\})$$

$$= \underbrace{T(\{0,1\}) + T(\{1,0\})}_{2!\,T^{avr}(2)} + \underbrace{T(\{0,1\}) + T(\{1,0\})}_{2!\,T^{avr}(2)} + \underbrace{T(\{0,1\}) + T(\{1,0\})}_{2!\,T^{avr}(2)}$$

since only relative order matters

Total recursive comparisons $\dfrac{3!}{2!}\,2!\,T^{avr}(2) = 3!\,T^{avr}(2)$

# Average-Case Analysis of *quick-select1*

- Let $n = 4, i = 1$

$$\sum_{\substack{I:size(I)=4,\\ pivot\ is\ 1}} T(I) = T(\{0,2,3,\mathbf{1}\}) + T(\{0,3,2,\mathbf{1}\}) + T(\{2,0,3,\mathbf{1}\}) + T(\{2,3,0,\mathbf{1}\}) + T(\{3,0,2,\mathbf{1}\}) + T(\{3,2,0,\mathbf{1}\})$$

- Total work is proportional to comparisons, will count comparisons

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| comparisons to partition: | 3 | 3 | 3 | 3 | 3 | 3 | Total: $3(3)!$ |
| instances | $\{0,2,3,\mathbf{1}\}$ | $\{0,3,2,\mathbf{1}\}$ | $\{2,0,3,\mathbf{1}\}$ | $\{2,3,0,\mathbf{1}\}$ | $\{3,0,2,\mathbf{1}\}$ | $\{3,2,0,\mathbf{1}\}$ | |
| partitions | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{3,2\}$ | |

Case 2: $k < i$

$$\underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)}$$

Total recursive comparisons $\dfrac{3!}{1!}1!\,T^{avr}(1) = 3!\,T^{avr}(1)$

[Case 1, total recursive comparisons: $= 3!\,T^{avr}(2)$ ]

Combining both cases, total recursive comparisons : $\leq 3!\max\{T^{avr}(1), T^{avr}(2)\}$

Adding comparisons to partition: $\leq 3(3)! + 3!\max\{T^{avr}(1), T^{avr}(2)\}$

# Average-Case Analysis of *quick-select1*

- Let $n = 4, i = 1$

$$\sum_{\substack{I:size(I)=4,\\ pivot\ is\ 1}} T(I) = T(\{0,2,3,\mathbf{1}\}) + T(\{0,3,2,\mathbf{1}\}) + T(\{2,0,3,\mathbf{1}\}) + T(\{2,3,0,\mathbf{1}\}) + T(\{3,0,2,\mathbf{1}\}) + T(\{3,2,0,\mathbf{1}\})$$

- Total work is proportional to comparisons, will count comparisons

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| comparisons to partition: | $3$ | $3$ | $3$ | $3$ | $3$ | $3$ | Total: $3(4-1)!$ |
| instances | $\{0,2,3,\mathbf{1}\}$ | $\{0,3,2,\mathbf{1}\}$ | $\{2,0,3,\mathbf{1}\}$ | $\{2,3,0,\mathbf{1}\}$ | $\{3,0,2,\mathbf{1}\}$ | $\{3,2,0,\mathbf{1}\}$ | |
| partitions | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{2,3\}$ | $\{0\}$ $\{3,2\}$ | $\{0\}$ $\{3,2\}$ | |

Case 2: $k < i$

$$\underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)} + \underbrace{T(\{0\})}_{1!\,T^{avr}(1)}$$

Total recursive comparisons $\frac{3!}{\phantom{1}}1!\,T^{avr}(1) = 3!\,T^{avr}(1)$

$$\boxed{\sum_{\substack{I:size(I)=n,\\ pivot\ is\ i}} T(I) \le (n-1)!\,cn + (n-1)!\,max\{T^{avr}(i)\,,T^{avr}(n-i-1)\,\}}$$

Comb

Adding comparisons to partition: $\le 3(3)! + 3!\max\{T^{avr}(1), T^{avr}(2)\}$

# Average-Case Analysis of *quick-select1*

$$T(n) \leq c \cdot n + \frac{1}{n}\sum_{i=0}^{n-1} max\{T(i), T(n-i-1)\}$$

**Theorem**: $T(n) \in O(n)$

**Proof**:

- will prove $T(n) \leq 4cn$ by induction on $n$
- base case, $n = 1$: $T(1) = c \leq 4c \cdot 1$
- induction hypothesis: assume $T(m) \leq 4cm$ for all $m < n$
- need to show $T(n) \leq 4cn$

induction hypothesis applies to each one of these

$$T(n) \leq c \cdot n + \frac{1}{n}\sum_{i=0}^{n-1} max\{T(i), T(n-i-1)\}$$

$$\leq c \cdot n + \frac{1}{n}\sum_{i=0}^{n-1} max\{4ci, 4c(n-i-1)\}$$

$$\leq c \cdot n + \frac{4c}{n}\sum_{i=0}^{n-1} max\{i, n-i-1\}$$

# Average-Case Analysis of *quick-select1*

exactly what we need for the proof

**Proof**: (cont.) $T(n) \le c \cdot n + \frac{4c}{n} \sum_{i=0}^{n-1} max\{i, n-i-1\} \le c \cdot n + \frac{4c}{n} \cdot \frac{3}{4} n^2 = 4cn$

$$\sum_{i=0}^{n-1} max\{i, n-i-1\} = \sum_{i=0}^{\frac{n}{2}-1} max\{i, n-i-1\} + \sum_{i=\frac{n}{2}}^{n-1} max\{i, n-i-1\}$$

$$= max\{0, \underline{n-1}\} + max\{1, \underline{n-2}\} + max\{2, \underline{n-3}\} + \cdots + max\left\{\frac{n}{2}-1, \underline{\frac{n}{2}}\right\}$$

$$+ max\left\{\underline{\frac{n}{2}}, \frac{n}{2}-1\right\} + max\left\{\underline{\frac{n}{2}+1}, \frac{n}{2}-2\right\} + \cdots + max\{\underline{n-1}, 0\}$$

$$= \underbrace{(n-1) + (n-2) + \cdots + \frac{n}{2}}_{\left(\frac{3n}{2}-1\right)\frac{n}{4}} + \underbrace{\frac{n}{2} + \left(\frac{n}{2}+1\right) + \cdots (n-1)}_{\left(\frac{3n}{2}-1\right)\frac{n}{4}} = \left(\frac{3n}{2}-1\right)\frac{n}{2}$$

$$\le \frac{3}{4} n^2$$

# Average-Case Analysis of *quick-select1*

- Proved average case time $T(n)$ is $O(n)$
- Average case is also $\Omega(n)$ since have to perform *partition*$(A, p)$
- Therefore average case is $T(n)$ is $\Theta(n)$

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# Randomized Algorithms

- A *randomized algorithm* is one which relies on some random numbers in addition to the input
- The cost will depend on both the input and the random numbers used
- **Goal**
  - shift the dependency of run-time from what we cannot control (the input), to what we can control (random numbers)
  - no more bad instances, just unlucky numbers
    - if running time is long on some instance, it's because we generated unlucky random numbers, not because of the instance itself
- Side note
  - computers cannot generate truly random numbers
  - we assume there is a pseudo-random number generator (PRNG), a deterministic program that uses an initial value or *seed* to generate a sequence of seemingly random numbers
  - quality of randomized algorithm depends on the quality of the PRNG

# Expected Running Time

- How do we measure the running time of a randomized algorithm?
  - it depends on the input $I$ and on $R$, the sequence of random numbers an algorithm choses during execution
- Define $T(I, R)$ to be running time of randomized algorithm for instance $I$ and $R$
- The *expected running time* $T^{exp}(I)$ for instance $I$ is expected value for $T(I, R)$

$$T^{exp}(I) = \boldsymbol{E}[T(I, R)] = \sum_{\substack{\text{all possible} \\ \text{sequences } R}} T(I, R) \cdot \Pr[R]$$

- *Worst-case expected running time*

$$T^{exp}(n) = \max_{\{I:size(I)=n\}} T^{exp}(I)$$

- *Average-case expected running time*

$$T^{exp}(n) = \frac{1}{|I: size(I) = n|} \sum_{I:Size(I)=n} T^{exp}(I)$$

- Usually design $A$ so that all instances of size $n$ have the same expected run time
- Thus the average and worst case expected run times are the same, and we just compute the worst case expected time

# Expected Running Time

- How do we measure the running time of a randomized algorithm?
  - it depends on the input $I$ and on $R$, the sequence of random numbers an algorithm choses during execution
- Define $T(I,R)$ to be running time of randomized algorithm for instance $I$ and $R$
- The *expected running time* $T^{exp}(I)$ for instance $I$ is expected value for $T(I,R)$

$$T^{exp}(I) = \boldsymbol{E}[T(I,R)] = \sum_{\substack{\text{all possible} \\ \text{sequences } R}} T(I,R) \cdot \Pr[R]$$

- *Worst-case expected running time* $T^{exp}(n) = \max_{\{I: size(I)=n\}} T^{exp}(I)$
- *Average-case expected running time* $T^{exp}(n) = \frac{1}{|I: size(I)=n|} \sum_{I: Size(I)=n} T^{exp}(I)$
- Usually design $A$ so that all instances of size $n$ have the same expected run time
- Thus average and worst case expected run times are usually the same
  - just compute the worst case expected time
- Sometimes we also want to know the running time if we got really unlucky with the random numbers $R$ we generate during the execution, or, formally

$$\max_{R} \max_{\{I: size(I)=n\}} T(I,R)$$

# Randomized QuickSelect: Shuffle

- **Goal**: create a randomized version of *QuickSelect* for which all input has the same expected run-time
- **First idea**: first randomly permute input using *shuffle* and then run selection algorithm

> *shuffle*$(A)$
> $A$ : array of size $n$
>     **for** $i \longleftarrow 0$ to $n-1$ **do**
>         *swap*$(A[i],\ A[random(i+1)])$

- $random(n)$ returns an integer uniformly sampled from $\{0, 1, 2, \ldots, n-1\}$
- can show that expected running time is $\Theta(n)$, the same as average running time

# Randomized QuickSelect: Shuffle

- **Goal**: create a randomized version of *QuickSelect* for which all input has the same expected run-time
- **First idea**: first randomly permute input using *shuffle* and then run selection algorithm

*shuffle*$(A)$
$A$ : array of size $n$
  **for** $i \longleftarrow 0$ to $n-1$ **do**
    *swap*$(A[i],\ A[random(i+1)])$

- $random(n)$ returns an integer uniformly sampled from $\{0, 1, 2, \dots, n-1\}$
- can show that expected running time is $\Theta(n)$, the same as average running time
- if we get very unlucky with random numbers, we could get a sorted or almost sorted array after shuffle, resulting in $O(n^2)$ performance for selection algorithm
  - probability of this happening is almost zero
- whereas the user is quite likely to give instance which is sorted or almost sorted to the selection algorithm
  - probability is far from zero, humans often produce almost sorted data

# Randomized QuickSelect: Random Pivot

- **Second idea**: select a random pivot from $\{0, 1, 2, \ldots, n-1\}$

*choose-pivot2*$(A)$
    **return** $random(A.size())$

- Simpler and more efficient than shuffling the array
- Usually fastest in practice
- Expected running time is again $\Theta(n)$

# Efficiency of Randomized QuickSelect

*quick-select2*$(A, k)$
$p \longleftarrow$ *choose-pivot2*$(A)$
"the rest"

*choose-pivot2*$(A)$
**return** $random(A.size())$

- Assume all elements of $A$ are distinct
- Select pivot with equal probability at each recursive call, and independently from other recursive calls
  - $P(\text{pivot has index } i) = \frac{1}{n}$ for any instance of size $n$
- $T^{exp}(I)$ depends only on the size of $I$, not the contents of $I$
- Let $T^{exp}(n)$ be expected time on an instance of size $n$
- Running time to partition array is $cn$, and with probability $1/n$ pivot-index is $i$

$$\underbrace{\boxed{\quad T^{exp}(i) \quad}}_{\text{size } i}\overset{\boldsymbol{i}}{\boxed{\, v \,}}\underbrace{\boxed{\quad T^{exp}(n-i-1) \quad}}_{\text{size } n-i-1}$$

running time if pivot index is $i \leq c \cdot n + max\{T^{exp}(i), T^{exp}(n-i-1)\}$

# Efficiency of Randomized QuickSelect

running time if pivot-index is $i \leq c \cdot n + max\{T^{exp}(i), T^{exp}(n-i-1)\}$

- Taking expectation over pivot index $i$

$$T^{exp}(n) = \sum_{i=0}^{n-1} (running\ time\ if\ pivot\ index\ is\ i) P(\text{index of pivot is } i)$$

$$\leq \sum_{i=0}^{n-1} (cn + max\{T^{exp}(i), Texp(n-i-1\}) \frac{1}{n}$$

$$\leq cn + \sum_{i=0}^{n-1} \frac{1}{n} max\{T^{exp}(i), T^{exp}(n-i-1)\}$$

- Same recurrence as for non-randomized average case
- Resolves to $\Theta(n)$ expected time on instance of size $n$
- Side note
  - there is selection algorithm “Median of Medians” (cs341) that has worst-case running time $O(n)$
    - uses double recursion
    - slower in practice

# QuickSelect: Badly Designed Randomization

*choose-random-pivot-badly*$(A)$

  **if** $A.size \geq 3$ **return** $random(3)$

  **else return** $0$

$$T^{exp}(n) = \max_{\{I : size(I) = n\}} T^{exp}(I)$$

- Worst instance is sorted array $I_n = \{0, 1, \dots, n-1\}$
- $T^{exp}(I_n) = \begin{cases} cn + \frac{1}{3}T^{exp}(I_{n-1}) + \frac{1}{3}T^{exp}(I_{n-2}) + \frac{1}{3}T^{exp}(I_{n-3}) & \text{if } n \geq 3 \\ c & \text{if } n < 3 \end{cases}$
- $T^{exp}(I_n) \geq cn + T(I_{n-3})$ if $n \geq 3$
- Resolves to $\Theta(n^2)$
- Worst case expected time is $\Theta(n^2)$

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# QuickSort

- Hoare developed *partition* and *quick-select* in 1960
- He also used them to *sort* based on partitioning

*quick-sort1*$(A)$

Input: array *A* of size *n*

**if** $n \leq 1$ **then return**

$p \leftarrow$ *choose-pivot1*$(A)$

$i \leftarrow$ *partition* $(A, p)$

*quick-sort1*$(A[0, 1, \dots, i-1])$

*quick-sort1*$(A[i+1, \dots, n-1])$

- Let $T(n)$ to be the runtime on size $n$ array
- If we know pivot-index $i$, then $T(n) = cn + T(i) + T(n-i-1)$
- Worst case $T(n) = T(n-1) + cn$
  - recurrence solved in the same way as quick–select1, $\Theta(n^2)$
- Best case $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn$
  - solved in the same way as merge-sort, $\Theta(n \log n)$

# Average-case analysis of quick-sort1

- Make the same assumptions as for quick-select1
- Deriving recurrence equation is similar to quick-select1, but recurse on both sides

i=2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | $v = 2$ | 3 | 5 | 8 | 9 | 6 | 4 | 7 |

recurse    recurse

- Using the same approach as for quick-select1, average running time is

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(cn + T(i) + T(n-i-1)\big), \qquad n \geq 2$$

- Running time is proportional to the number of comparisons
- Recurrence for counting comparisons

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(n + T(i) + T(n-i-1)\big), \qquad n \geq 2$$

# Average-case analysis of quick-sort1

- First let us get a simpler recursive expression for $T(n)$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\left(n + T(i) + T(n-i-1)\right)$$

$$= n + \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{i=0}^{n-1} T(n-i-1)$$

$$T(0) + T(1) + \cdots + T(n-1) \qquad T(n-1) + T(n-2) + \cdots + T(0)$$

$$= n + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

- Thus $\quad T(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$

# Average-case analysis of quick-sort1

$$T(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \text{ is } \Theta(n \log n)$$

**Proof**

Multiply by $n$: $\quad nT(n) = n^2 + 2\sum_{i=0}^{n-1} T(i)$

Plug in $n-1$: $\quad (n-1)T(n-1) = (n-1)^2 + 2\sum_{i=0}^{n-2} T(i)$

Subtract: $\quad nT(n) - (n-1)T(n-1) = 2n - 1 + 2T(n-1)$

Rearrange : $\quad nT(n) = (n+1)T(n-1) + 2n - 1$

Divide by $(n+1)n$: $\quad \dfrac{T(n)}{n+1} = \dfrac{T(n-1)}{n} + \dfrac{2n-1}{n(n+1)}$

Let $A(n) = \frac{T(n)}{n+1}$ : $\quad A(n) = A(n-1) + \dfrac{2n-1}{n(n+1)} = A(n-2) + \dfrac{2(n-1)-1}{(n-1)n} + \dfrac{2n-1}{n(n+1)}$

$$= \cdots = \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} = \underbrace{\sum_{i=1}^{n} \frac{2}{i+1}}_{\Theta(\log n)} - \underbrace{\sum_{i=1}^{n} \frac{1}{i(i+1)}}_{\Theta(1)}$$

Therefore: $\quad A(n) = c \log n$

Finally: $\quad T(n) = (n+1)A(n) = c(n+1)\log n \in \Theta(n \log n)$

# Improvement ideas for QuickSort

- Randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*
- The auxiliary space is $\Omega$(recursion depth)
  - $\Theta(n)$ in the worst-case
  - can be reduce to $\Theta(\log n)$ worst-case by
    - recurse in smaller sub-array first
    - replacing the other recursion by a while-loop (tail call elimination)
- Stop recursion when, say $n \leq 10$
  - array is not completely sorted, but almost sorted
  - at the end, run insertionSort, it sorts in just $O(n)$ time since all items are within 10 units of the required position
- Arrays with many duplicates sorted faster by changing *partition* to produce three subsets

| | | |
|---|---|---|
| $< v$ | $= v$ | $> v$ |

- Programming tricks
  - instead of passing full arrays, pass only the range of indices
  - avoid recursion altogether by keeping an explicit stack

# QuickSort with Tricks

*quick-sort3*$(A, n)$

initialize a stack $S$ of index-pairs with $\{(0, n-1)\}$

**while** $S$ is not empty

$(l, r) \leftarrow S.pop()$ // get the next subproblem

**while** $r - l + 1 > 10$ // work on it if it's larger than 10

$p \leftarrow$ *choose-pivot2*$(A,\ l,\ r)$

$i \leftarrow$ *partition* $(A, l, r, p)$

**if** $i - l > r - i$ **do** // is left side larger than right?

$S.push\big((l, i-1)\big)$ // store larger problem in $S$ for later

$l \leftarrow i + 1$ // next work on the right side

**else**

$S.push\big((i+1, r)\big)$ // store larger problem in $S$ for later

$r \leftarrow i - 1$ // next work on the left side

*InsertionSort*$(A)$

- This is often the most efficient sorting algorithm in practice

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# Lower bounds for sorting

- We have seen many sorting algorithms

| Sort | Running Time | Analysis |
|---|---|---|
| Selection Sort | $\Theta(n^2)$ | worst-case |
| Insertion Sort | $\Theta(n^2)$ | worst-case |
| Merge Sort | $\Theta(n \log n)$ | worst-case |
| Heap Sort | $\Theta(n \log n)$ | worst-case |
| quick-sort1<br>quick-sort2 | $\Theta(n \log n)$<br>$\Theta(n \log n)$ | average-case<br>expected |

- **Question**: Can one do better than $\Theta(n \log n)$ running time?
- **Answer**: *It depends on what we allow*
  - No: comparison-based sorting lower bound is $\Omega(n \log n)$
    - no restriction on input, just must be able to compare
  - Yes: non-comparison-based sorting can achieve $O(n)$
    - restrictions on input

# The Comparison Model

- All sorting algorithms seen so far are in the comparison model
- In the *comparison model* data can only be accessed in two ways
  - comparing two elements
    - $A[i] \leq A[j]$
  - moving elements around (e.g. copying, swapping)
- This makes very few assumptions on the things we are sorting
  - just count the number of above operations
- Under comparison model, will show that any sorting algorithm requires $\Omega(n \log n)$ comparisons
- This lower bound is not for an algorithm, it is for the sorting problem
- How can we talk about problem without algorithm?
  - count number of comparisons any sorting algorithm has to perform

# Decision Tree

- Decision tree succinctly describes all the decisions that are taken during the execution of an algorithm and the resulting outcome
- For each sorting algorithm we can construct a corresponding decision tree
- Given decision tree, we can deduce the algorithm
- Decision tree can be constructed for any algorithm, not just sorting

# Decision Tree Example

- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements $[x_0, x_1, x_2]$

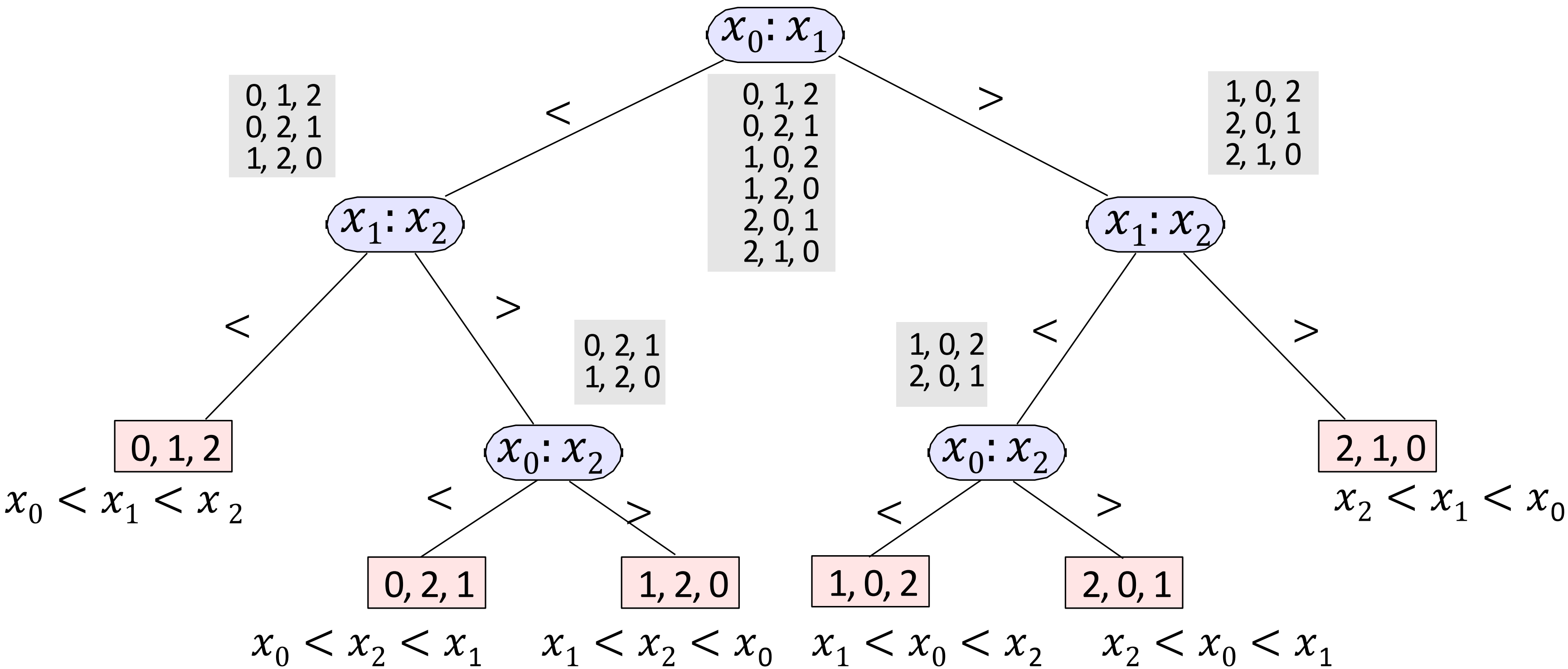
Set of all possible inputs

| | | |
|---|---|---|
| 0, 1, 2 | $x_0 < x_1 < x_2$ | output $[x_0, x_1, x_2]$ |
| 0, 2, 1 | $x_0 < x_2 < x_1$ | output $[x_0, x_2, x_1]$ |
| 1, 0, 2 | $x_1 < x_0 < x_2$ | output $[x_1, x_0, x_2]$ |
| 1, 2, 0 | $x_2 < x_0 < x_1$ | output $[x_2, x_0, x_1]$ |
| 2, 0, 1 | $x_1 < x_2 < x_0$ | output $[x_1, x_2, x_0]$ |
| 2, 1, 0 | $x_2 < x_1 < x_0$ | output $[x_2, x_1, x_0]$ |

- Have to determine which of the 6 inputs we are given before can give output
  - unique output for each distinct input

# Decision Tree

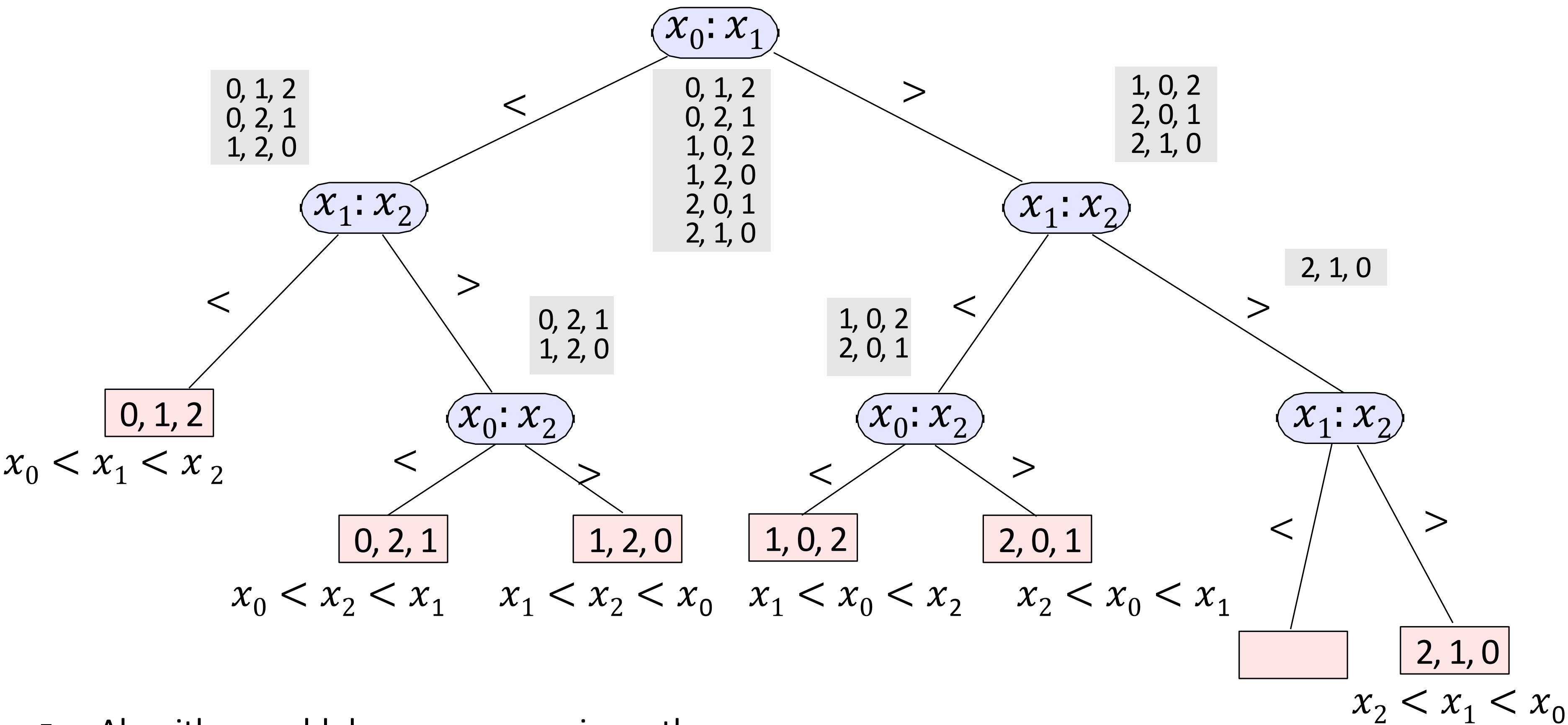
- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements




- Root corresponds to the set of all possible inputs
- Interior nodes are comparisons: each comparison splits the set of possible inputs into two
- Know correct sorting order only when the set of possible inputs shrinks to size one
    - nodes where possible input shrunk to size one are leaves, when reach them, can output sorting result
- Sorting algorithm will traverse a path starting at root and ending at a leaf
    - length of the path is the number of comparisons to be made
- Tree height is the number of comparisons required for sorting in the worst case

# Decision Tree

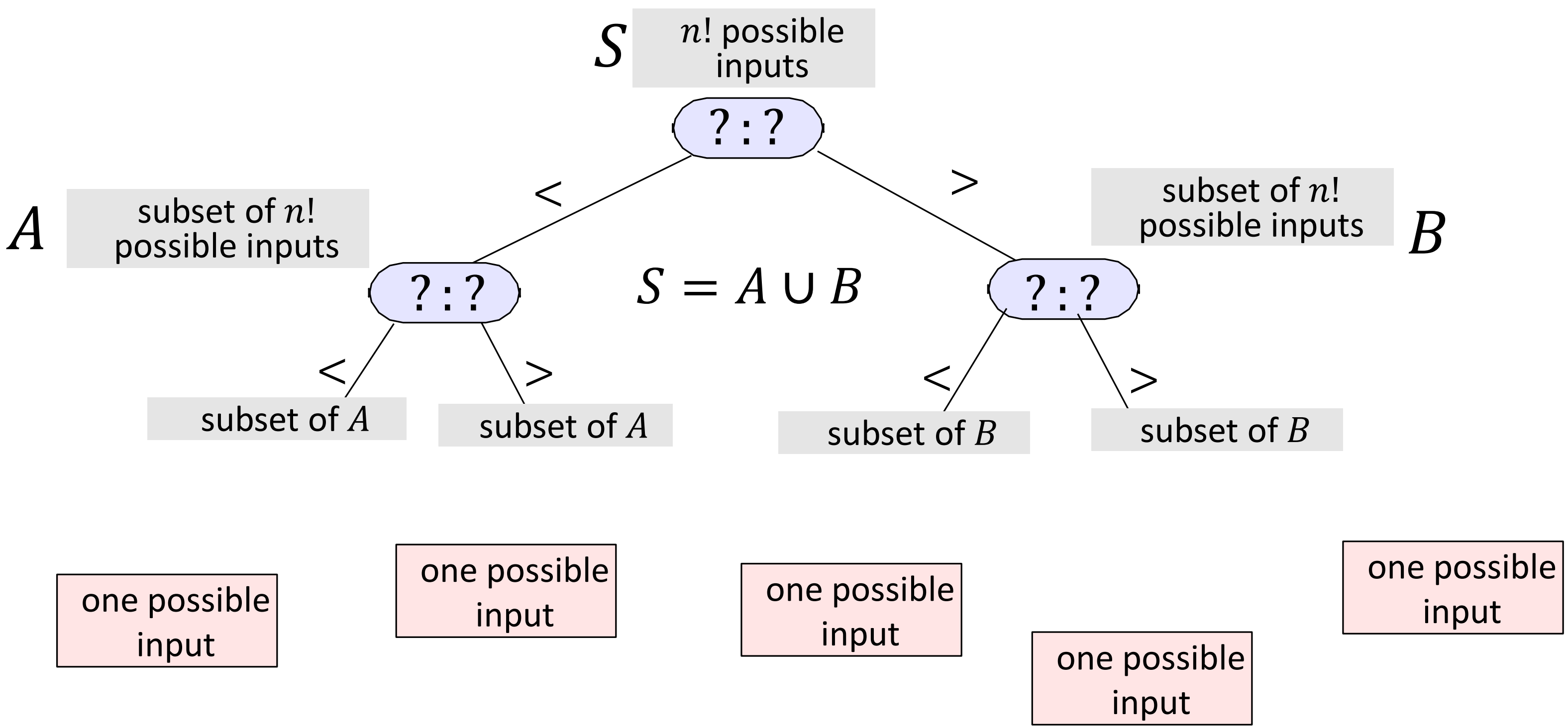
- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements




- Algorithm could do more comparisons than necessary
- Thus can have more leafs than possible inputs
- But the number of leaves must be *at least* the number of possible inputs

# Decision Tree

- *Decision tree* for any comparison-based sorting algorithm, $n$ non-repeating elements




- Tree must have at least $n!$ leaves
- Binary tree with height $h$ has at most $2^h$ leaves
- Height $h$ must be at least such that $2^h \geq n!$
- Tree height is the number of comparisons required in the worst case

# Lower bound for sorting in the comparison model

**Theorem:** Any correct comparison-based sorting algorithm requires at least $\Omega(n\log n)$ comparisons

**Proof:**

- There exists a set of $n!$ possible inputs s.t. each leads to a different output
- Decision tree must have at least $n!$ leaves
- Binary tree with height $h$ has at most $2^h$ leaves
- Height $h$ must be at least such that $2^h \geq n!$
- Taking logs of both sides

$$h \geq \log(n!) = \log(n(n-1)\ldots\cdot 1) = \overbrace{\log n + \cdots + \log(\tfrac{n}{2}+1)}^{\geq \log\frac{n}{2}} + \cancel{\log\tfrac{n}{2} + \cdots + \log 1}$$

$$\geq \underbrace{\log\frac{n}{2} + \cdots + \log\frac{n}{2}}_{\frac{n}{2}\text{ of them}} = \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}\log n - \frac{n}{2} \in \Omega(n\log n)$$

# Outline

- Sorting and Randomized Algorithms
  - QuickSelect
  - Randomized Algorithms
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

# Non-Comparison-Based Sorting

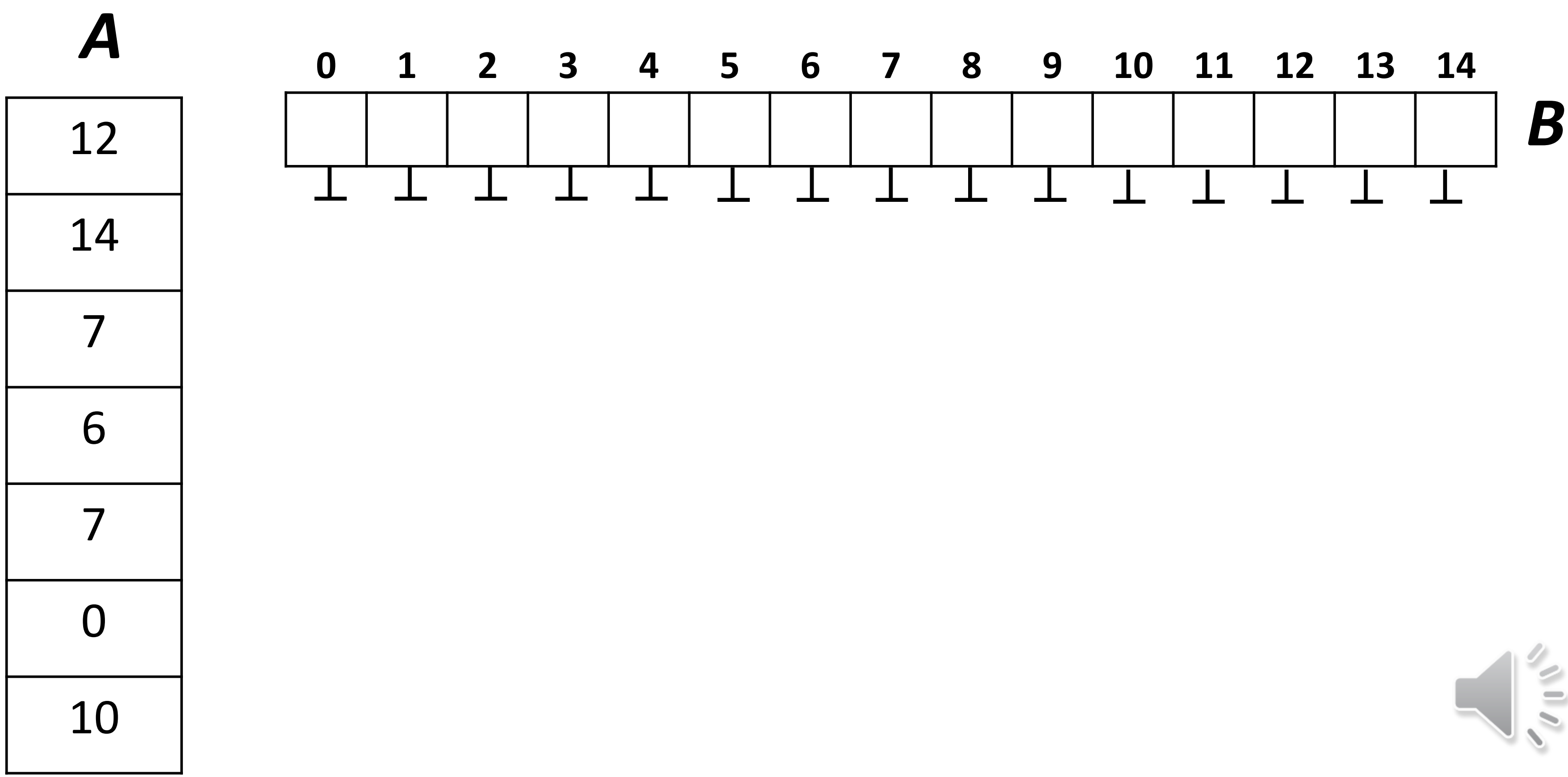
- Sort without comparing items to each other
- Non-comparison based sorting is less general than comparison based
- In particular, we need to make assumptions about items we sort
  - unlike in comparison based sorting, which sorts any data, as long as it can be compared
- Will assume we are sorting non-negative integers
  - can adapt to negative integers
  - also to some other data types, such as strings
  - but cannot sort arbitrary data

# Non-Comparison-Based Sorting

- Simplest example
  - suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- For non-comparison sorting, running time depends on both
  - array size $n$
  - $L$

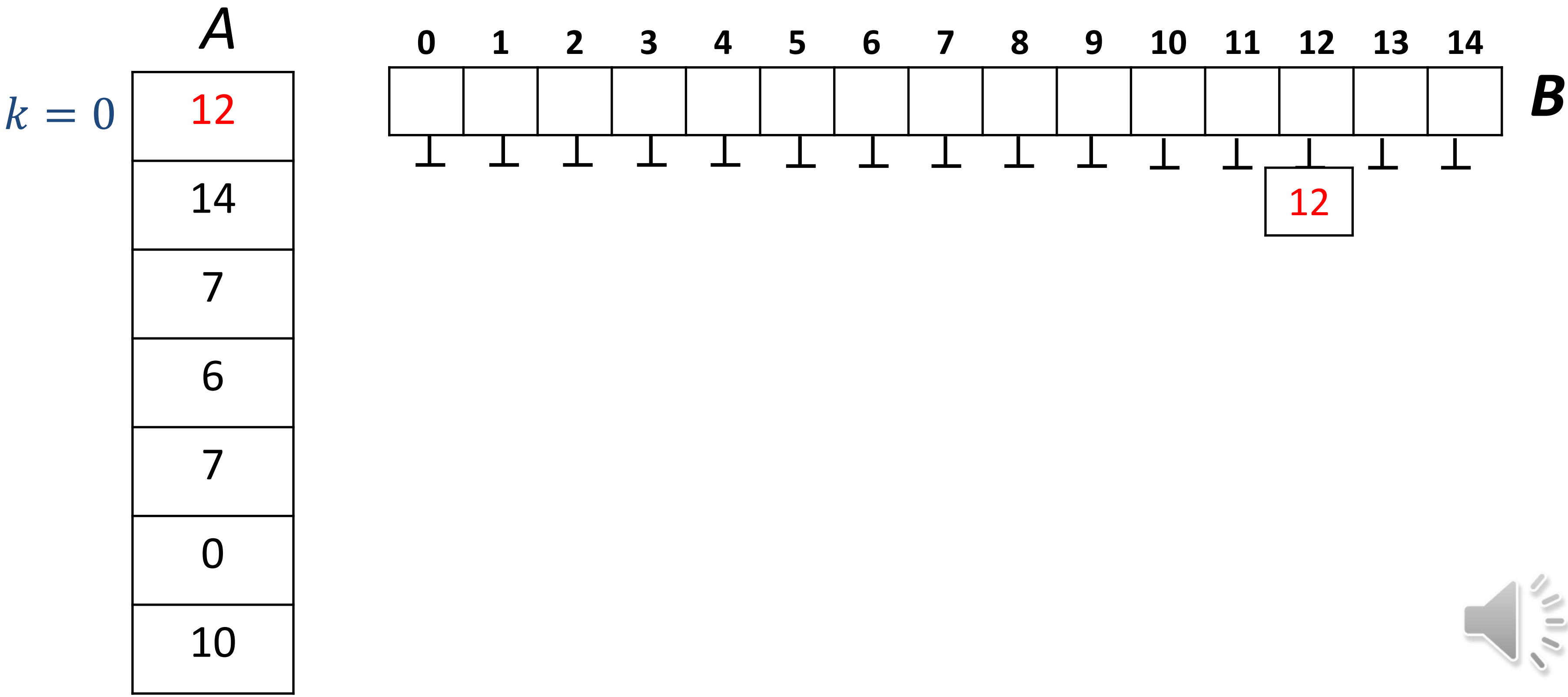
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
  - i.e. array of initially empty linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



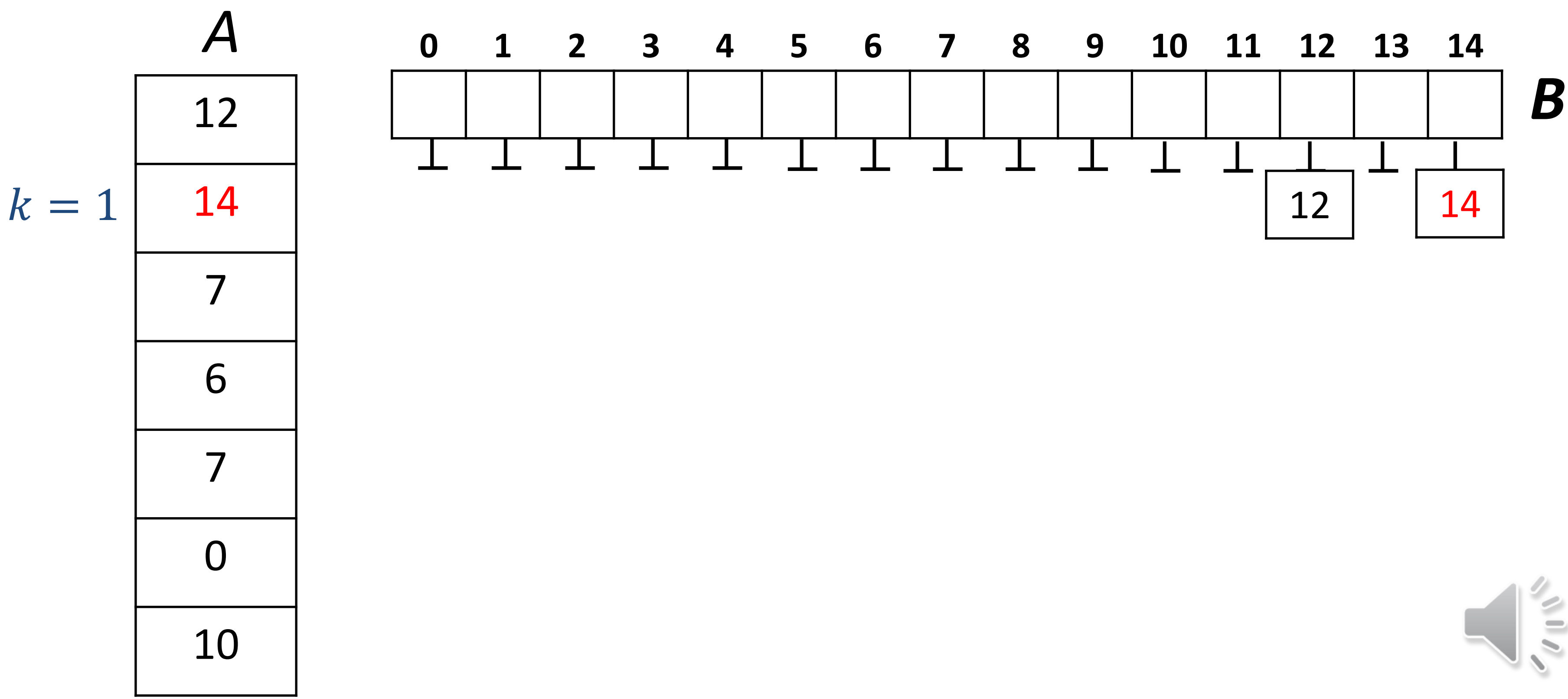
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \dots, L-1]$
- Use an axillary *bucket array* $B[0, \dots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



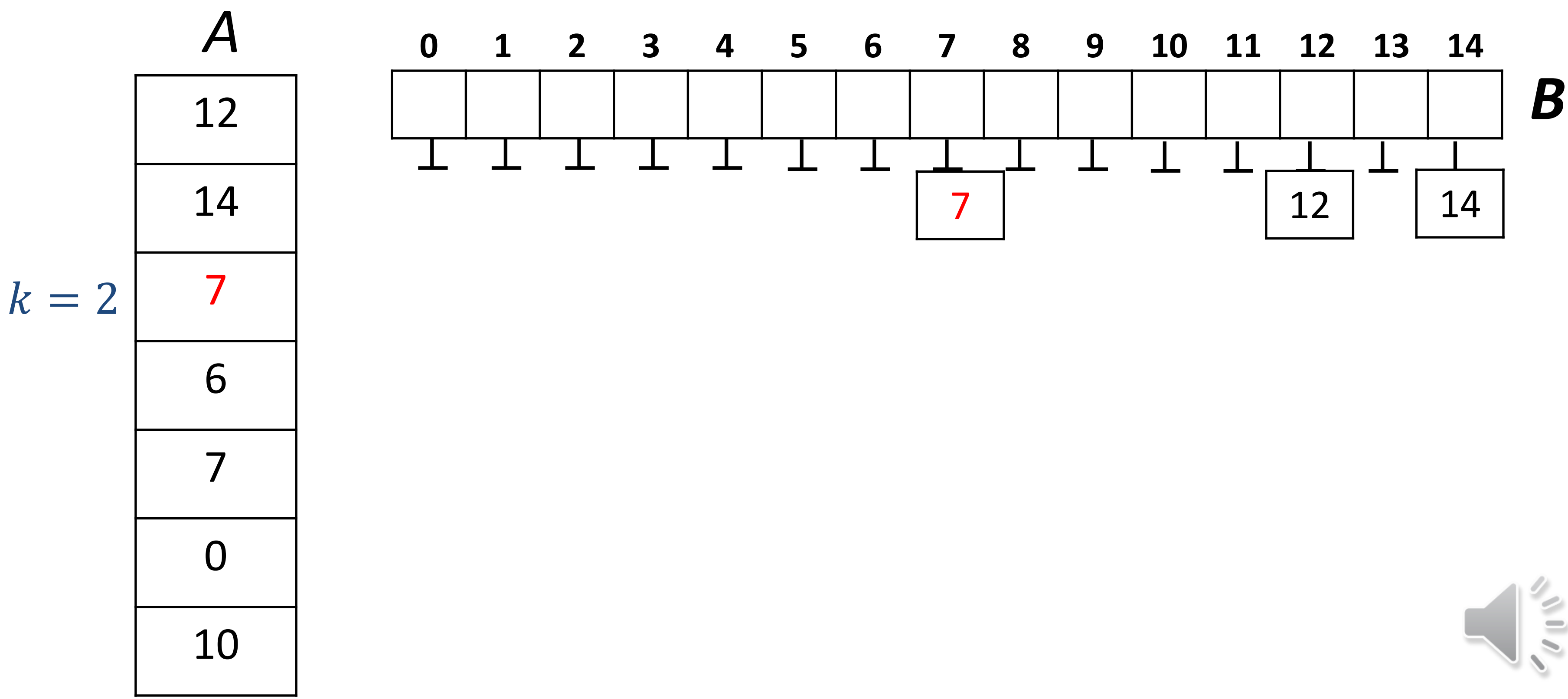
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
    - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



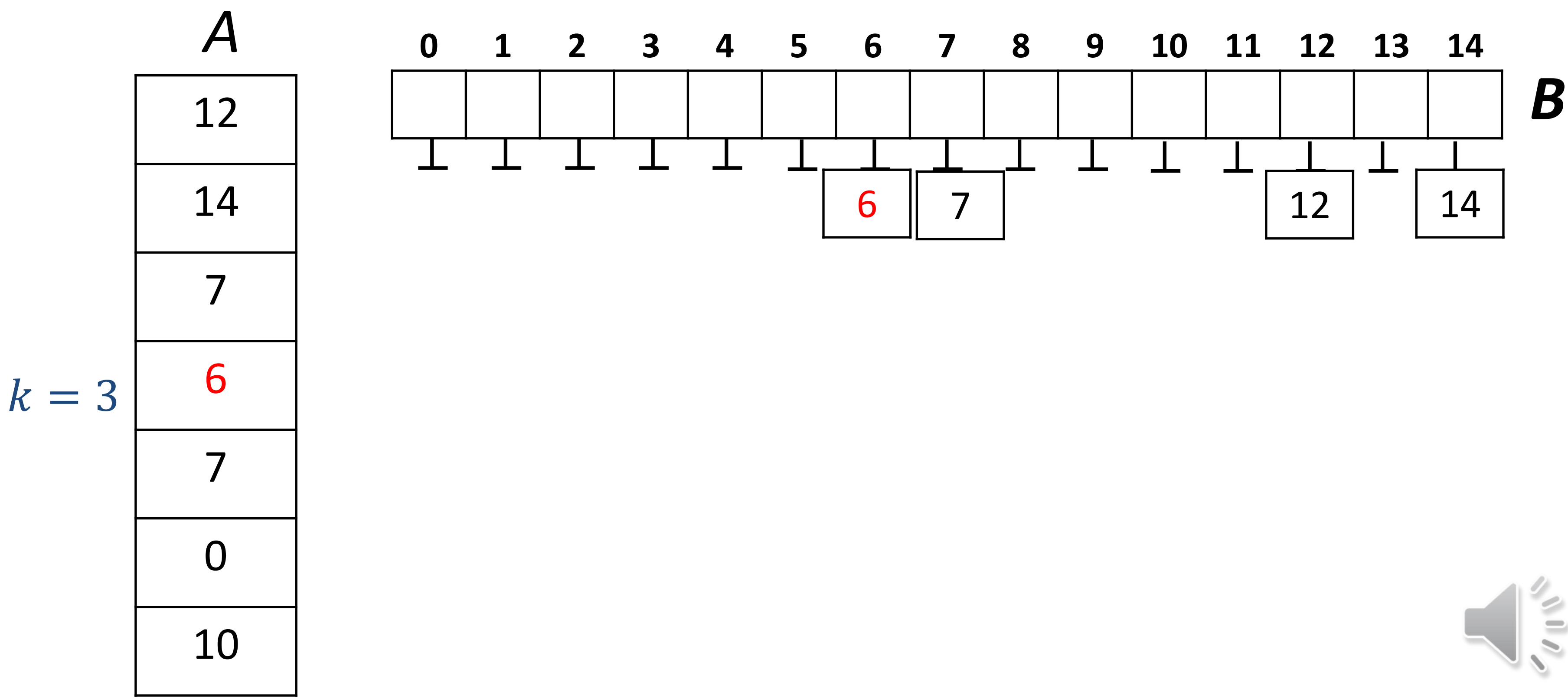
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \dots, L-1]$
- Use an axillary *bucket array* $B[0, \dots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



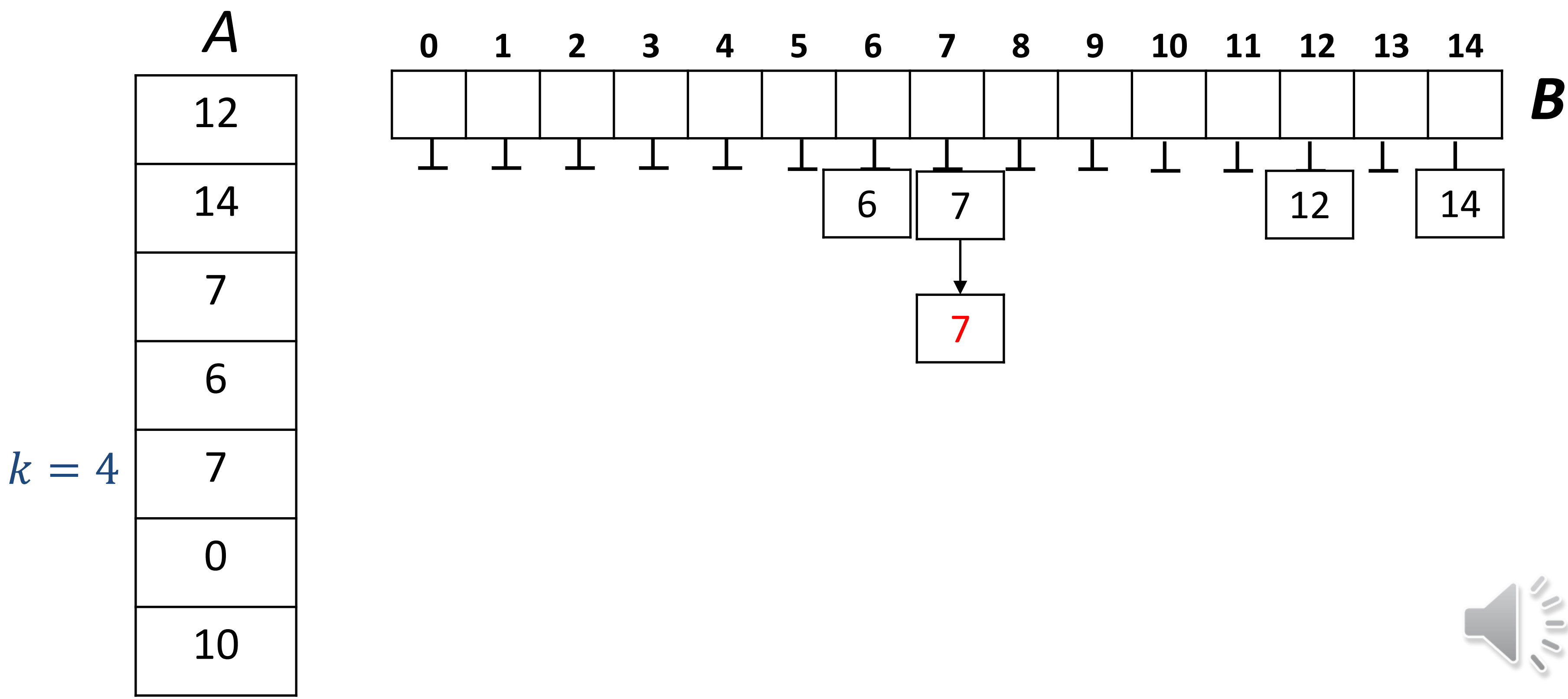
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



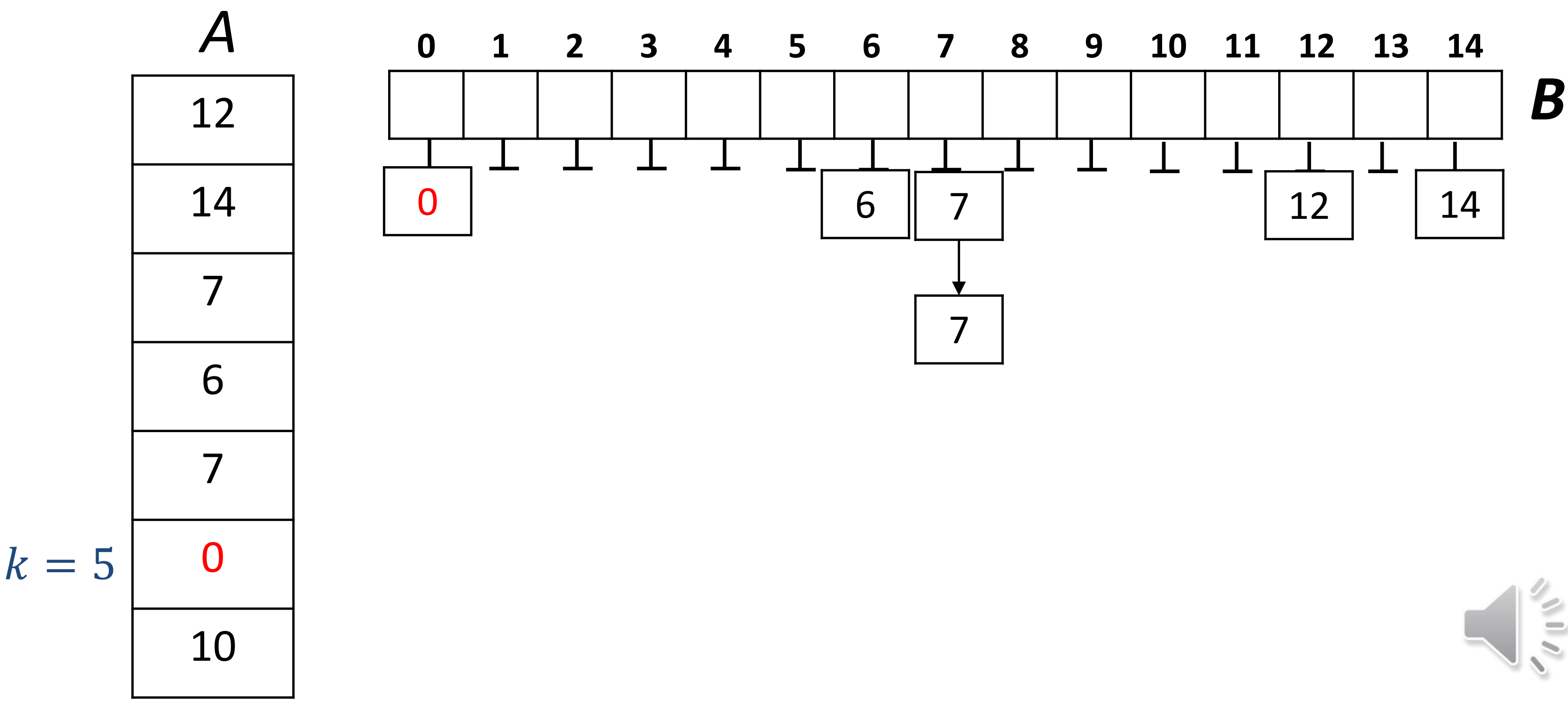
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



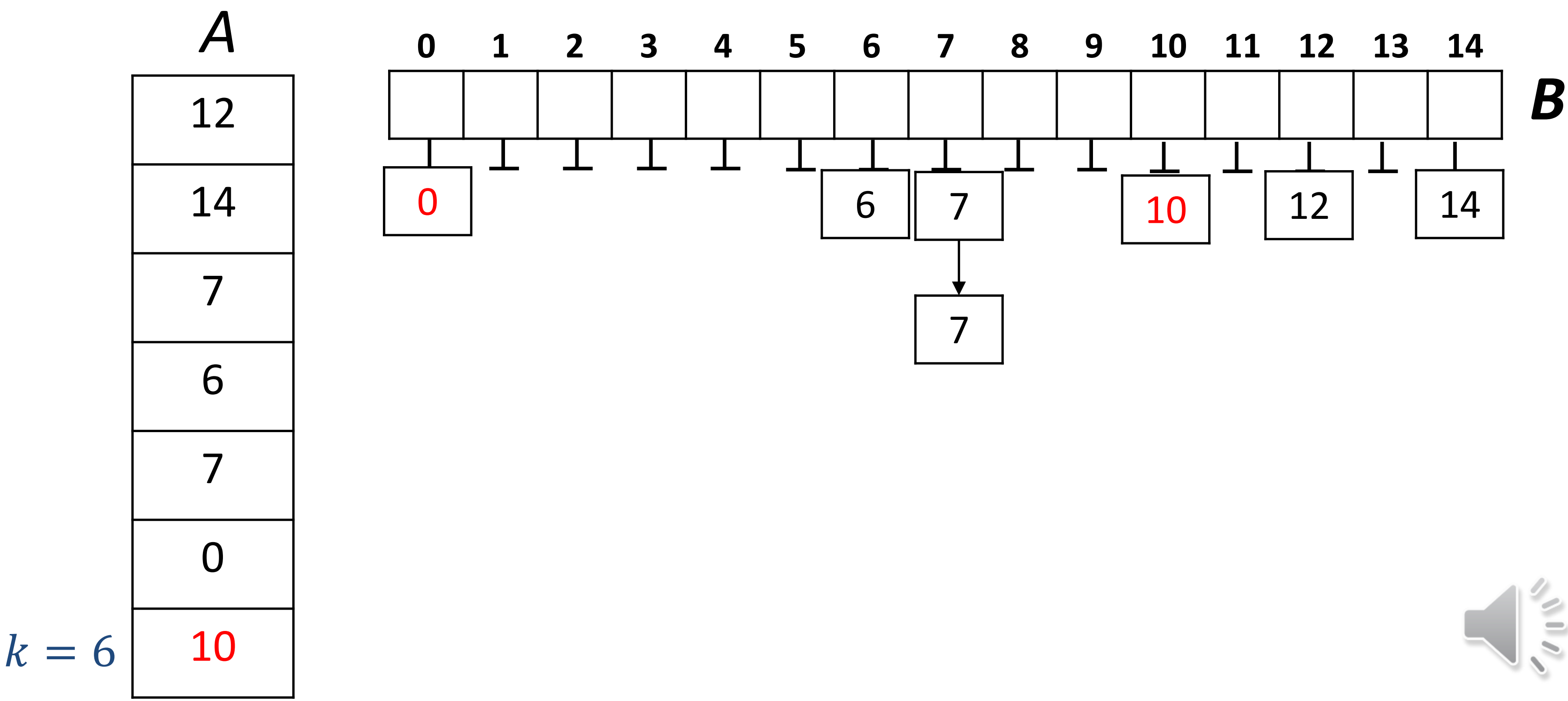
# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \dots, L-1]$
- Use an axillary *bucket array* $B[0, \dots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



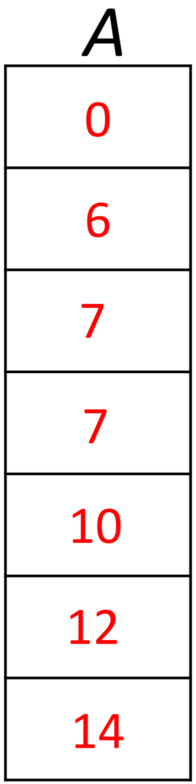
# Bucket Sort

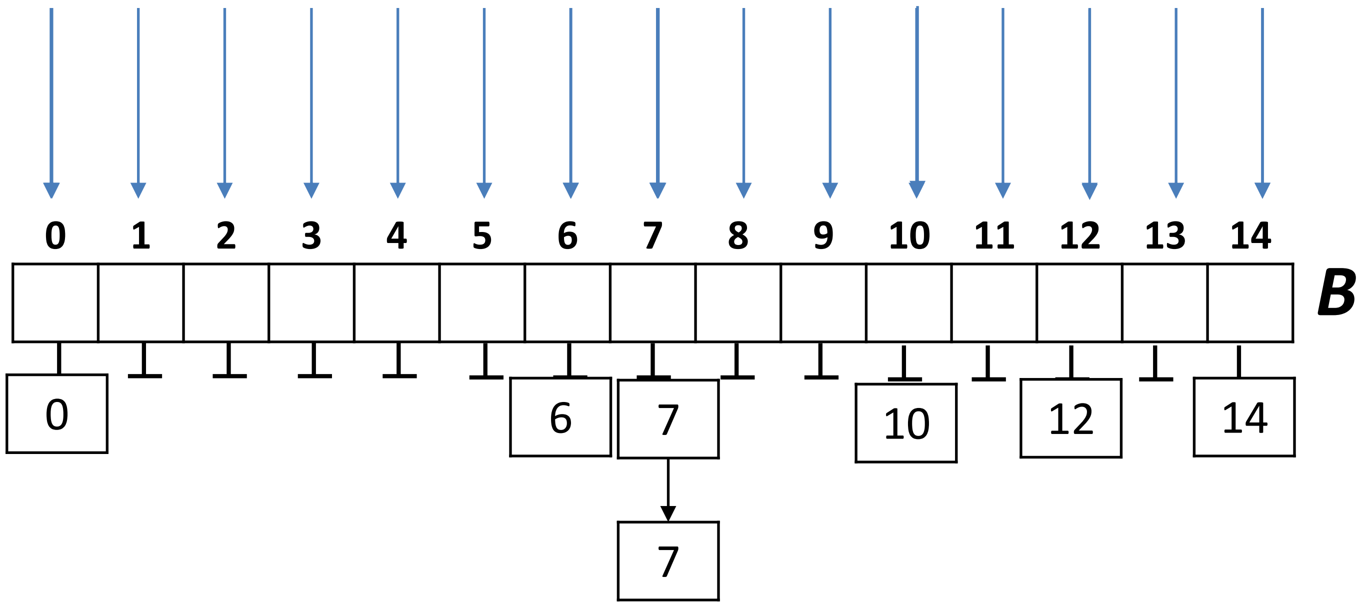
- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
    - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



# Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$
- Now iterate through $B$ and copy non-empty buckets to $A$




- Time complexity is $\Theta(L+n)$
  - $n$ is size of $A$

# Digit Based Non-Comparison-Based Sorting

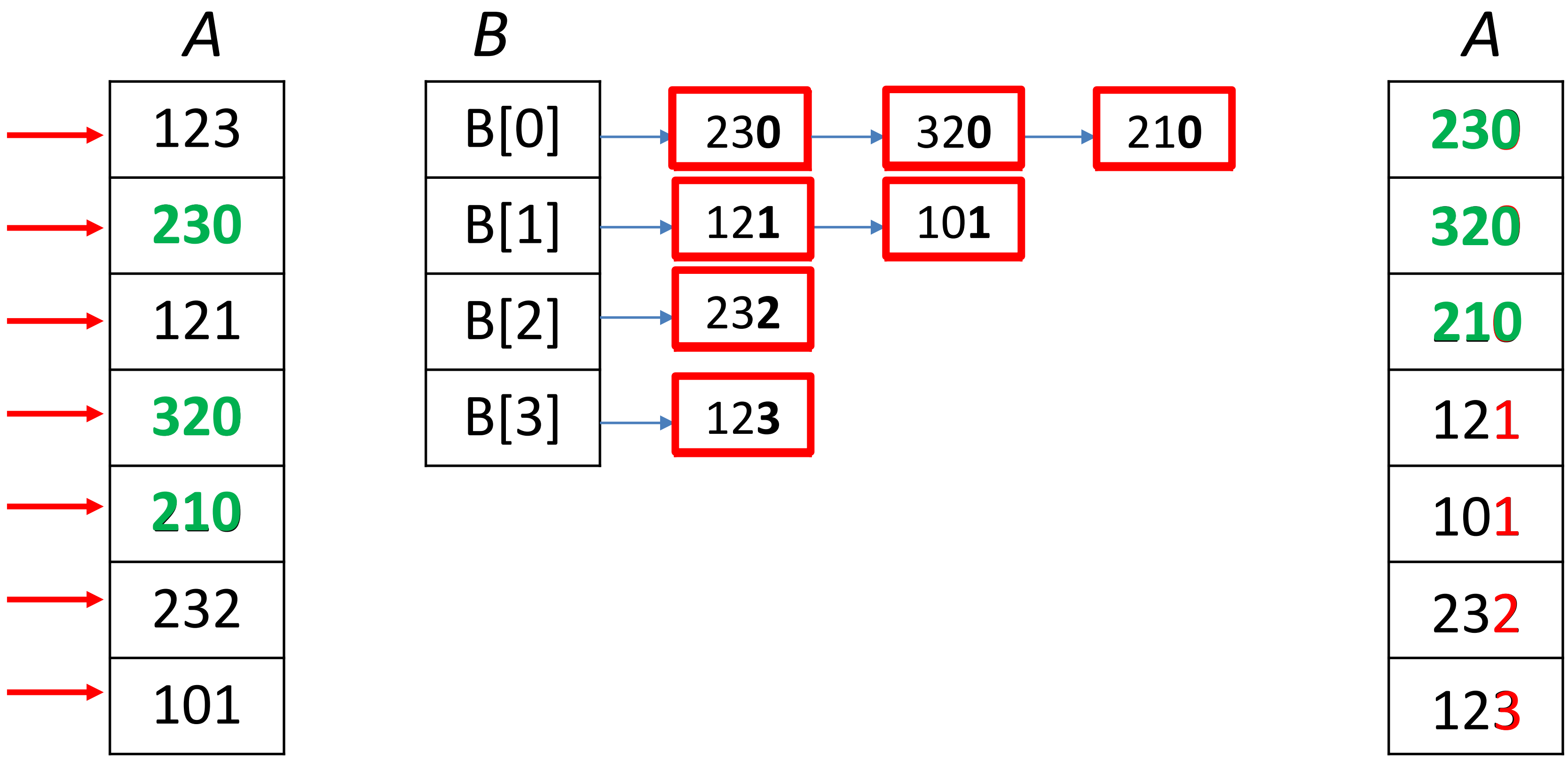
- Running time of bucket sort is $\Theta(L+n)$
  - $n$ is size of $A$
  - $L$ is range $[0, L)$ of integers in $A$
- What if $L$ is much larger than $n$?
  - i.e. $A$ has size $100$, range of integers in $A$ is $[0, \dots, 99999]$
- Assume at most $m$ digits in any key
  - pad with leading 0s

| | | | | | | |
|---|---|---|---|---|---|---|
| 123 | 230 | 021 | 320 | 210 | 232 | 101 |

- Can sort 'digit by digit', can go
  - forward, from digit $1 \rightarrow m$ (more obvious)
  - backward, from from digit $m \rightarrow 1$ (less obvious)
  - bucketsort is perfect for sorting 'by digit'
- Example: $A$ has size 100, range of integers in $A$ is [0,…,99999]
  - integers have at most $5$ digits, need only $5$ iterations of bucketsort

# Bucket Sort on Last Digit

- Equivalent to normal bucket sort if we redefine comparison
  - $a \leq b$ if the last digit of $a$ is smaller than (or equal) to the last digit of $b$




- Bucket sort is stable: equal items stay in original order
  - crucial for developing LSD radix sort later

# Base $R$ number representation

- Number of distinct digits gives the number of buckets $R$
- Useful to control number of buckets
  - larger $R$ means less digits (less iterations), but more work per iteration (larger bucket array)
  - may want exactly $2$, or $4$, or even $128$ buckets
- Can do so with base $R$ representation
  - digits go from $0$ to $R - 1$
  - $R$ buckets
  - numbers are in the range $\{0, 1, \dots, R^m - 1\}$
- From now on, assume keys are numbers in base $R$ ($R$: radix)
  - $R = 2, 10, 128, 256$ are common
- Example ($R = 4$)

| | | | | | | |
|---|---|---|---|---|---|---|
| 123 | 230 | 21 | 320 | 210 | 232 | 101 |

# Single Digit Bucket Sort

*Bucket-sort*$(A, d)$

$A$ : array of size $n$, contains numbers with digits in $\{0, \ldots, R-1\}$

$d$: index of digit by which we wish to sort

```
initialize array B[0,...,R−1] of empty lists (buckets)
for i ⟵ 0 to n−1 do
    next ⟵ A[i]
    append next at end of B[dth digit of next]
i ⟵ 0
for j ⟵ 0 to R−1 do
    while B[j] is non-empty do
        move first element of B[j] to A[i++]
```

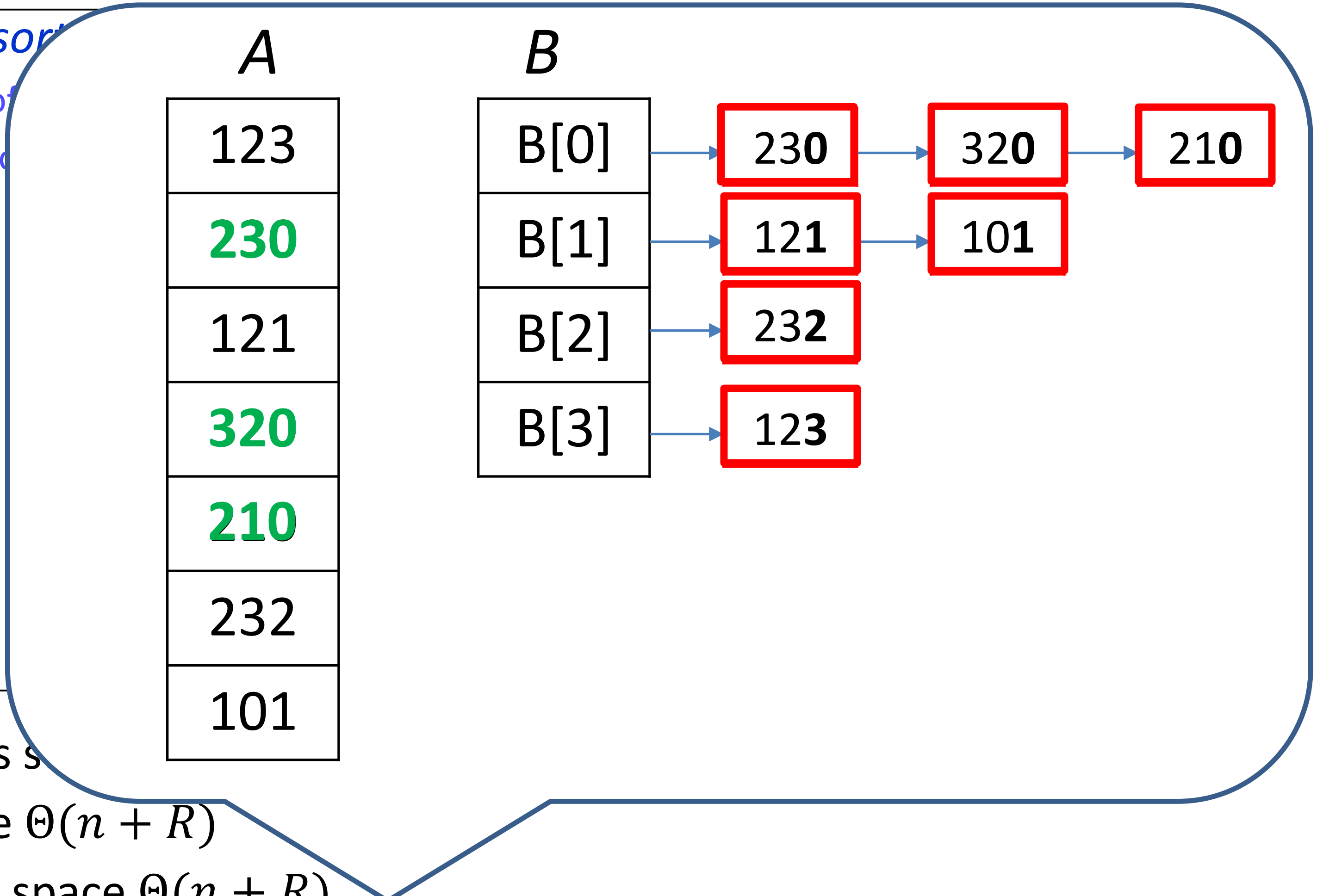
- Sorting is stable: equal items stay in original order
- Run-time $\Theta(n + R)$
- Auxiliary space $\Theta(n + R)$
  - $\Theta(R)$ for array $B$, and linked lists are $\Theta(n)$

# Single Digit Bucket Sort

*Bucket-sor*

$A$ : array of

$d$: index c




- Sorting is s
- Run-time $\Theta(n + R)$
- Auxiliary space $\Theta(n + R)$
    - $\Theta(R)$ for array $B$, and linked lists are $\Theta(n)$
- Can replace lists by two auxiliary arrays of size $R$ and $n$, resulting in *count-sort*
    - no details

# MSD-Radix-Sort

- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit

| |
|---|
| 123 |
| 232 |
| 021 |
| 320 |
| 210 |
| 230 |
| 101 |

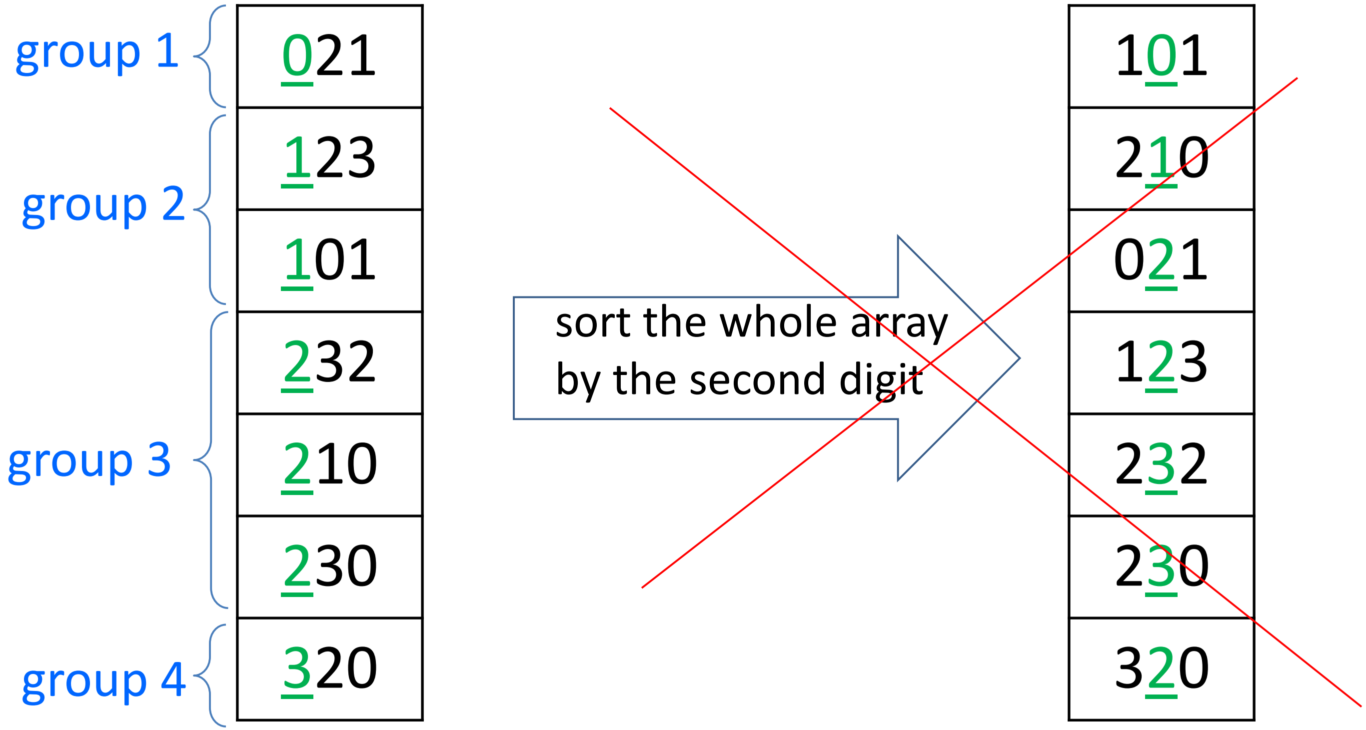
# MSD-Radix-Sort

- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit

| |
|---|
| 123 |
| 232 |
| 021 |
| 320 |
| 210 |
| 230 |
| 101 |

# MSD-Radix-Sort

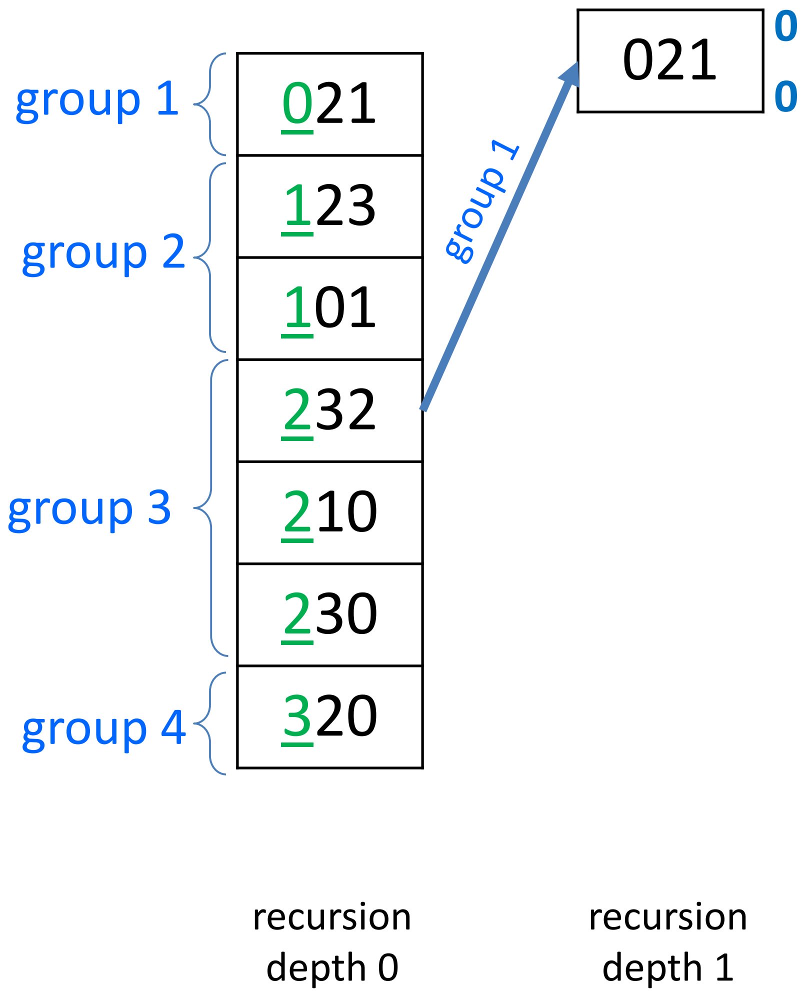
- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit




- Cannot sort the whole array by the second digit, will mess up the order
- Have to break down in groups by the first digit
    - each group can be safely sorted by the second digit
    - call sort recursively on each group, with appropriate array bounds

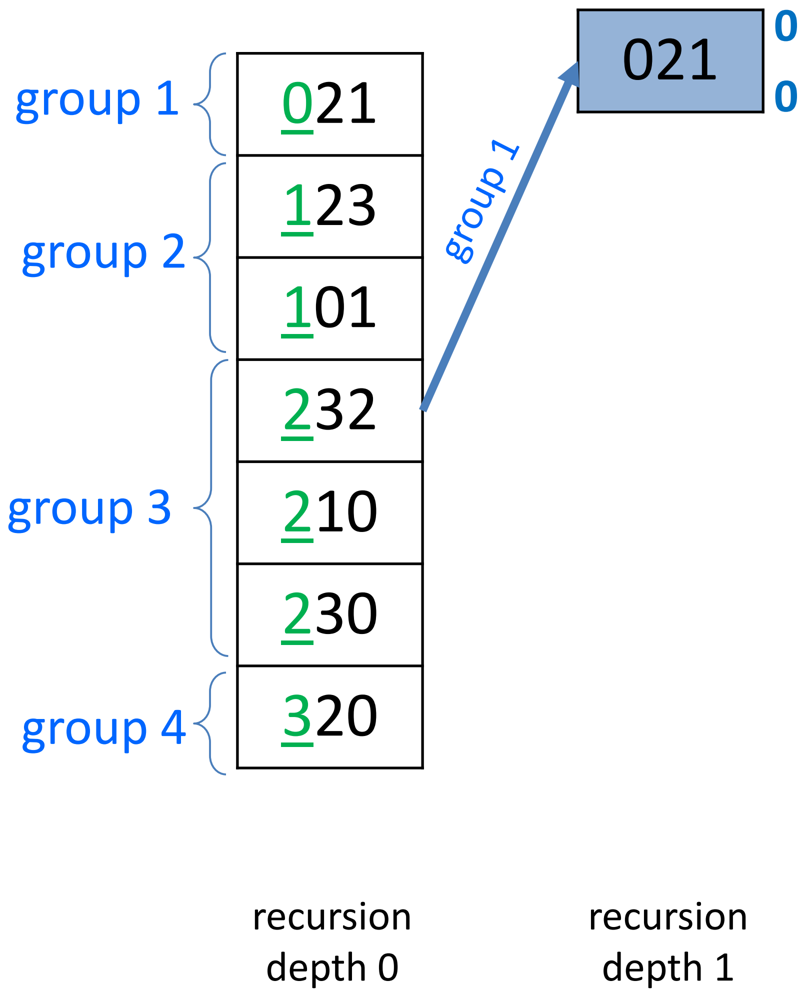
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



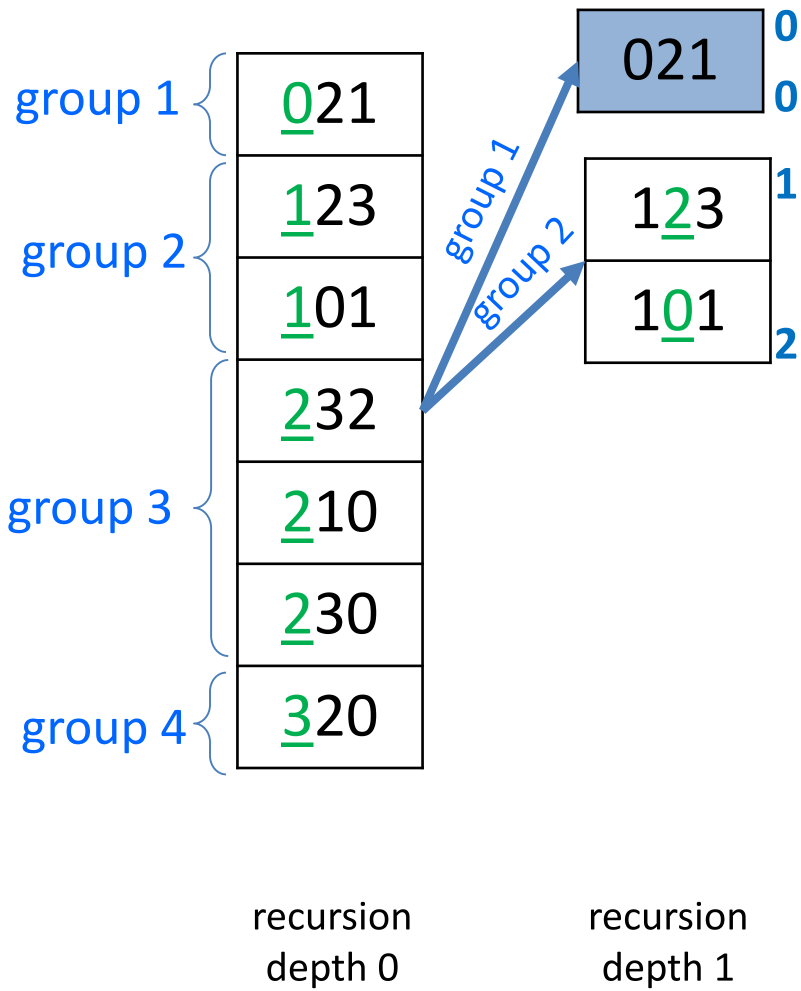
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



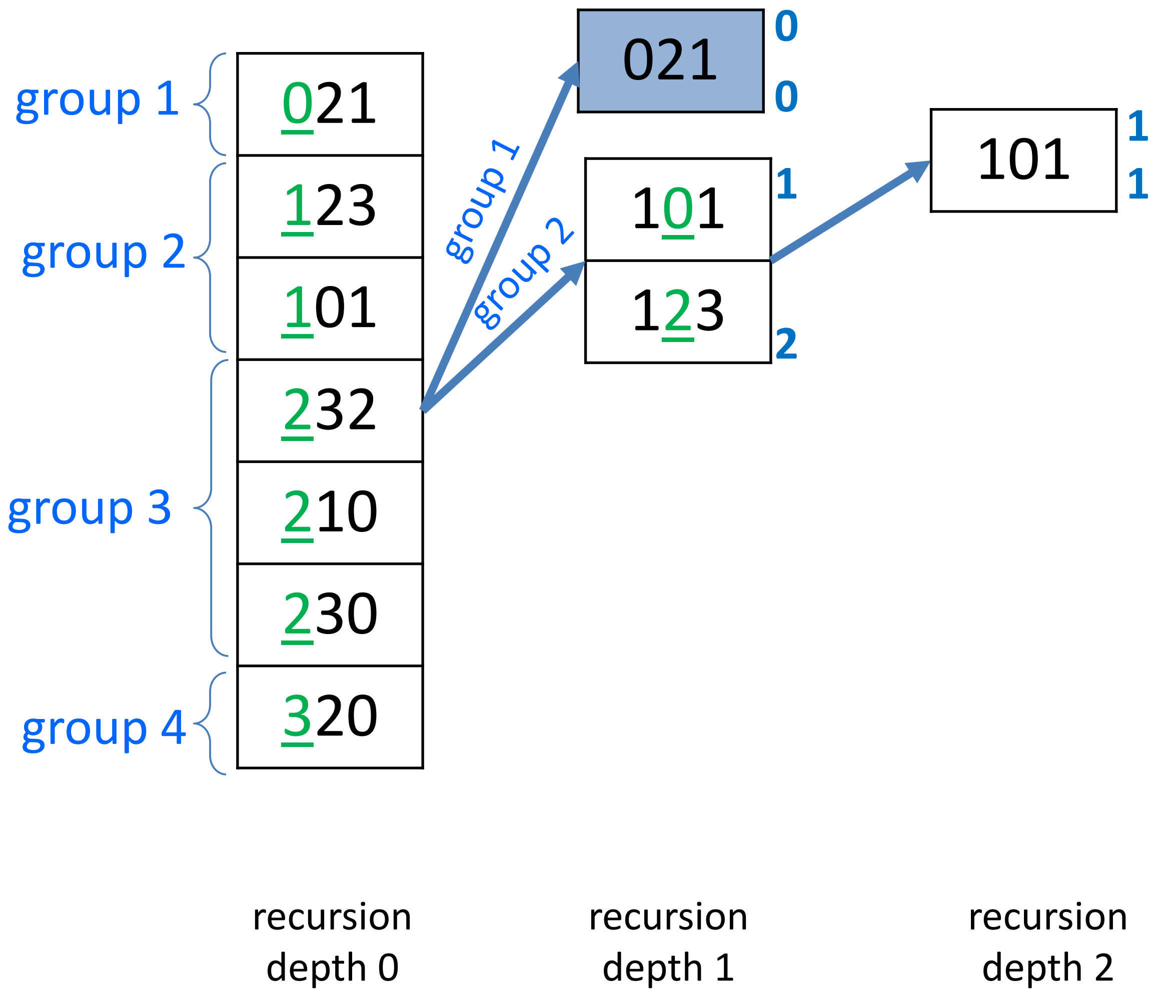
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



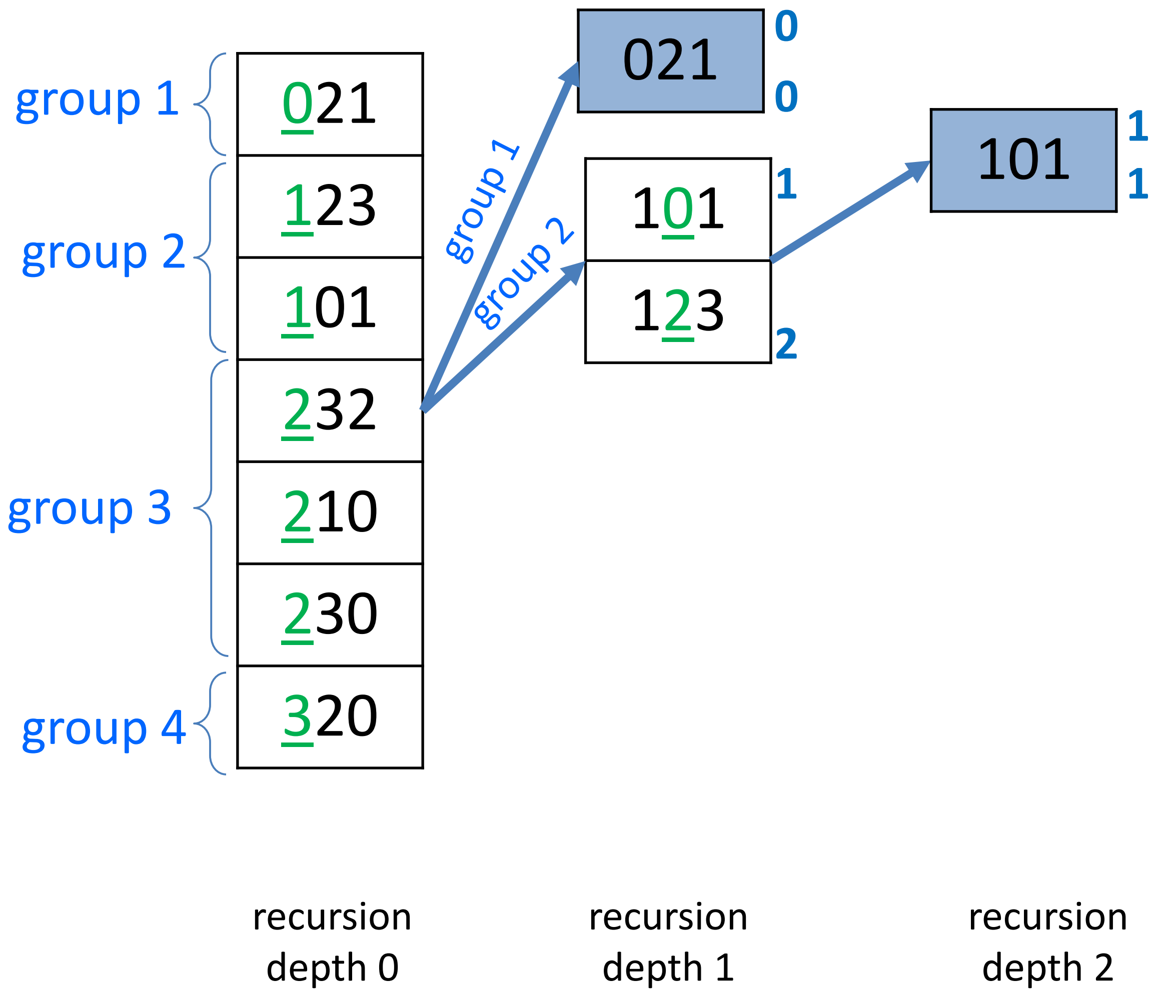
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



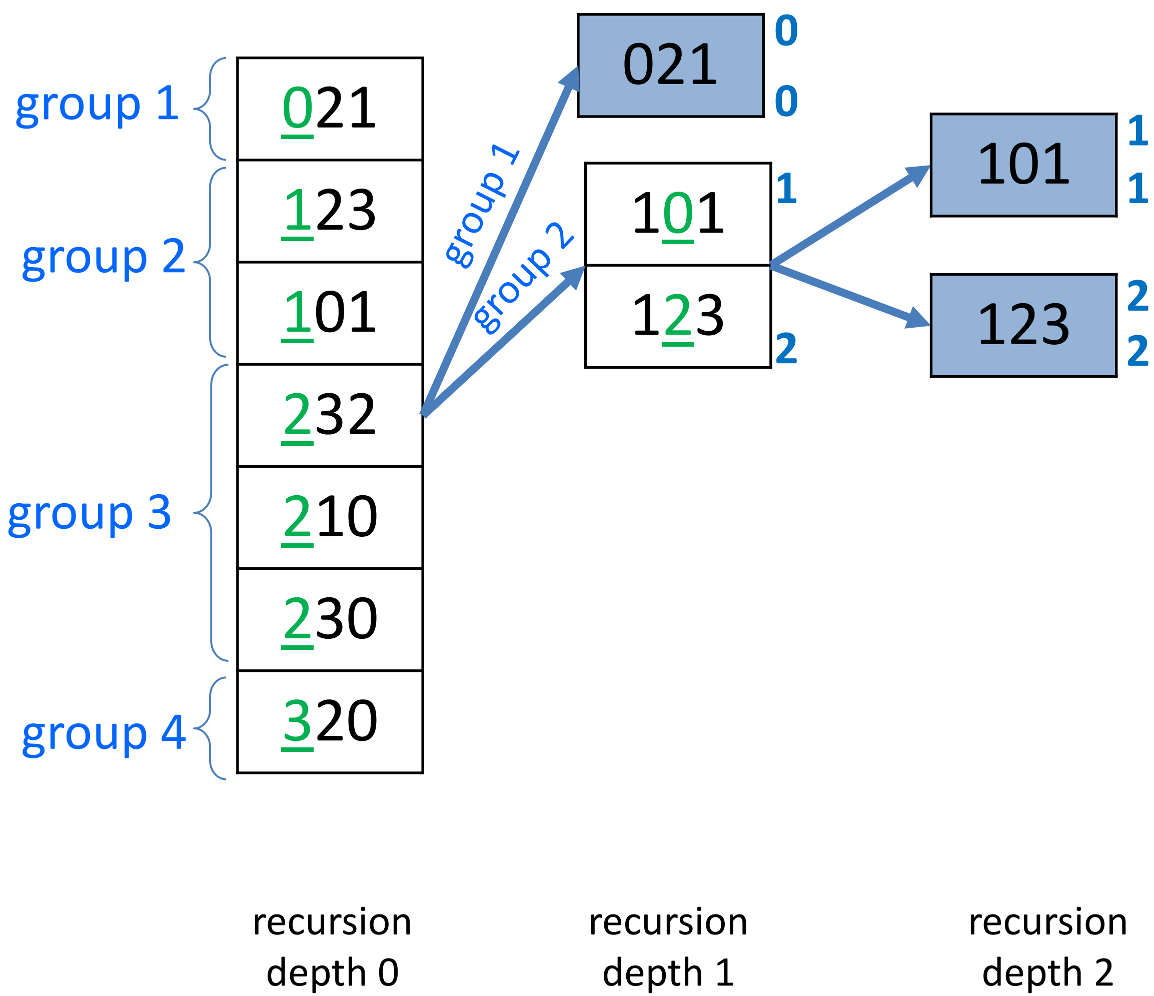
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



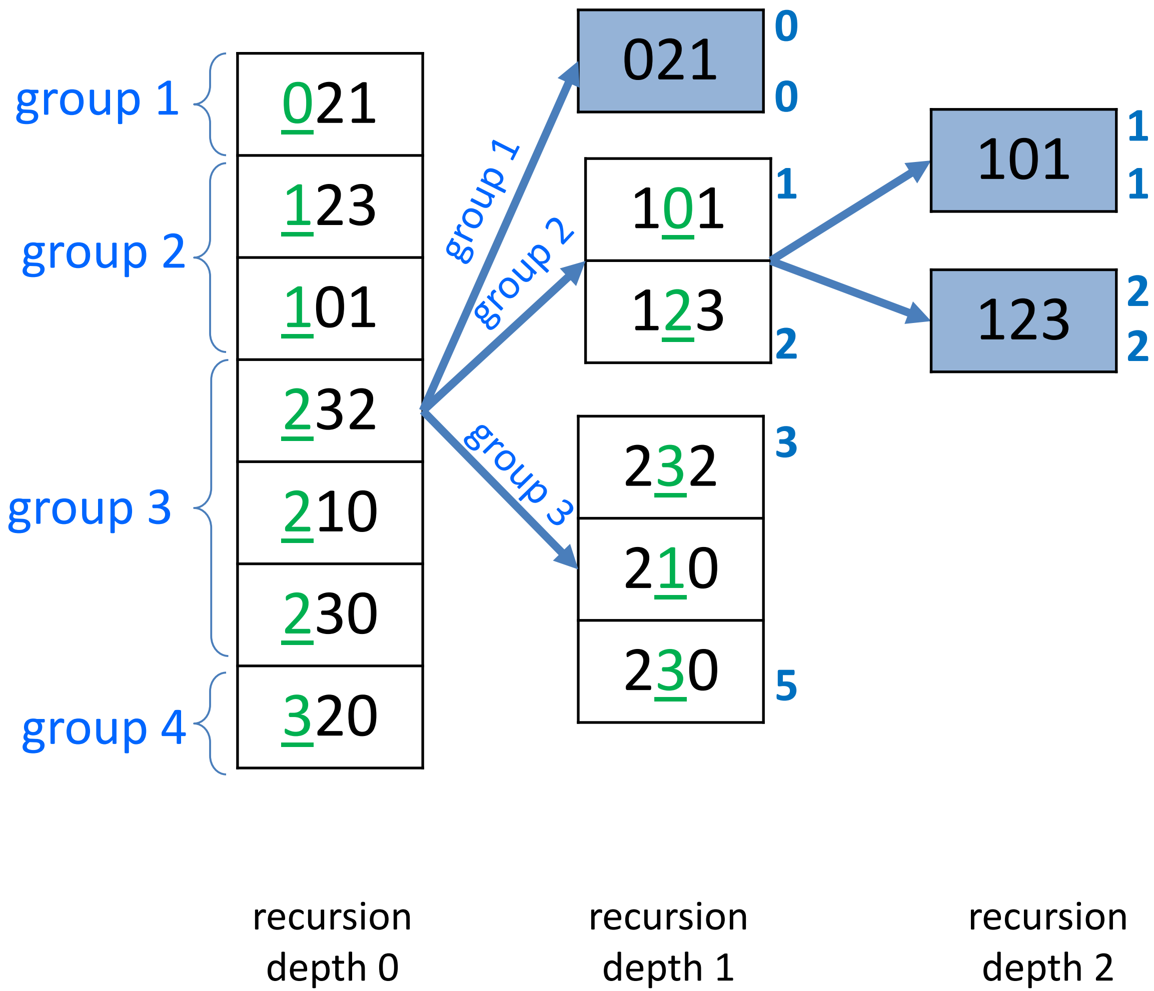
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



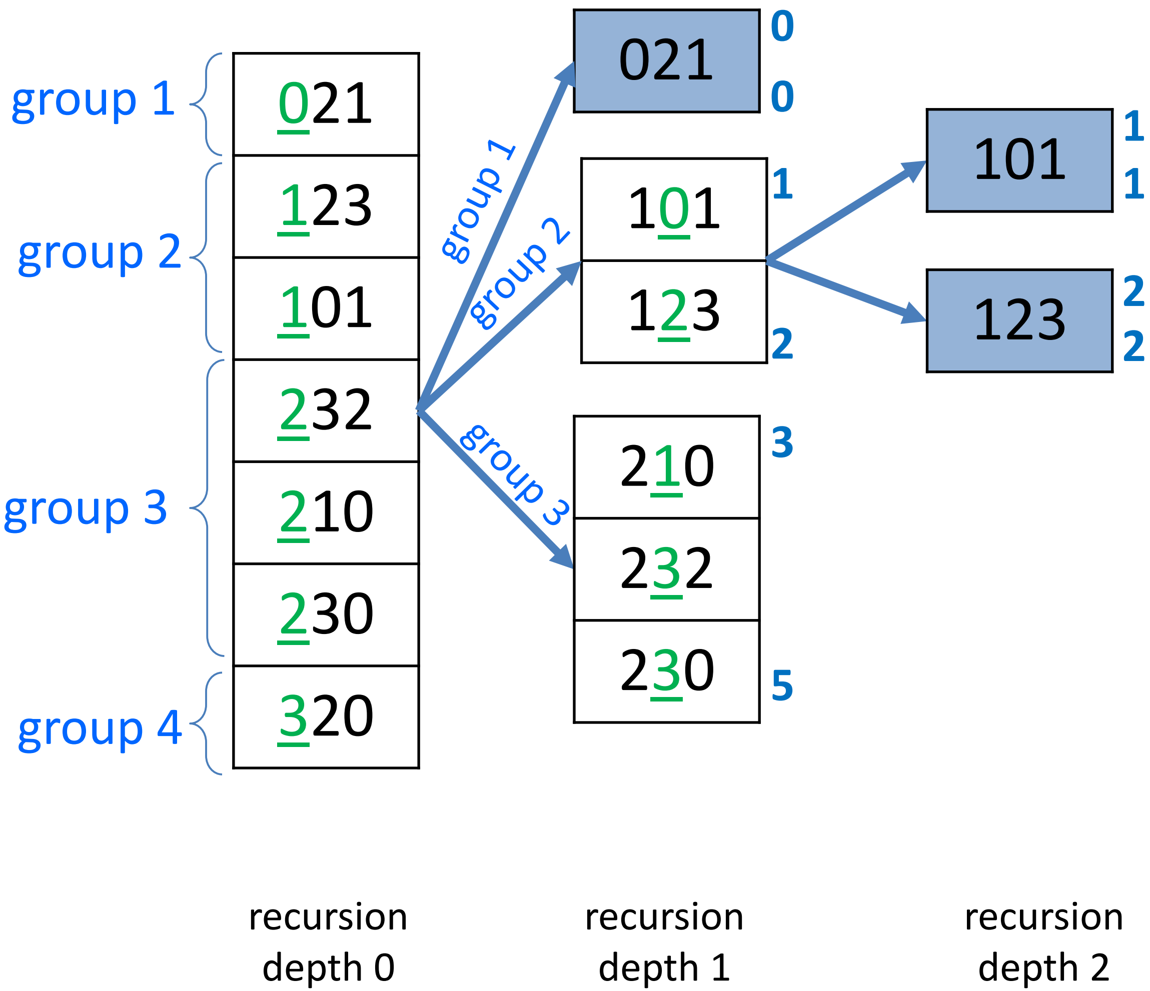
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



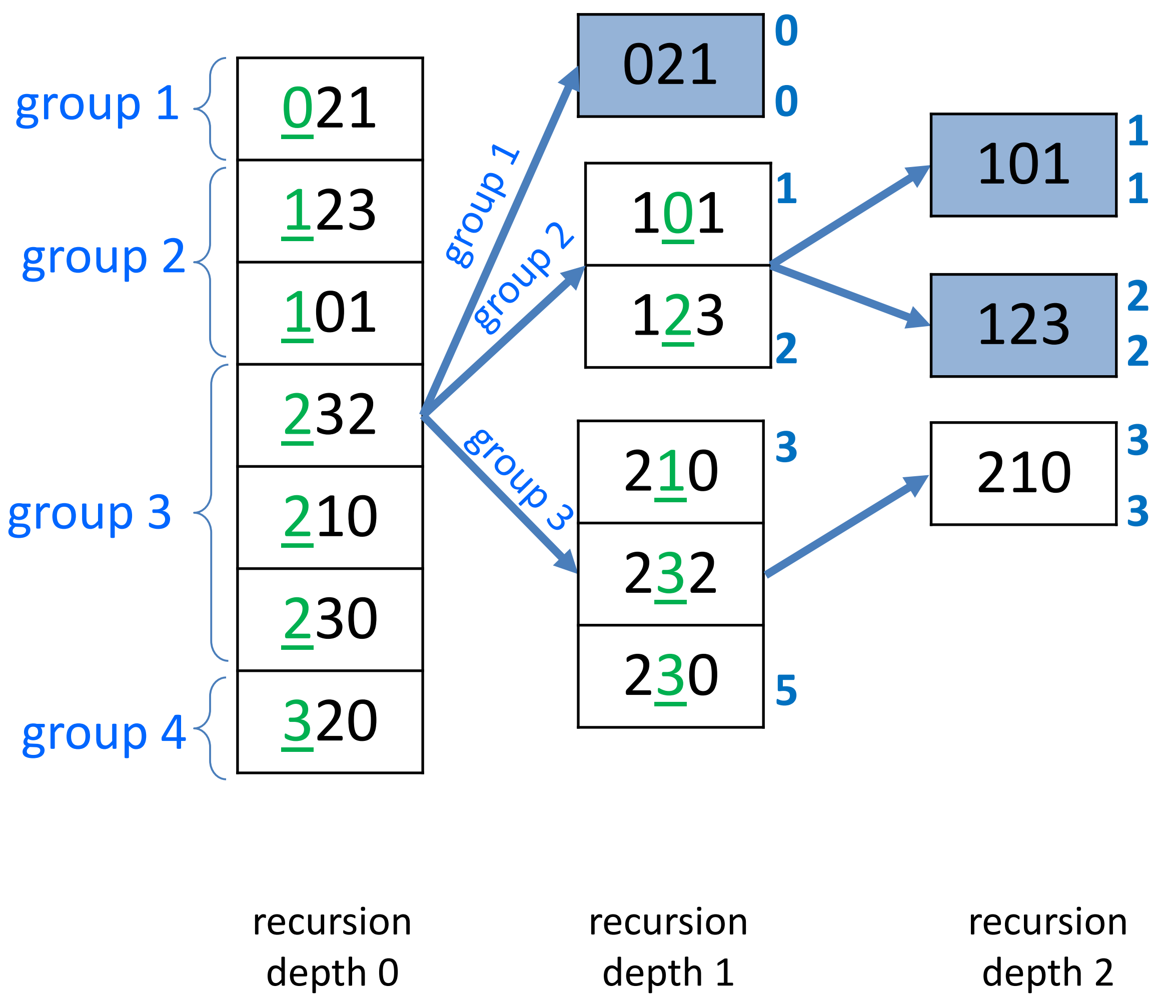
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



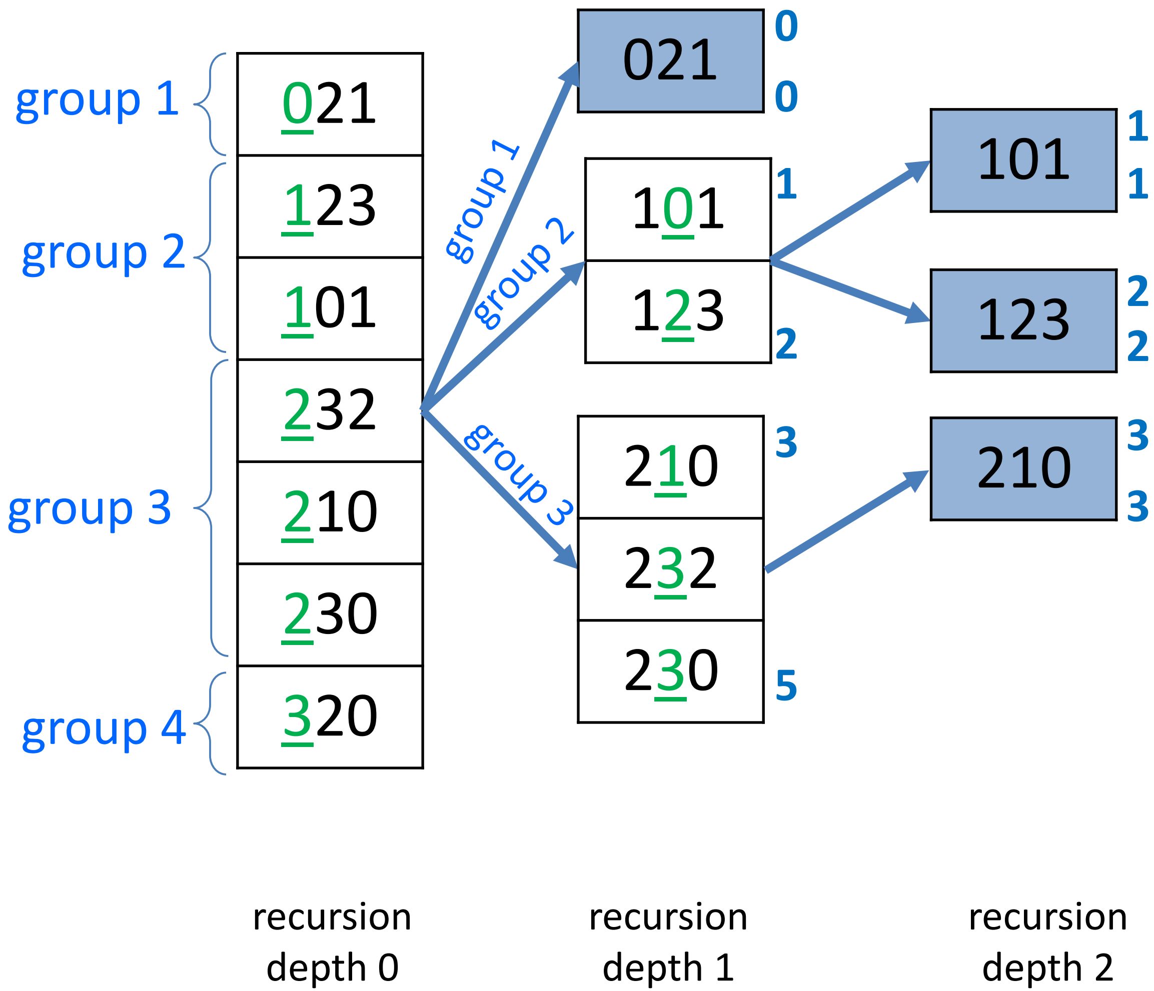
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



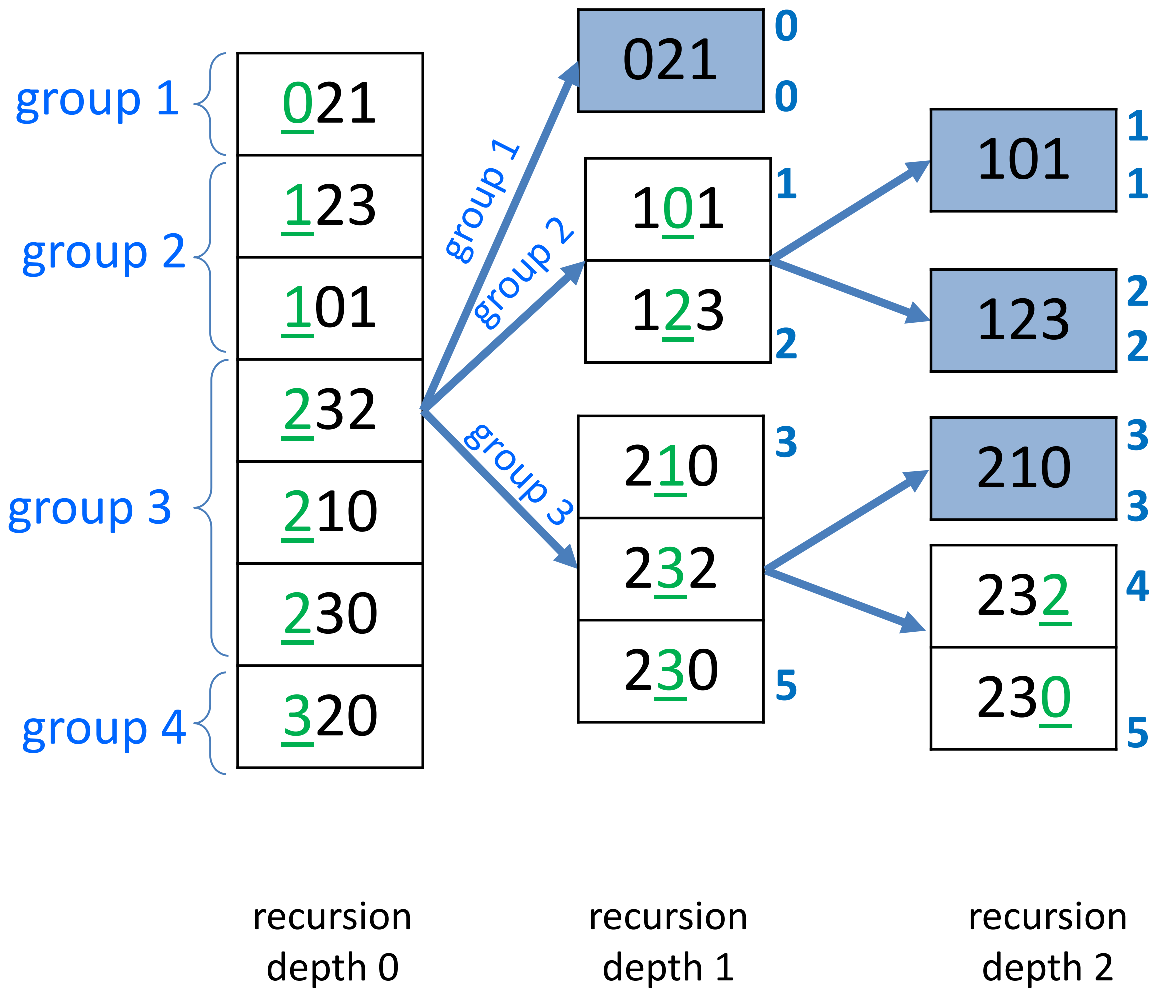
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



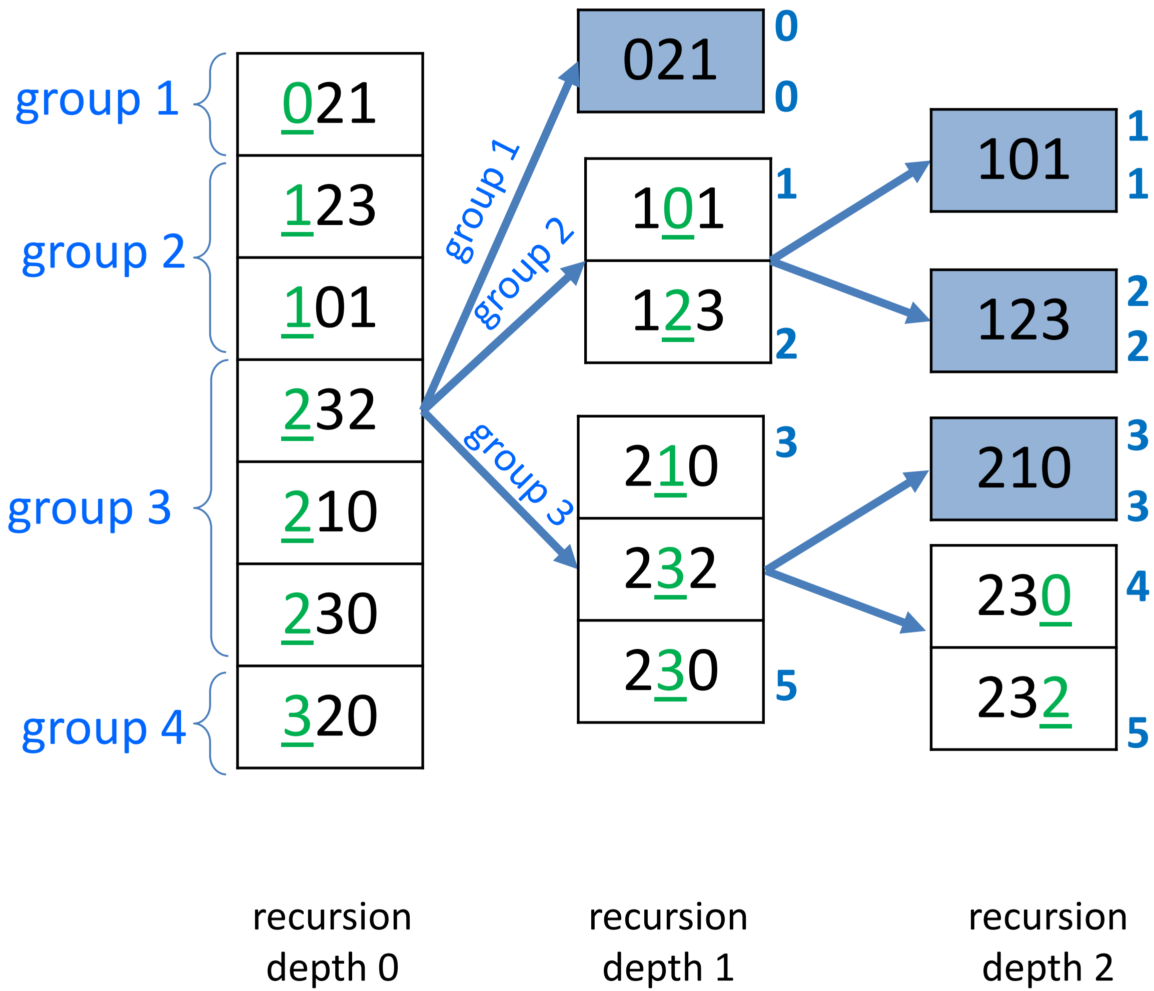
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



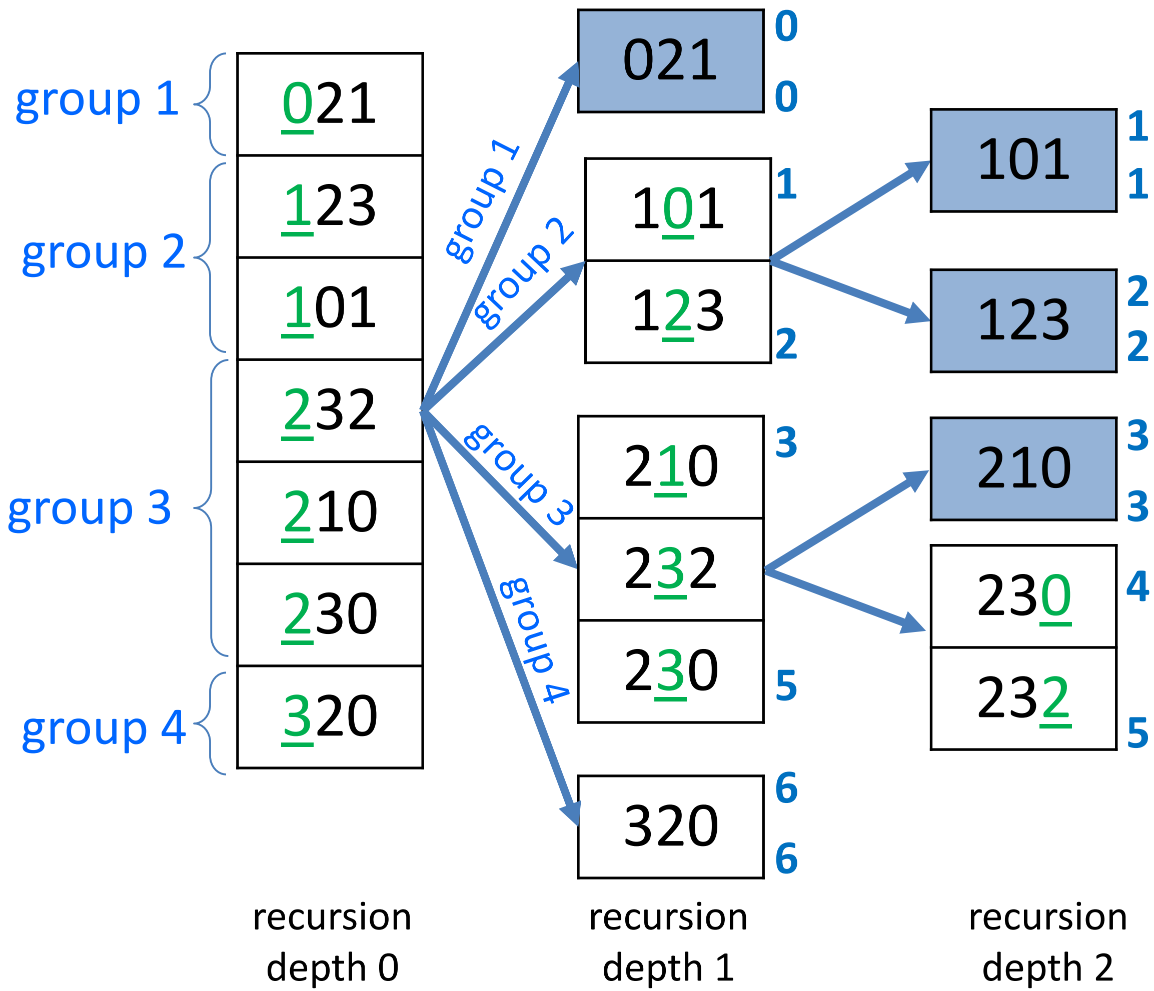
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



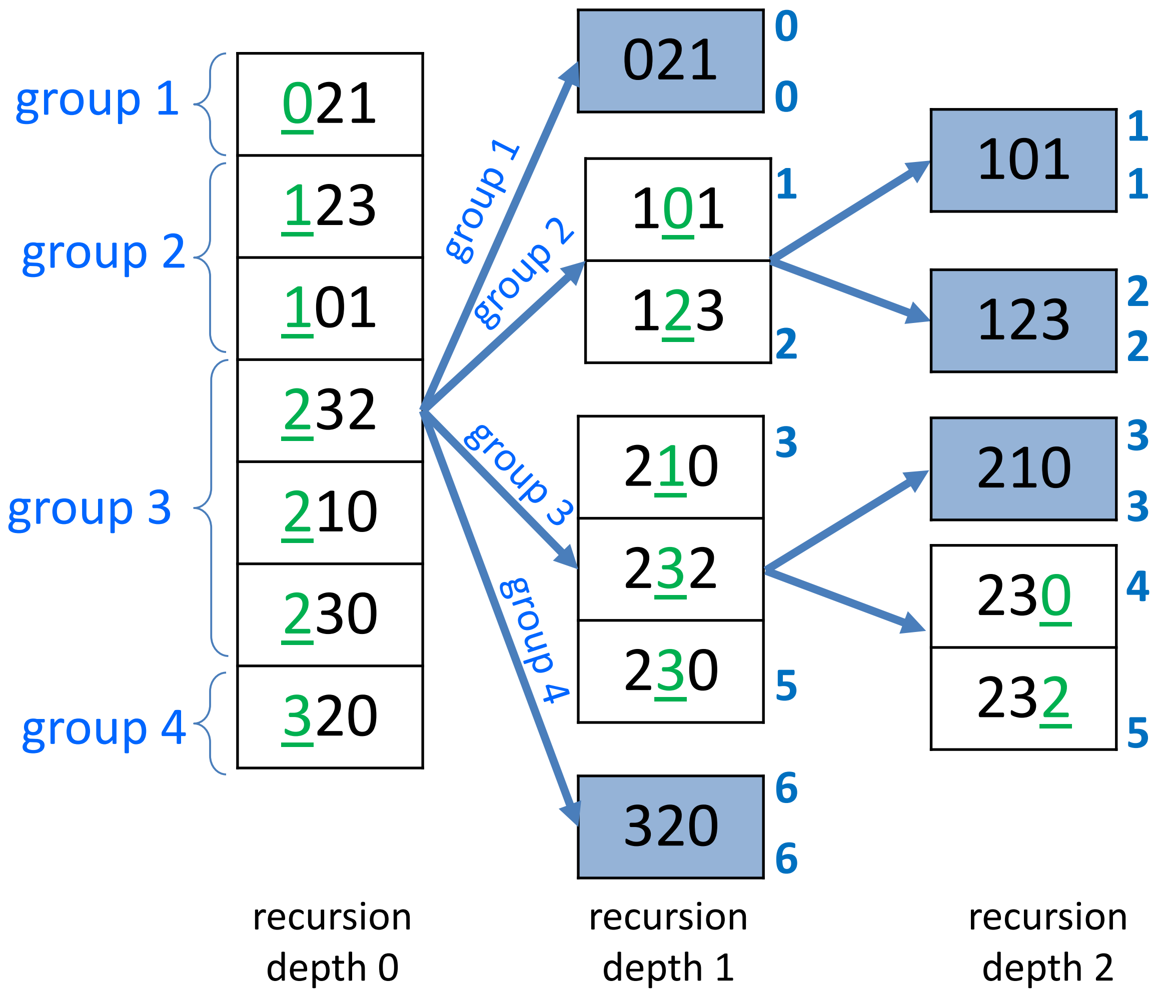
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



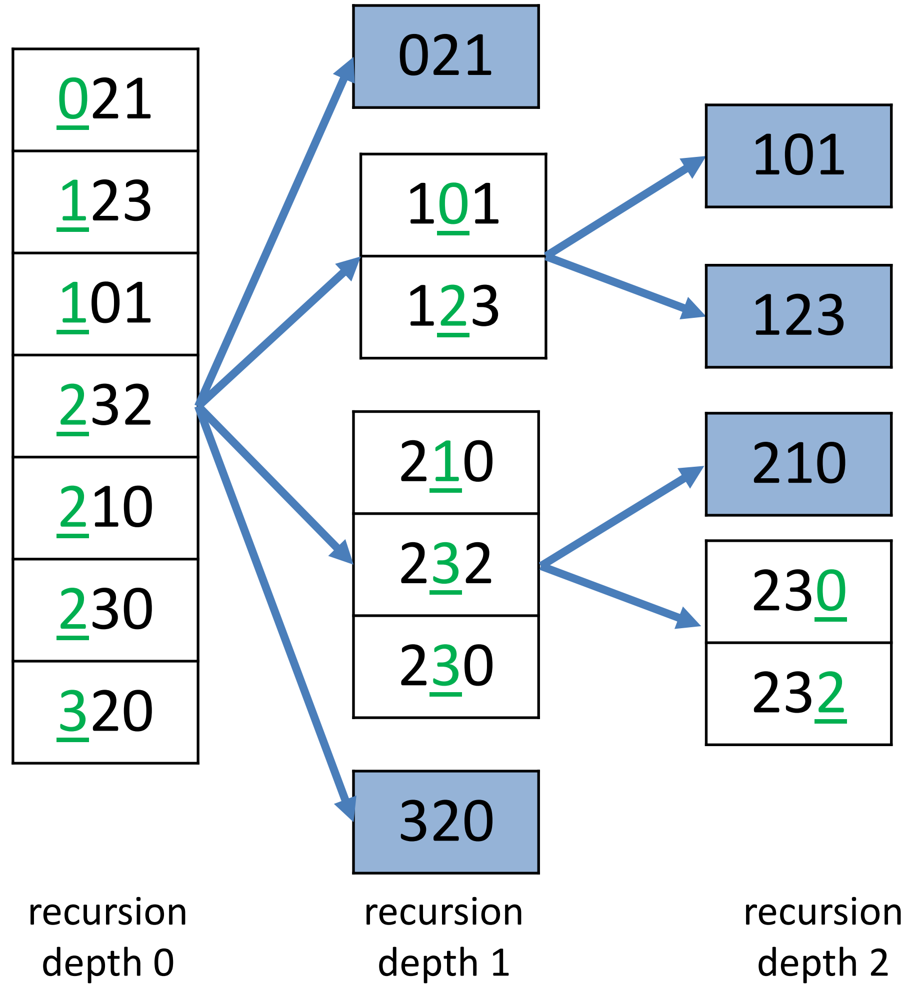
# MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



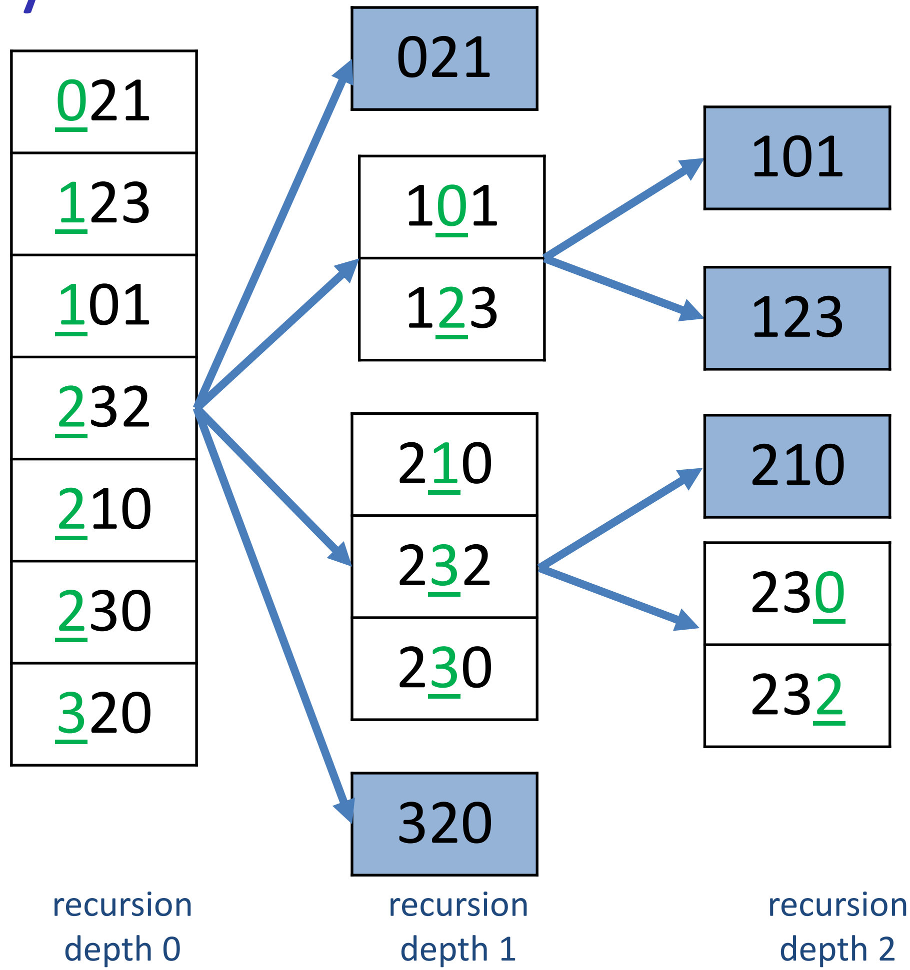
# MSD-Radix-Sort Space Analysis

- Bucket-sort
  - auxiliary space $\Theta(n + R)$
- Recursion depth is $m - 1$
  - auxiliary space $\Theta(m)$
- Total auxiliary space $\Theta(n + R + m)$



# MSD-Radix-Sort Time Analysis

- Time spent for each recursion depth
  - Depth 0
    - one bucket sort on $n$ items
    - $\Theta(n+R)$
  - All other depths
    - lets $k$ be the number of bucket sorts at each depth
    - $k \le n$
      - cannot have more bucket sorts than the array size
    - each bucket sort is on $n_i$ items
    - $\sum_{i=0}^{k} n_i = n$
    - each bucket sort is $n_i + R$
    - $\sum_{i=0}^{k}(n_i + R) = n + \sum_{i=0}^{k} R \le n + nR$
    - total time at any depth is $O(nR)$
- Number of depths is at most $m-1$
- Total time $O(mnR)$



# MSD-Radix-Sort Time Analysis

- Total time $O(mnR)$
- This is $O(n)$ if sort items in limited range
  - suppose $R = 2$, and we sort are $n$ integers in the range $[0, 2^{10})$
  - then $m = 10,\ R = 2,\ \text{and sorting is } O(n)$
    - note that $n$, the number of items to sort, can be arbitrarily large

# MSD-Radix-Sort Time Analysis

- Total time $O(mnR)$
- This is $O(n)$ if sort items in limited range
  - suppose $R = 2$, and we sort are $n$ integers in the range $[0, 2^{10})$
  - then $m = 10,\ R = 2,$ and sorting is $O(n)$
    - note that $n$, the number of items to sort, can be arbitrarily large
- This does not contradict $\Omega(n \log n)$ bound on the sorting problem, since the bound applies to comparison-based sorting

# MSD-Radix-Sort Pseudocode

- Sorts array of $m$-digit radix-$R$ numbers recursively
- Sort by leading digit, then each group by next digit, etc.

*MSD-Radix-sort*$(A,\ l \leftarrow 0, r \leftarrow n-1, d \leftarrow leading\ digit\ index)$

$l, r$ : indexes between which to sort, $0 \leq l, r \leq n-1$

    **if** $l < r$

        *bucket-sort*$(A\,[l \ldots r],\ d)$

        **if** there are digits left

            $l' \leftarrow l$

            **while** $(l' < r)$ **do**

                let $r' \geq l'$ be the maximal s.t $A\,[l' \ldots r']$ have the same $d$th digit

                *MSD-Radix-sort*$(A, l', r', d+1)$

                $l' \leftarrow r' + 1$

- Run-time $O(mnR)$
- Auxiliary space is $\Theta(m+n+R)$ for bucket sort and recursion stack
- Drawback of *MSD-Radix-sort* is many recursions

# LSD-Radix-Sort

- **Idea**: apply single digit bucket sort from least significant digit to the most significant digit
- Observe that digit bucket sort is stable
  - equal elements stay in the original order
  - therefore, we can apply single digit bucket sort to the **whole array**, and the output will be sorted after iterations over all digits

# LSD-Radix-Sort

| | | | | | |
|---|---|---|---|---|---|
| 123 | 230 | 230 | 101 | 101 | 101 |
| 230 | 320 | 320 | 210 | 210 | 121 |
| 121 | 210 | 210 | 320 | 320 | 123 |
| 320 | 121 | 121 | 121 | 121 | 210 |
| 210 | 101 | 101 | 123 | 123 | 230 |
| 232 | 232 | 232 | 230 | 230 | 232 |
| 101 | 123 | 123 | 232 | 232 | 320 |
| prepare to sort by last digit | last digit sorted | prepare to sort by middle digit | last two digits sorted | prepare to sort by first digit | last three digits sorted |

- $m$ bucket sorts, on $n$ items each, one bucket sort is $\Theta(n + R)$
- Total time cost $\Theta(m(n + R))$

# LSD-Radix-Sort

*LSD-radix-sort*$(A)$
$A$: array of size *n*, contains *m*-digit radix-*R* numbers
**for** $d \leftarrow$ least significant **down to** most significant digit **do**
*bucket-sort*$(A, d)$

- Loop invariant: after iteration $i$, $A$ is sorted w.r.t. the last $i$ digits of each entry
- Time cost $\Theta(m(n + R))$
- Auxiliary space $\Theta(n + R)$

# Summary

- Sorting is an important and *very* well-studied problem
- Can be done in $\Theta(n \log n)$ time
    - faster is not possible for general input
- HeapSort is the only $\Theta(n \log n)$ time algorithm we have seen with $O(1)$ auxiliary space
- MergeSort is also $\Theta(n \log n)$ time
- Selection and insertion sorts are $\Theta(n^2)$
- QuickSort is worst-case $\Theta(n^2)$, but often the fastest in practice
- BucketSort and RadixSort can achieve $o(n \log n)$ if the input is special
- Best-case, worst-case, average-case can all differ
- Randomized algorithms can eliminate “bad cases”, resulting in the same expected time for all cases