CS 240 – Data Structures and Data Management

Module 3: Sorting, Average-case and Randomization

T. Biedl E. Kondratovsky M. Petrick O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2022

Outline

- Sorting, Average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

Outline

- Sorting, Average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

Average Case Analysis

- Worst-case run time: our default for analysis
- Best-case run time: sometimes useful
- For many algorithms, best-case and worst case runtimes are the same
- But for some algorithms best-case and worst case differ significantly
    - worst-case runtime can be too pessimistic, best-case too optimistic
    - true for many algorithms we study in this module
    - average-case run time analysis is useful especially in such cases
- Recall average case runtime definition
    - let $\mathbb{I}_n$ be the set of all instances of size n

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|}$$

- Pros
    - more accurate picture of how an algorithm performs in practice
        - **provided all instances are equally likely**
- Cons
    - usually difficult to compute
    - average-case and worst case run times are often the same (asymptotically)

Average Case Analysis: Example 1

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|}$$

$\color{blue}{sortednessTester}(A, n)$
A: array storing n distinct numbers
for $i \leftarrow 1\ to\ n-1$ **do**
    if $A[i-1] > A[i]$ **then return** *false*
return *true*

- Best-case is $O(1)$, worst case is $\Theta(n)$
- For average case, need to take average running time over **all** inputs
- How to deal with infinite $\mathbb{I}_n$?
    - there are infinitely many arrays of n numbers

Average Case Analysis: Example 1

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|}$$

$sortednessTester(A, n)$
A: array storing n distinct numbers
for $i \leftarrow 1$ *to* $n-1$ **do**
  if $A[i-1] > A[i]$ **then return** *false*
return *true*

- Observe: $sortednessTester$ acts the same on two inputs below

14	22	43	6	1	11	7

15	23	44	5	1	12	8

- Only the relative order matters, not the actual numbers
    - true for many (but not all) algorithms
    - if true, can use this to simplify average case analysis

Sorting Permutations

- Characterize input by its sorting permutation π
  - sorting permutation tells us how to sort the array
    - stores array indexes in the order corresponding to the sorted array

	0	1	2	3	4	5	6
A	14	2	3	5	1	11	7

$$\pi = (4, 1, 2, 3, 6, 5, 0)$$

$\pi(0)$

$\pi(1)$

$\pi(2)$ $\pi(6)$

$$A[\pi(0)] \leq A[\pi(1)] \leq A[\pi(2)] \leq A[\pi(3)] \leq A[\pi(4)] \leq A[\pi(5)] \leq A[\pi(6)]$$

$$1 \leq 2 \leq 3 \leq 5 \leq 7 \leq 11 \leq 14$$

sorted!

Sorting Permutations

- Arrays with the same relative order have the same sorting permutations

	0	1	2	3	4	5	6
A	15	3	4	6	1	12	8

$$\pi = (4, 1, 2, 3, 6, 5, 0)$$

$$A[\pi(0)] \leq A[\pi(1)] \leq A[\pi(2)] \leq A[\pi(3)] \leq A[\pi(4)] \leq A[\pi(5)] \leq A[\pi(6)]$$

$$1 \leq 3 \leq 4 \leq 6 \leq 8 \leq 12 \leq 15$$

sorted!

Average Time with Sorting Permutations

- There are $n!$ sorting permutations for arrays of size n
  - let Π_n be the set of all sorting permutations of size n
    - $\Pi_3 = \{(0,1,2), (0,2,1), (1,0,2), (2,0,1), (1,2,0), (2,1,0)\}$
- Define average cost is the sum of costs of all permutations, divided by $n!$

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|} = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

- Averaging 'by parts': to average over a set, can divide the set into *equal* parts, average over each individual part, and then average the individual averages

3	**2**	**3**
5	**7**	**8**
4	**5**	**1**
8	**9**	**8**

average = 5.25

3	**2**	**3**	average = 8/3
5	**7**	**8**	average = 20/3
4	**5**	**1**	average = 10/3
8	**9**	**8**	average = 25/3

average = 5.25

Average Time with Sorting Permutations

- Average cost is the sum of costs of all permutations, divided by $n!$

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|} = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

...
$(7, 20, 10)$
$(-3, 6.6, 1.8)$
$(10, 21, 13)$
...

infinite set

instances with sorting permutation $\pi = (0, 1, 2)$
instances with sorting permutation $\pi = (0, 2, 1)$
instances with sorting permutation $\pi = (1, 0, 2)$
instances with sorting permutation $\pi = (2, 0, 1)$
instances with sorting permutation $\pi = (1, 2, 0)$
instances with sorting permutation $\pi = (2, 1, 0)$

$T(0, 2, 1)$

all instances of size 3

- Defining average for infinite set is tricky, but since running time is the same *number* for each element of the set, intuitively, the average should be equal to that *number*
- Do these subsets have equal size?
  - instead of allowing an infinite set of numbers, suppose there are m numbers in total
  - each subset has size $\binom{m}{3}$

Average Time with Sorting Permutations

- Average cost is the sum of costs of all permutations, divided by $n!$

$$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{|\mathbb{I}_n|} = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

…
$(7, 20, 10)$
$(-3, 6.6, 1.8)$
$(10, 21, 13)$
…

infinite set

instances with sorting permutation $\pi = (0, 1, 2)$	$T(0, 1, 2)$	average
instances with sorting permutation $\pi = (0, 2, 1)$	$T(0, 2, 1)$	
instances with sorting permutation $\pi = (1, 0, 2)$	$T(1, 0, 2)$	
instances with sorting permutation $\pi = (2, 0, 1)$	$T(2, 0, 1)$	
instances with sorting permutation $\pi = (1, 2, 0)$	$T(1, 2, 0)$	
instances with sorting permutation $\pi = (2, 1, 0)$	$T(2, 1, 0)$	

all instances of size 3

- Defining average for infinite set is tricky, but since running time is the same *number* for each element of the set, intuitively, the average should be equal to that *number*
- Do these subsets have equal size?
  - instead of allowing an infinite set of numbers, suppose there are m numbers in total
  - each subset has size $\binom{m}{3}$

Average Case Analysis: Example 1

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

sortednessTester(A, n)
A: array storing n distinct numbers
for $i \leftarrow 1$ *to* $n-1$ **do**
    if $A[i-1] > A[i]$ **then return** *false*
return *true*

- Runtime is proportional to the number of comparisons
- So let $T(\pi)$ be the number of comparisons
  - for some permutations π, do exactly 1 comparison: $T(\pi) = 1$
  - for some permutations π, do exactly 2 comparisons: $T(\pi) = 2$
  - …
  - for some permutations π, do exactly $n-1$ comparisons: $T(\pi) = n-1$
- Average running time

$$T^{avg}(n) = \frac{1}{n!} \sum_{k=1}^{n-1} k \cdot (\text{\#permutations with exactly } k \text{ comparisons})$$

Average Case Analysis: Example 1

$$T^{avg}(n) = \frac{1}{n!}\sum_{k=1}^{n-1} k \cdot (\text{\#permutations with exactly } k \text{ comparisons})$$

perm with exactly k comp
perm with exactly $k+1$ comp
perm with exactly $k+2$ comp
…
perm with exactly $n-1$ comp

perm with exactly $k+1$ comp
perm with exactly $k+2$ comp
…
perm with exactly $n-1$ comp

#permutations with at least k comparisons

− #permutation with at least $k+1$ comparisons

#permutations with exactly k comparisons

$$T^{avg}(n) = \frac{1}{n!}\sum_{k=1}^{n-1} k \cdot (\text{\#perm with at least } k \text{ comp} - \text{\#perm with at least } k+1 \text{ comp})$$

Average Case Analysis: Example 1

$$T^{avg}(n) = \frac{1}{n!}\sum_{k=1}^{n-1} k \cdot (\#\text{perm with at least } k \text{ comp} - \#\text{perm with at least } k+1 \text{ comp})$$

- Permutations with at least 1 comparison
  - all $n!$ permutations
- Permutations with at least 2 comparisons
  - $A[0] < A[1]$
  - $0, 1$ occur in sorted order : $(4,3,2,0,1)$, $(4,3,0,2,1)$, $(4,0,3,2,1)$
  - $\binom{n}{2}(n-2)!$
- Permutations with at least 3 comparisons
  - $0, 1, 2$ occur in sorted order : $(4,3,0,1,2)$, $(4,0,3,1,2)$, $(0,1,3,4,2)$
  - $\binom{n}{3}(n-3)!$
- Permutations with at least k comparisons
  - $0, 1, \dots, k$ occur in sorted order
  - $\binom{n}{k}(n-k)! = \dfrac{n!}{k!}$

Average Case Analysis: Example 1

- Let π_k stand for # of permutations with at least k comparisons
  - there are $\frac{n!}{k!}$ of them
- From Taylor expansion, $\sum_{k=0}^{\infty} \frac{1}{k!} = e \approx 2.8$

$$T^{avg}(n) = \frac{1}{n!}\sum_{k=1}^{n-1} k \cdot (\pi_k - \pi_{k+1}) = \frac{1}{n!}\left(\sum_{k=1}^{n-1} k \cdot \pi_k - \sum_{k=1}^{n-1} k \cdot \pi_{k+1}\right)$$

$$= \frac{1}{n!}(1 \cdot \pi_1 + \underline{2 \cdot \pi_2} + \underline{\underline{3 \cdot \pi_3}} \ldots + (n-1) \cdot \pi_{n-1} - \underline{1 \cdot \pi_2} - \underline{\underline{2 \cdot \pi_3}} - \cdots - (n-1) \cdot \pi_n$$

$$= \frac{1}{n!}(\pi_1 + \pi_2 + \pi_3 \ldots + \pi_{n-1} - (n-1) \cdot \underbrace{\boxed{\pi_n}}_{=0})$$

$$= \frac{1}{n!}\sum_{k=1}^{n-1} \pi_k \quad = \frac{1}{n!}\sum_{k=1}^{n-1} \frac{n!}{k!} \quad = \sum_{k=1}^{n-1} \frac{1}{k!} < 2.8$$

- Average running time of $sortednessTester(A, n)$ is $O(1)$
  - much better than the worst case $\Theta(n)$

Average Case Analysis: Example 2

$avgCaseDemo(A, n)$
A: array storing n distinct numbers
if $n \leq 2$ **return**
if $A[n-2] < A[n-1]$ **then** $avgCaseDemo(A[0,\ n/2-1,\ n/2)$ // good case
else $avgCaseDemo(A[0,\ n-3,\ n-2)$ // bad case

- Let $T(n)$ be the number of recursions
  - asymptotically the same as running time
- Best case (array sorted in increasing order)
  - always get the good case, array size is divided by 2 at each recursion
  - $T(n) = T(n/2) + 1$
  - resolves to $\Theta(\log(n))$
- Worst case (array sorted in decreasing order)
  - always get the bad case, array size decreases by 2 at each recursion
  - $T(n) = T(n-2) + 1$
  - resolves to $\Theta(n)$
- Average case?

Average Case Analysis: Example 2

$avgCaseDemo(A, n)$

A: array storing n distinct numbers

if $n \leq 2$ **return**

if $A[n-2] < A[n-1]$ **then** $avgCaseDemo(A\ [0,\ n/2-1,\ n/2])$ // good case

else $avgCaseDemo(A[0,\ n-3,\ n-2])$ // bad case

- $avgCaseDemo$ runtime is equal for instances with same relative element order
- Again, use sorting permutations to compute average running time

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

- Call permutation π is good if it leads to a good case
  - ex: $(0, 1, 3, 2, 4)$
- Call permutation π bad if it leads to a bad case
  - ex: $(1, 4, 0, 2, 3)$
- Exactly half of the permutations are good
  - $(0, 1, 3, 2, 4) \leftrightarrow (0, 1, 4, 2, 3)$
  - $n!/2$ good permutations, $n!/2$ bad permutations

Average Case Analysis: Example 2

$avgCaseDemo(A, n)$

A: array storing n distinct numbers

if $n \le 2$ **return**

if $A[n-2] < A[n-1]$ **then** $avgCaseDemo(A[0,\ n/2-1,\ n/2])$ // good case

else $avgCaseDemo(A[0,\ n-3,\ n-2])$ // bad case

- For recursive algorithms, we typically derive recurrence equation and solve it
- Easy to derive recursive formula for one instance π

$$T(\pi) = \begin{cases} 1 + T(\text{first } \dfrac{n}{2} \text{ items}) & \text{if } \pi \text{ is good} \\ 1 + T(\text{first } n-2 \text{ items}) & \text{if } \pi \text{ is bad} \end{cases}$$

- Tempting, but incorrect $\cancel{T^{avg}(n) = \begin{cases} 1 + T^{avg}(n/2) & \text{if } \pi \text{ is good} \\ 1 + Tavgg(n-2) & \text{if } \pi \text{ is bad} \end{cases}}$

- Can derive formula for the **sum** of instances π (but it is not trivial)

$$\sum_{\pi \in \Pi_n} T(\pi) = \sum_{\pi \in \Pi_n:\ \pi \text{ is good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n:\ \pi \text{ is bad}} (1 + T^{avg}(n-2))$$

Average Case Analysis: Example 2

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

$$\sum_{\pi \in \Pi_n} T(\pi) = \sum_{\pi \in \Pi_n:\ \pi \text{ is good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n:\ \pi \text{ is bad}} (1 + T^{avg}(n-2))$$

- Can derive recurrence equation for the average case

$$T^{avg}(n) = \frac{1}{n!}\left(\sum_{\pi \in \Pi_n: \pi \text{ is good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n: \pi \text{ is bad}} (1 + T^{avg}(n-2)) \right)$$

$$= \frac{1}{n!}\left(\frac{n!}{2}(1 + T^{avg}(n/2)) + \frac{n!}{2}(1 + T^{avg}(n-2)) \right)$$

- Simplifying,

$$T^{avg}(n) = 1 + \tfrac{1}{2} T^{avg}(n/2) + \tfrac{1}{2} T^{avg}(n-2)$$

Average Case Analysis: Example 2

$$T^{avg}(n) = 1 + \frac{1}{2} T^{avg}(\,n/2) + \frac{1}{2} T^{avg}(\,n-2)$$

Theorem: $T^{avg}(n) \le 2\log(n)$

Proof: (by induction)

- true for $n \le 2$ (no recursion in these cases, $T^{avg}(n) = 0$)
- assume $n \ge 3$ and the theorem holds for all $m < n$
- $T^{avg}(n) = 1 + \frac{1}{2}\underbrace{T^{avg}(\,n/2)}_{\text{induction hypothesis}} + \frac{1}{2}\underbrace{T^{avg}(\,n-2)}_{\text{induction hypothesis}}$

$$\le 1 + \frac{1}{2} 2\log(n/2) + \frac{1}{2} 2\log(n-2)$$

$$\le 1 + \frac{1}{2} 2(\log(n) - 1) + \frac{1}{2} 2\log(n)$$

$$= 2\log(n)$$

- Therefore, average-case running time is $O(\log(n))$
  - better than worst case $\Theta(n)$

Outline

- Sorting, average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

Randomized Algorithms: Motivation

- Suppose an algorithm has a better average-case than worst-case runtime
  - if any instance is equally likely, then such algorithm is good "as is"
  - but humans often generate instances that are far from equally likely
    - most often we sort data which is already almost sorted
  - randomization improves runtime when instances are not equally likely

$avgCaseDemo(A, n)$
A: array storing n distinct numbers
if $n \leq 2$ **return**
if $A[n-2] < A[n-1]$ **then** $avgCaseDemo(A[0,\ n/2-1,\ n/2)$ // good case
else $avgCaseDemo(A[0,\ n-3,\ n-2)$ // bad case

- Recall $avgCaseDemo$ has worst case $\Theta(n)$, average case $O(\log(n))$
- If user mostly calls $avgCaseDemo$ on array that is almost reverse sorted, running time, on average will be $\Theta(n)$
- If we shuffle array A before calling $avgCaseDemo$, probability of A being almost reverse sorted is tiny
  - on average, runtime will be $O(\log(n))$
  - shifted dependence from what we cannot control (user) to what we can control (random number generation)

Randomized Algorithms

- A *randomized algorithm* is one which relies on some random numbers in addition to the input
- The runtime 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 runtime 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 runtime* $T^{exp}(I)$ 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 runtime*

$$T^{exp}(n) = \max_{I \in \mathbb{I}_n} T^{exp}(I)$$

- Could also talk about best-case and average-case expected running time
- However, in this course, we only consider worst-case expected running time
  - usally a randomized algorithm is designed so that all instances of size n have the same expected run 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, i.e. worst case

$$\max_{R} \max_{I \in \mathbb{I}_n} T(I,R)$$

Randomized Algorithm $expectedDemo$

$expectedDemo(A, n)$

A: array storing n distinct numbers

if $n \leq 2$ **return**

if $random(2)$ **swap** $A[n-2]$ **and** $A[n-1]$

if $A[n-2] < A[n-1]$ **then** $expectedDemo(A[0,\ n/2-1,\ n/2)$ // good case

else $expectedDemo(A[0,\ n-3,\ n-2)$ // bad case

- Function $random(n)$ returns an integer sampled uniformly from $\{0, 1, \ldots, n-1\}$
- $\Pr(\text{good case}) = \Pr(\text{bad case}) = \frac{1}{2}$
    - for any array A
- As before, let $T(n)$ be the number of recursions
    - running time is proportional to the number of recursions

Expected running time of *expectedDemo*

$expectedDemo(A, n)$
A: array storing n distinct numbers
if $n \leq 2$ **return**
if $random(2)$ **swap** $A[n-2]$ **and** $A[n-1]$
if $A[n-2] < A[n-1]$ **then** $expectedDemo(A[0,\ n/2-1,\ n/2)$ // good case
else $expectedDemo(A[0,\ n-3,\ n-2)$ // bad case

- Number of recursions on array A if random outcomes are $R = \langle x,\ R' \rangle$

$$T(A,R) = T(A, \langle x, R' \rangle) = \begin{cases} 1 + T(A[0 \dots n/2-1],\ R') & \text{if } x \text{ is good} \\ 1 + T(A[0 \dots n-3],\ R') & \text{if } x \text{ is bad} \end{cases}$$

Expected running time of $expectedDemo$

$$T(A,R) = T(A, \langle x, R' \rangle) = \begin{cases} 1 + T(A[0 \ldots\ n/2-1],\ R') & \text{if } x \text{ is good} \\ 1 + T(A[0 \ldots n-3],\ R') & \text{if } x \text{ is bad} \end{cases}$$

- Summing up over all sequences of random outcomes

$$\sum_R T(A,R) \cdot \Pr(R) = \sum_{\langle x,R' \rangle} T(A, \langle x, R' \rangle) \cdot \Pr(x)\Pr(R')$$

$$= \sum_{\langle x=0,R' \rangle} T(A, \langle x, R' \rangle) \cdot \Pr(x)\Pr(R') + \sum_{\langle x=1,R' \rangle} T(A, \langle x, R' \rangle) \cdot \Pr(x)\Pr(R')$$

$$= \frac{1}{2} \sum_{\langle x=0,R' \rangle} T(A, \langle x, R' \rangle) \cdot \Pr(R') \quad + \frac{1}{2} \sum_{\langle x=1,R' \rangle} T(A, \langle x, R' \rangle) \cdot \Pr(R')$$

one of these is $1 + T(A[0 \ldots\ n/2-1],\ R')$, the other $1 + T(A[0 \ldots n-3],\ R')$

$$= \frac{1}{2} \sum_{R'} \big(1 + T(A[0 \ldots\ n/2-1], R')\big) \cdot \Pr(R') \ + \frac{1}{2} \sum_{R'} \big(1 + T(A[0 \ldots n-3], R')\big) \cdot \Pr(R')$$

Expected running time of *expectedDemo*

$$\sum_R T(A,R) \cdot \Pr(R) =$$

$$\frac{1}{2}\sum_{R'}\left(1 + T(A[0 \ldots n/2-1], R')\right) \cdot \Pr(R') + \frac{1}{2}\sum_{R'}\left(1 + T(A[0 \ldots n-3], R')\right) \cdot \Pr(R')$$

$$= \frac{1}{2}\sum_{R'} 1 \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T\left(A\left[0 \ldots \frac{n}{2}-1\right], R'\right) \cdot \Pr(R')$$

$$+ \frac{1}{2}\sum_{R'} 1 \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0 \ldots n-3], R') \cdot \Pr(R')$$

$$= 1 + \frac{1}{2}\sum_{R'} T\left(A\left[0 \ldots \frac{n}{2}-1\right], R'\right) \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0 \ldots n-3], R') \cdot \Pr(R')$$

$$\sum_{R'} T\left(A\left[0 \ldots \frac{n}{2}-1\right], R'\right) \cdot \Pr(R') \le \max_{A' \in \mathbb{I}_{n/2}} \sum_{R'} T(A', R') \cdot \Pr(R')$$

Expected running time of *expectedDemo*

$$\sum_{R} T(A,R)\cdot \Pr(R) =$$

$$\frac{1}{2}\sum_{R'}\left(1+T(A[0\,\ldots\,n/2-1],R')\right)\cdot \Pr(R') + \frac{1}{2}\sum_{R'}\left(1+T(A[0\ldots n-3],R')\right)\cdot \Pr(R')$$

$$= \frac{1}{2}\sum_{R'} 1\cdot \Pr(R') + \frac{1}{2}\sum_{R'} T\left(A\left[0\ldots\frac{n}{2}-1\right],R'\right)\cdot \Pr(R')$$

$$+ \frac{1}{2}\sum_{R'} 1\cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0\ldots n-3],R')\cdot \Pr(R')$$

$$= 1 + \frac{1}{2}\sum_{R'} T\left(A\left[0\ldots\frac{n}{2}-1\right],R'\right)\cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0\ldots n-3],R')\cdot \Pr(R')$$

$$\leq 1 + \max_{A'\in\mathbb{I}_{n/2}} \sum_{R'} T(A',R')\cdot \Pr(R') + \frac{1}{2}\boxed{\sum_{R'} T(A[0\ldots n-3],R')\cdot \Pr(R')}$$

$$\sum_{R'} T(A[0\ldots n-3],R')\cdot \Pr(R') \leq \max_{A'\in\mathbb{I}_{n-2}} \sum_{R'} T(A',R')\cdot \Pr(R')$$

Expected running time of *expectedDemo*

$$\sum_{R} T(A,R) \cdot \Pr(R) =$$

$$\frac{1}{2}\sum_{R'}\left(1 + T(A[0 \ldots n/2-1], R')\right) \cdot \Pr(R') + \frac{1}{2}\sum_{R'}\left(1 + T(A[0 \ldots n-3], R')\right) \cdot \Pr(R')$$

$$= \frac{1}{2}\sum_{R'} 1 \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T\left(A\left[0 \ldots \frac{n}{2}-1\right], R'\right) \cdot \Pr(R')$$

$$+ \frac{1}{2}\sum_{R'} 1 \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0 \ldots n-3], R') \cdot \Pr(R')$$

$$= 1 + \frac{1}{2}\sum_{R'} T\left(A\left[0 \ldots \frac{n}{2}-1\right], R'\right) \cdot \Pr(R') + \frac{1}{2}\sum_{R'} T(A[0 \ldots n-3], R') \cdot \Pr(R')$$

$$\leq 1 + \frac{1}{2}\underbrace{\max_{A' \in \mathbb{I}_{n/2}} \sum_{R'} T(A', R') \cdot \Pr(R')}_{T^{exp}(n/2)} + \frac{1}{2}\underbrace{\max_{A' \in \mathbb{I}_{n-2}} \sum_{R'} T(A', R') \cdot \Pr(R')}_{T^{exp}(n-2)}$$

Expected running time of $expectedDemo$

- For any $A \in \mathbb{I}_n$, it holds

$$\sum_R T(A,R) \cdot \Pr(R) \leq 1 + \frac{1}{2} T^{exp}(n/2) \; + T^{exp}(n-2)$$

- Therefore

$$T^{exp}(n) = \max_{A \in \mathbb{I}_n} \sum_R T(A,R) \cdot \Pr(R) \leq 1 + \frac{1}{2} T^{exp}(n/2) \; + T^{exp}(n-2)$$

- Same recurrence as for $averCaseDemo$
    - but it was much easier to derive this relation
    - usually expected runtime is easier to derive than the average case runtime
- Therefore, expected running time is $O(\log(n))$

Outline

- Sorting, average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - 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 $\le v$
  - pivot-value v is in $A[i]$
  - all items in $A\ [i+1, \ldots, n-1]$ are $\ge v$
  - 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-pivot(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

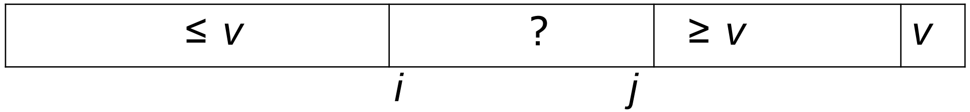
almost done, just swap with pivot v

						j = 6	**i = 7**		
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. $0 \leq p < n$

swap$(A[n-1],\ A[p])$
$i \leftarrow -1, \quad j \leftarrow n-1, \quad v \leftarrow A[n-1]$

loop

do $i \leftarrow i + 1$ **while** $A[i] < v$

do $j \leftarrow j - 1$ **while** $j \geq i$ **and** $A[j] > v$

if $i \geq j$ **then break**

else *swap*$(A[i],\ A[j])$

end loop

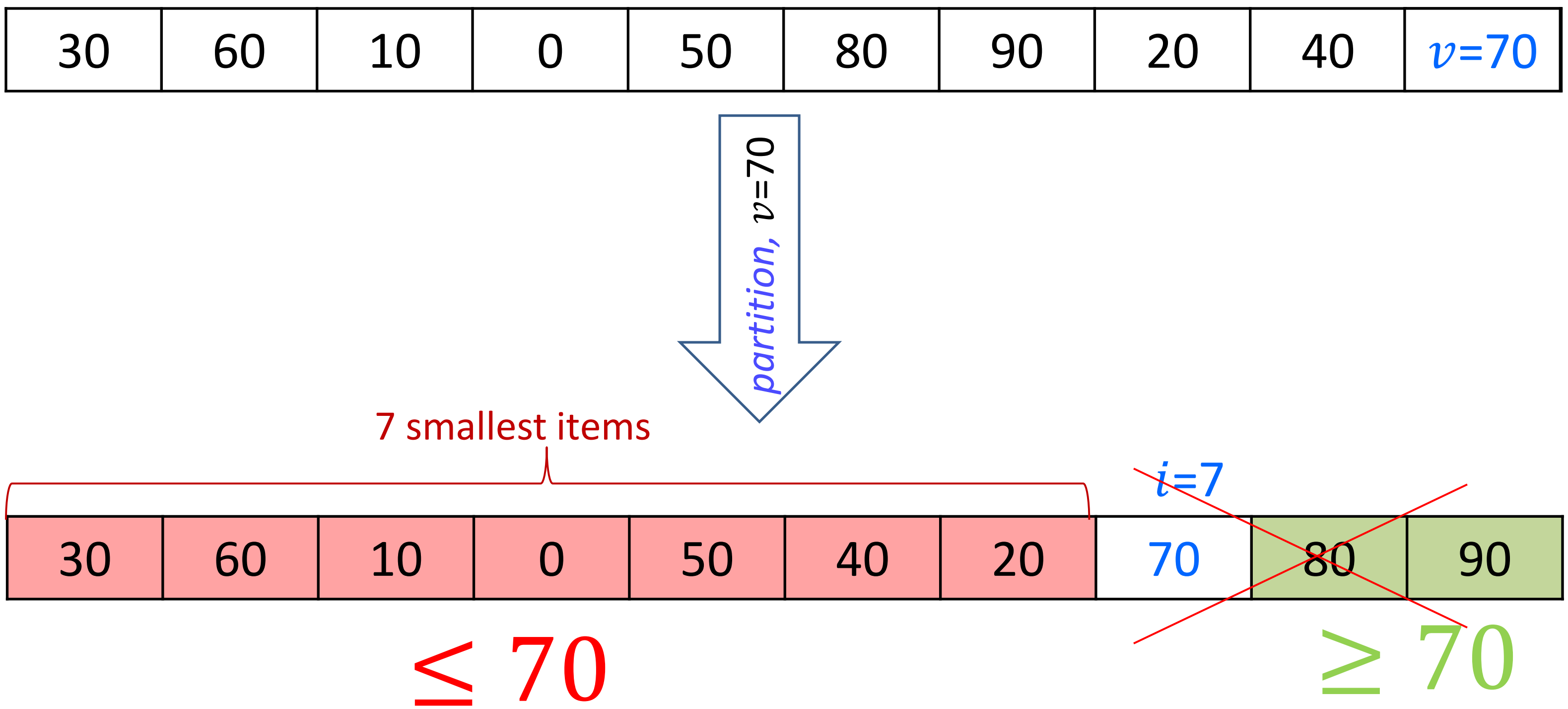
swap$(A[n-1],\ A[i])$

return i

- Running time is $\Theta(n)$

Quick Select Algorithm

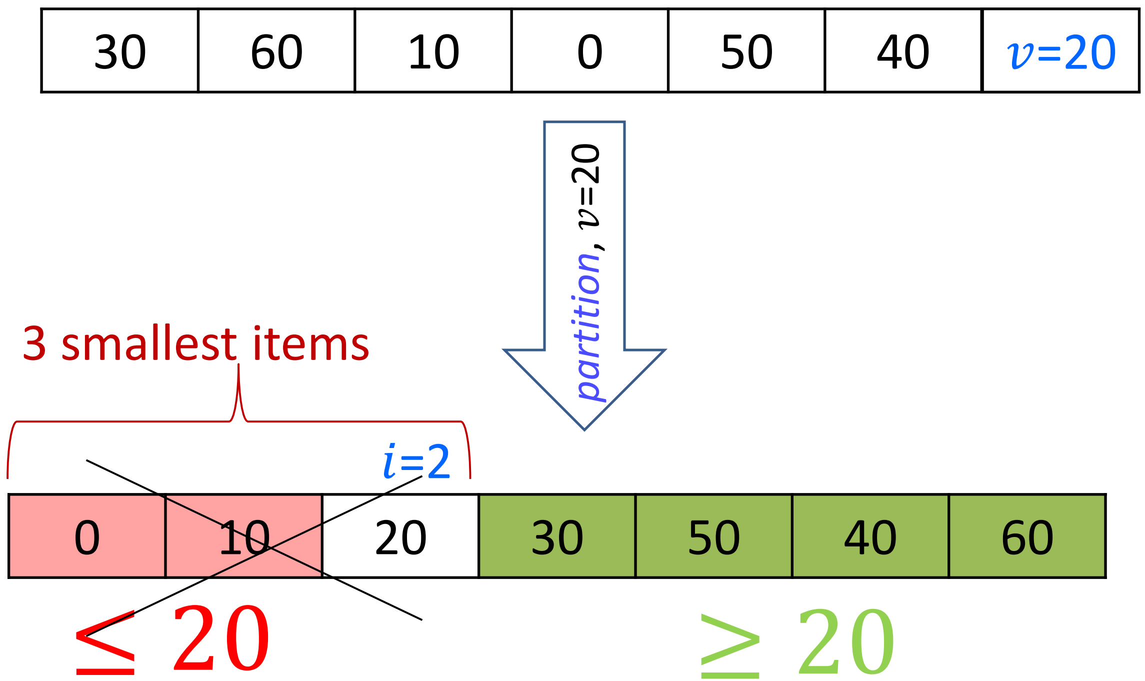
- Find item that would be in $A[k]$ if A was sorted
- Similar to quick-sort, but recurse only on one side ("quick-sort with pruning")
- Example: select($k = 4$)
  - [the correct answer is 40 in this case]



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

Quick Select Algorithm

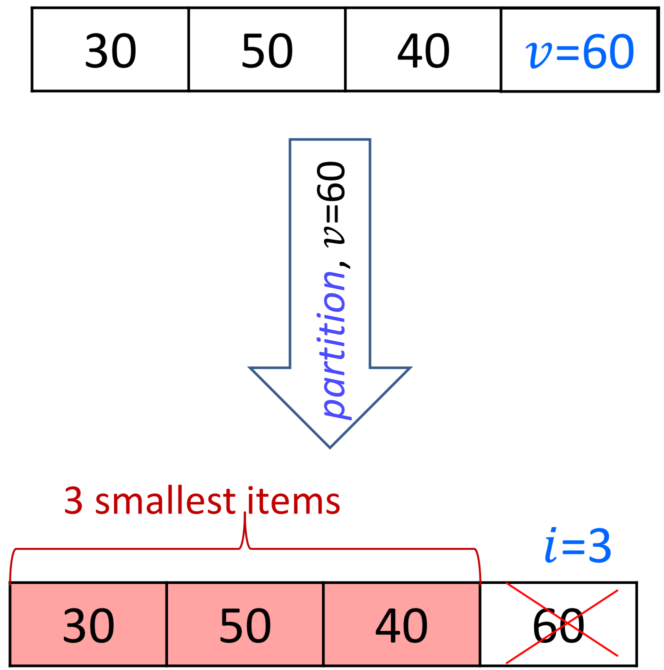
- Example continued: select($k = 4$)



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

Quick Select Algorithm

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

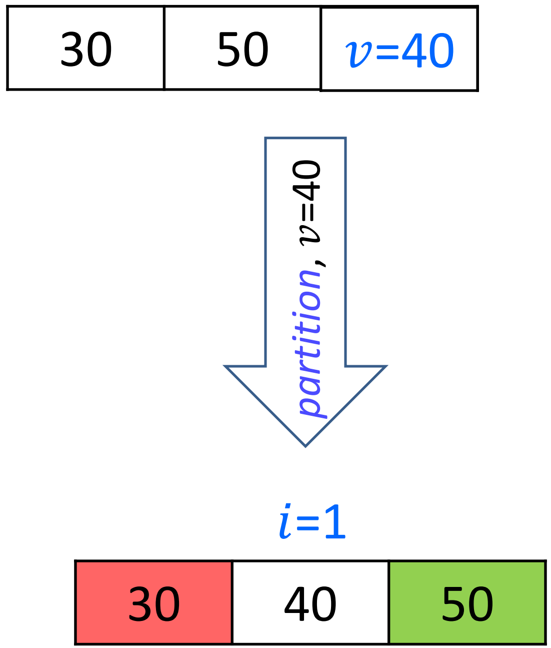


≤ 60

- $i > k$, 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

QuickSelect(A, k)

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

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

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

if $i = k$ **then**

  return $A[i]$

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

  return *QuickSelect*$(A[0, 1, \dots, i-1],\ k)$

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

  return *QuickSelect*$(A[i+1, \dots, n-1],\ k-(i+1))$

- **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

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

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

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

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

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

- Use again sorting permutations $T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$

- Let $T(n)$ be the number of comparisons (proportional to runtime)
- Assume that sorting permutation π gives pivot index i
- Let B be the new array (after partition)

	i	
$B[0 \dots i-1]$	v	$B[i+1 \dots n-1]$

$$T(\pi) \le n + \max\{T(\underbrace{B[0 \dots i-1]}_{\text{size } i}), T(\underbrace{B[i+1 \dots n-1]}_{\text{size } n-i-1})\}$$

- **Option 1**:
  - first prove that this implies (very complicated)

$$\sum_{\substack{\pi \in \Pi_n: \\ pivot-idx\ i}} T(\pi) \le \sum_{\substack{\pi \in \Pi_n: \\ pivot-idx\ i}} (n + \max\{T^{avg}(i), T^{avg}(n-i-1)\})$$

  - then derive and solve average case recursive relationship (not too difficult)

Average-Case Analysis of *quick-select*

- **Option 2**: Prove average case run time via randomization
  - simpler than option 1
  - randomization is useful in practice
- Need to discuss
  1. how to randomize QuickSelect (RandomizedQuickSelect)?
  2. what is the expected run-time of RandomizedQuickSelect?
  3. what does expected run time of RandomizedQuickSelect imply for average run-time of QuickSelect?

Randomized QuickSelect: Shuffle

- **First idea**: first randomly permute input using *shuffle* and then run selection algorithm

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

- $random(n)$ returns an integer uniformly sampled from $\{0, 1, 2, \ldots, n-1\}$
- Works well but we can do randomization directly within the sorting algorithm

Randomized QuickSelect: Random Pivot

- **Second idea**: change pivot selection

```
RandomizedQuickSelect(A, k)
        ...
        p ← random(A.size)
        ...
```

- Let $T(A, k, R)$ be the runtime on array A of size n, selecting kth element, using a sequence of random numbers R
  - assume all array elements are distinct, and $n \geq 2$
- Let $R = \langle x, R' \rangle$ and suppose x corresponds to pivot-index i
  - we recurse in an array of size i or $n - i - 1$ (or algorithms stops)
  - call the new array (after partition) B

$$T(A, k, \langle x, R' \rangle) \leq cn + \begin{cases} T(B[0 \dots i-1], k, R') & \text{if } i > k \\ T(B[i+1 \dots n-1], k-i-1, R') & \text{if } i < k \\ 0 & \text{otherwise} \end{cases}$$

- $T(A, k, \langle x, R' \rangle) \leq cn + [T(B[0 \dots i-1], k, R')$ or $T(B[i+1 \dots n-1], k-i-1, R')]$

Analysis of Randomized QuickSelect

- $T(A, k, \langle x, R' \rangle) \leq cn + [T(B[0 \dots i-1], k, R')$ or $T(B[i+1 \dots n-1], k-i-1, R')]$
  - i is pivot index corresponding to x

$$\sum_{R} T(A,k,R)\Pr(R) = \sum_{\langle x, R' \rangle} T(A,k,\langle x, R' \rangle)\Pr(\langle x, R' \rangle) = \sum_{x=0}^{n-1} \sum_{R'} T(A,k,\langle x, R' \rangle) \Pr(x) \Pr(R')$$

$$= \frac{1}{n}\left(\sum_{R'} T(A,k,\langle 0, R' \rangle)P(R') + \sum_{R'} T(A,k,\langle 1, R' \rangle)P(R') + \dots + \sum_{R'} T(A,k,\langle n-1, R' \rangle)P(R') \right)$$

each x corresponds to some pivot-index $i \in \{0, \dots, n-1\}$, and each element of $\{0, \dots, n-1\}$ appears exactly one time in this sum as pivot-index for some x

- Rewriting the sum in terms of pivot indexes, and using the inequality from top of the slide

$$\sum_{R} T(A,k,R)\Pr(R) \leq \frac{1}{n} \sum_{i=0}^{n-1} \sum_{R'} \left(cn + [T(B[0 \dots i-1], k, R') \text{ or } T(B[i+1 \dots n-1], k-i-1, R')] \right) \Pr(R')$$

$$= \underbrace{\frac{1}{n} \sum_{i=0}^{n-1} \sum_{R'} Pr(R')cn}_{= cn} + \frac{1}{n} \sum_{i=0}^{n-1} \sum_{R'} [T(B[0 \dots i-1], k, R') \text{ or } T(B[i+1 \dots n-1], k-i-1, R')] \Pr(R')$$

Analysis of Randomized QuickSelect

$$\sum_{R} T(A,k,R)\Pr(R) \leq cn + \frac{1}{n}\sum_{i=0}^{n-1}\sum_{R'} [T(B[0\ldots i-1],k,R') \text{ or } T(B[i+1\ldots n-1],k-i-1,R')]\Pr(R')$$

$$= cn + \frac{1}{n}\sum_{i=0}^{n-1}\left[\sum_{R'}[T(B[0\ldots i-1],k,R')\Pr(R') \text{ or } \sum_{R'} T(B[i+1\ldots n-1],k-i-1,R')]\, i-1,R')\Pr(R')\right]$$

$$\leq cn + \frac{1}{n}\sum_{i=0}^{n-1}\max\left\{\sum_{R'} T(B[0\ldots i-1],k,R')\Pr(R'), \sum_{R'} T(B[i+1\ldots n-1],k-i-1,R')]\Pr(R')\right\}$$

$$\leq cn + \frac{1}{n}\sum_{i=0}^{n-1}\max\left\{\max_{m}\max_{C\in\mathbb{I}_i}\sum_{R'} T(C,m,R')\Pr(R'), \max_{m}\max_{C\in\mathbb{I}_{n-i-1}}\sum_{R'} T(C,m,R')\Pr(R')\right\}$$

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

- Since this bound is true for any A and k, it is be true for the worst case A and k

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

Analysis of Randomized QuickSelect

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

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

Proof:

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

induction hypothesis applies to each one of these

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

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

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

Analysis of Randomized QuickSelect

exactly what we need for the proof

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

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

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

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

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

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

Analysis of Randomized QuickSelect

- Thus expected runtime of *RandomizedQuickSelect* is $\Theta(n)$
- This is generally the fastest implementation of a selection algorithm
- There is a selection algorithm that has worst-case running time $O(n)$
  - CS341
  - but it uses double recursion and is slower in practice

Expected vs. Average-case runtime

- Assume we have an algorithm $\mathbb{A}$ that solves Selection or Sorting
- Create a randomized algorithm $\mathbb{B}$ that solves the same problem as $\mathbb{A}$ as follows
    - let I be the given instance (an array)
    - randomly (and uniformly) permute I to get I'
        - can do this with *shuffle*
            - for *QuickSelect*, choosing pivot randomly is equivalent to shuffling
    - call algorithm $\mathbb{A}$ on input I'
- Claim: $T_{\mathbb{B}}^{exp}(n) = T_{\mathbb{A}}^{avg}(n)$
- Proof:
    - let I be an instance, and π be its sorting permutation
        - $\pi(I) = I_{sorted}$
    - let σ be the sorting permutation applied during shuffling to I
        - $I' = \sigma(I)$
        - $\sigma^{-1}(I') = I$
        - $\pi \circ \sigma^{-1}(I') = \pi(I) \ = I_{sorted}$
        - I' has sorting permutation $\pi \circ \sigma^{-1}$

Expected vs. Average-case runtime

- Assume we have an algorithm $\mathbb{A}$ that solves Selection or Sorting
- Create a randomized algorithm $\mathbb{B}$ that solves the same problem as $\mathbb{A}$ as follows
    - let I be the given instance (an array)
    - randomly (and uniformly) permute I to get I'
    - call algorithm $\mathbb{A}$ on input I'
- Claim: $T_{\mathbb{B}}^{exp}(n) = T_{\mathbb{A}}^{avg}(n)$
- Proof:
    - let I be an instance, and π be its sorting permutation
    - let σ be the sorting permutation applied during shuffling to I,
        - $I' = \sigma(I)$
        - I' has sorting permutation $\pi \circ \sigma^{-1}$

$$T_{\mathbb{B}}^{exp}(n) = \max_{I \in \mathbb{I}_n} \sum_R T_{\mathbb{B}}(I, R)\Pr(R) = \max_{I \in \mathbb{I}_n} \sum_R T_{\mathbb{A}}(I')\Pr(R) = \max_{\pi \in \Pi_n} \frac{1}{n!} \sum_{\sigma \in \Pi_n} T_{\mathbb{A}}(\pi \circ \sigma^{-1})$$

σ goes over all permutations, so $\pi \circ \sigma^{-1}$ also goes over all permutations

Example: $\pi = (2,0,1)$

$\sigma \in \Pi_n =$ $(0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1), (2,1,0)$

apply σ^{-1} $(0,1,2), (0,2,1), (1,0,2), (2,0,1), (1,2,0), (2,1,0)$

apply π $(2,0,1), (1,0,2), (2,1,0), (1,2,0), (0,1,2), (0,2,1)$

Expected vs. Average-case runtime

- Assume we have an algorithm $\mathbb{A}$ that solves Selection or Sorting
- Create a randomized algorithm $\mathbb{B}$ that solves the same problem as $\mathbb{A}$ as follows
  - let I be the given instance (an array)
  - randomly (and uniformly) permute I to get I'
  - call algorithm $\mathbb{A}$ on input I'
- Claim: $T_{\mathbb{B}}^{exp}(n) = T_{\mathbb{A}}^{avg}(n)$
- Proof:
  - let I be an instance, and π be its sorting permutation
  - let σ be the sorting permutation applied during shuffling to I,
    - $I' = \sigma(I)$
    - I' has sorting permutation $\pi \circ \sigma^{-1}$

$$T_{\mathbb{B}}^{exp}(n) = \max_{I \in \mathbb{I}_n} \sum_{R} T_{\mathbb{B}}(I, R)\Pr(R) = \max_{I \in \mathbb{I}_n} \sum_{R} T_{\mathbb{A}}(I')\Pr(R) = \max_{\pi \in \Pi_n} \frac{1}{n!} \sum_{\sigma \in \Pi_n} T_{\mathbb{A}}(\pi \circ \sigma^{-1})$$

- Change summation variable to τ

$$= \max_{\pi} \frac{1}{n!} \sum_{\tau \in \Pi_n} T_{\mathbb{A}}(\tau) = \max_{\pi} T_{\mathbb{A}}^{avg}(n) = T_{\mathbb{A}}^{avg}(n)$$

Expected vs. Average-case runtime

- Assume we have an algorithm $\mathbb{A}$ that solves Selection or Sorting
- Create a randomized algorithm $\mathbb{B}$ that solves the same problem as $\mathbb{A}$ as follows
    - let I be the given instance (an array)
    - randomly (and uniformly) permute I to get I'
        - can do this with *shuffle*
            - for *QuickSelect*, choosing pivot randomly is equivalent to shuffling
    - call algorithm $\mathbb{A}$ on input I'
- Claim: $T_{\mathbb{B}}^{exp}(n) = T_{\mathbb{A}}^{avg}(n)$

- Since *RandomizedQuickSelect* has expected running time $O(n)$, then the average case of *QuickSelect* is also $O(n)$

Outline

- Sorting, average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - 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

> *QuickSort*(A)
>
> Input: array *A* of size *n*
>
> **if** $n \leq 1$ **then return**
>
> $p \leftarrow$ *choose-pivot*(A)
>
> $i \leftarrow$ *partition* $(A\,,p)$
>
> *QuickSort*$(A[0, 1, \ldots, i-1])$
>
> *QuickSort*$(A[i+1, \ldots, n-1])$

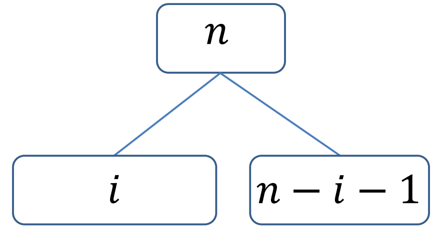
- Let $T(n)$ to be the runtime on size n array
- If we know pivot-index i, then $T(n) = cn + T(i) + T(n-i-1)$
- Worst case $T(n) = T(n-1) + cn$
  - recurrence solved in the same way as quick–select1, $\Theta(n^2)$
- Best case $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn$
  - solved in the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

Randomized QuickSort: Random Pivot

```
RandomizedQuickSort(A)
        ...
        p ← random(A.size)
        ...
```

- Analysis is similar to that of *RandomizedQuickSelect*
  - but recurse both in array of size i and array of size $n-i-1$
- Can derive $T^{exp}(n) \leq n + \frac{2}{n}\sum_{i=0}^{n-1} T^{exp}(i)$
  - then show that this is $O(n \log n)$
- However there is an easier analysis

Expected Recursion depth of Randomized QuickSort



- Analyze expected height of recursion tree
- Define $H(A,R)$ be the height for instance A and random numbers R
- $H^{exp}(n) = \max_A \sum_R \Pr(R)\, H(A,R)$
- Let A have size larger than 1, $R = \langle x, R' \rangle$ and suppose x leads to pivot-index i

$$H(A,R) \le 1 + max\{H(\underbrace{\text{instance of size } i}_{A_i}, R'), H(\underbrace{\text{instance of size } n-i-1}_{A_{n-i-1}}, R')\}$$

- Summing up over all R, and switching from x to corresponding pivot-index i

$$\sum_R \Pr(R)\, H(A,R) = \sum_{x=0}^{n-1} \sum_{R\prime} \Pr(x) \Pr(R')\, H(A, \langle x, R' \rangle)$$

$$\le \frac{1}{n} \sum_{i=0}^{n-1} \sum_{R\prime} \Pr(R')(1 + max\{H(A_i, R'), H(A_{n-i-1}, R')\})$$

$$\le 1 + \frac{1}{n} \sum_{i=0}^{n-1} max \left\{ \sum_{R\prime} \Pr(R') H(A_i, R')\,, \sum_{R\prime} \Pr(R') H(A_{n-i-1}, R') \right\}$$

Expected Recursion depth of Randomized QuickSort

$$\sum_{R} \Pr(R)\, H(A,R) \leq 1 + \frac{1}{n}\sum_{i=0}^{n-1} max\left\{\sum_{R\prime} \Pr(R')H(A_i,R')\,,\sum_{R\prime} \Pr(R')H(A_{n-i-1},R')\right\}$$

$$\leq 1 + \frac{1}{n}\sum_{i=0}^{n-1} max\left\{\max_{C\in\mathbb{I}_i}\sum_{R\prime} \Pr(R')H(C,R')\,,\max_{C\in\mathbb{I}_{n-i-1}}\sum_{R\prime} \Pr(R')H(C,R')\right\}$$

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

- Since this holds for any instance A, it will hold for worst-case instance

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

Expected Recursion depth of Randomized QuickSort

$$H^{exp}(n) \le 1 + \frac{1}{n}\sum_{i=0}^{n-1} max\{H^{exp}(i), H^{exp}(n-i-1)\} \text{ for } n > 1$$

- Claim: $H^{exp}(n)$ is $O(\log n)$
- Proof: show by induction $H^{exp}(n) \le 2\log_{4/3} n$ (assume n is divisible by 4)
  - for $n = 1$, $H^{exp}(n) \le \log_{\frac{4}{3}} n = 0$, so the statement holds
  - let $n > 1$, and assume statement holds for all $m < n$
  - $H^{exp}(n) \le 1 + \frac{1}{n}\sum_{i=0}^{n-1} max\{H^{exp}(i), H^{exp}(n-i-1)\}$

induction hypothesis

$$\le 1 + \frac{1}{n}\sum_{i=0}^{n-1} max\{2\log_{4/3} i, 2\log_{4/3}(n-i-1)\}$$

$$\le 1 + \frac{2}{n}\sum_{i=0}^{n-1} max\{\log_{4/3} i, \log_{4/3}(n-i-1)\}$$

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

Expected Recursion depth of Randomized QuickSort

$H^{exp}(n) \leq$

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

$$\leq \log_{\frac{4}{3}}(n-1) \quad \leq \log_{\frac{4}{3}}(n-1) \quad \underbrace{\leq \log_{\frac{4}{3}}\left(\frac{3n}{4}\right) \quad \leq \log_{\frac{4}{3}}\left(\frac{3n}{4}\right) \quad \leq \log_{\frac{4}{3}}\left(\frac{3n}{4}\right) \quad \leq \log_{\frac{4}{3}}\left(\frac{3n}{4}\right)}_{\frac{n}{2}\text{ terms}} \quad \leq \log_{\frac{4}{3}}(n-1)$$

$$\leq 1+\frac{2}{n}\left(\frac{n}{2}\log_{\frac{4}{3}}(n-1)+\frac{n}{2}\log_{\frac{4}{3}}\left(\frac{3n}{4}\right)\right) = 1+\log_{\frac{4}{3}}(n-1)+\log_{\frac{4}{3}} n+\log_{4/3} 3/4$$

$$\leq 2\log_{4/3} n$$

- So expected height of the recursion tree is $O(\log n)$
- We do $\Theta(n)$work on each level of the recursion tree
- Expected runtime of *RandomizedQuickSelect* is $O(n \log n)$
- Average case runtime of *QuickSelect* is $O(n \log n)$

Improvement ideas for QuickSort

- The auxiliary space is Ω(recursion depth)
  - $\Theta(n)$ in the worst case, $\Theta(\log n)$ average 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

$< \boldsymbol{v}$	$= \boldsymbol{v}$	$> \boldsymbol{v}$

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

QuickSort with Tricks

QuickSortImproves(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-pivot-improved*$(A,\ l,\ r)$
$i \leftarrow$ *partition-improved* (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
  - although worst-case is $\Theta(n^2)$

Outline

- Sorting, average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - 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
quickSort RandomizedQuickSort	$\Theta(n \log n)$ $\Theta(n \log n)$	average-case 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 $[x_0,x_1,x_2]$

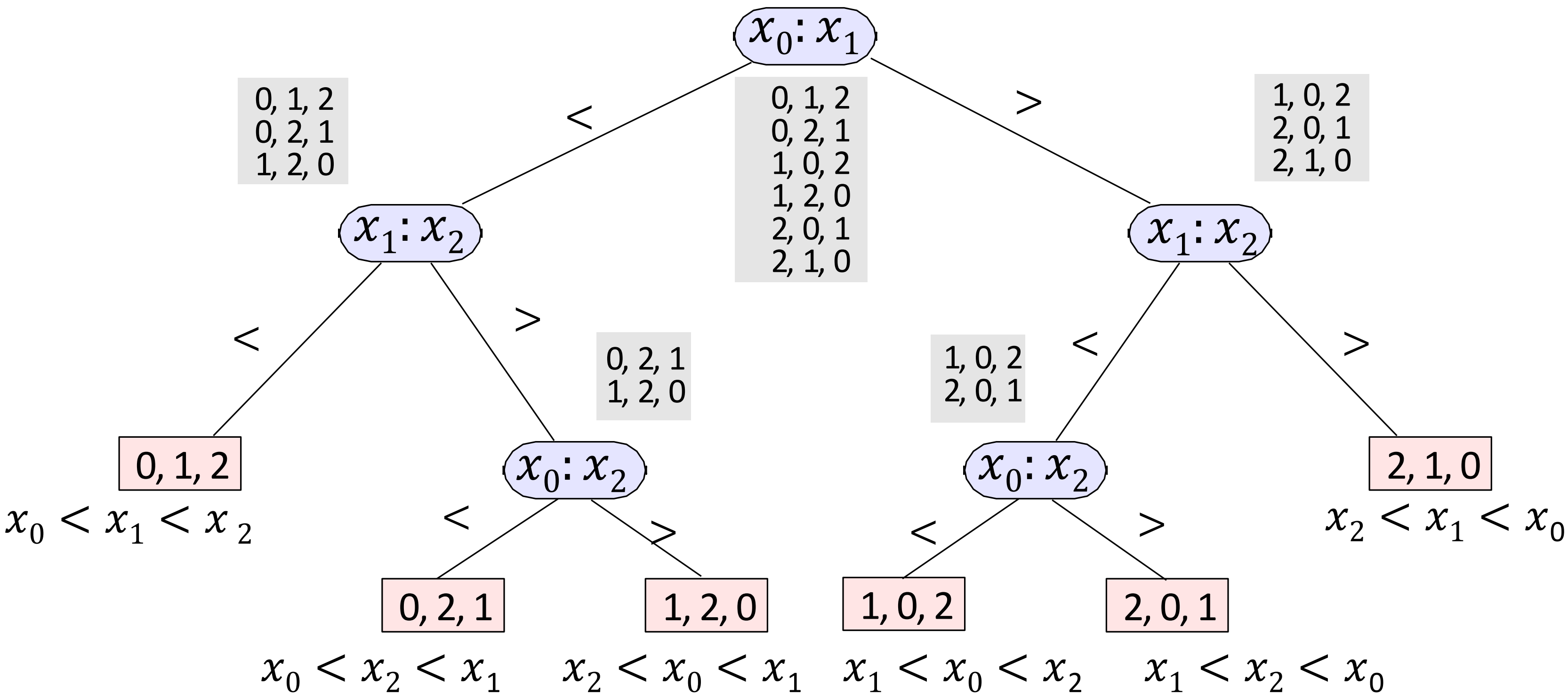
Set of all possible inputs

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

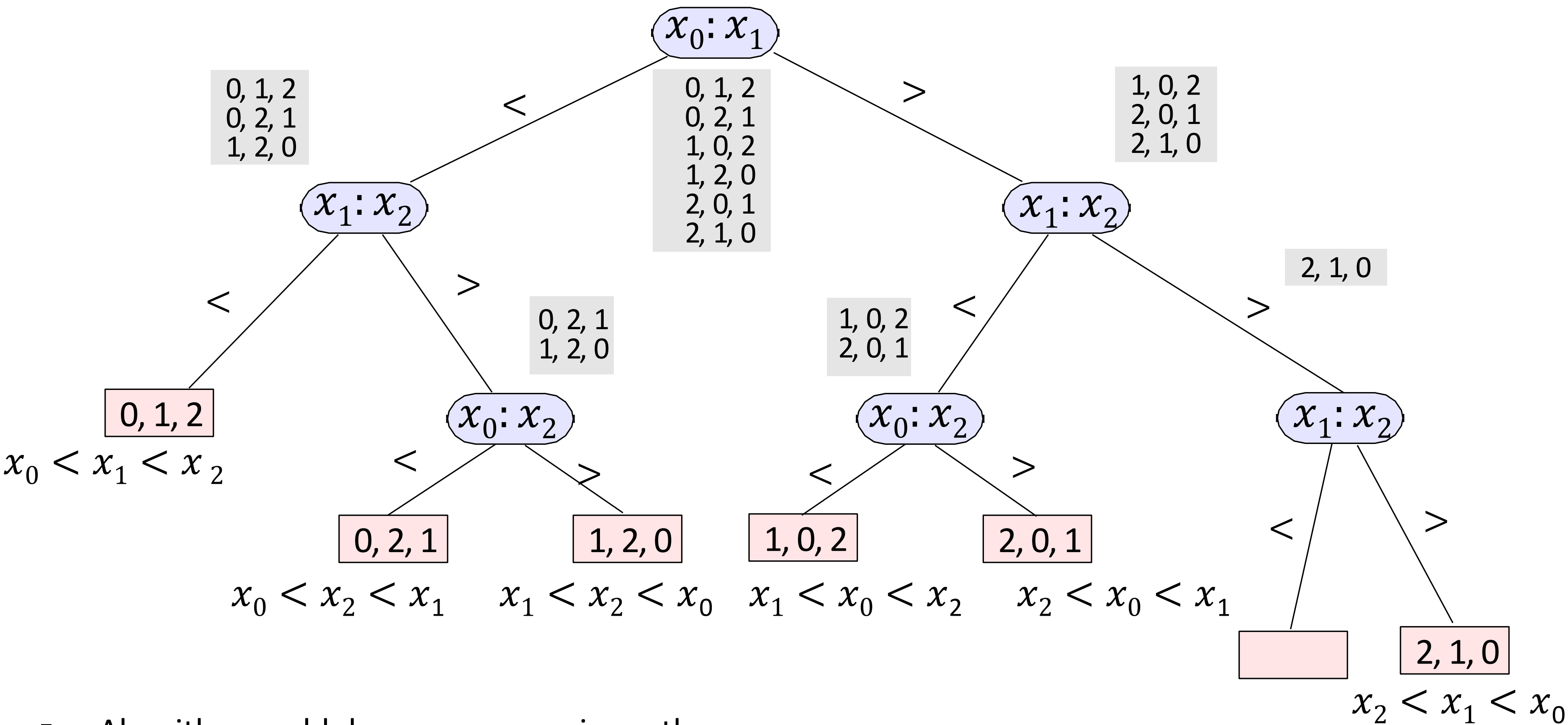
- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements



- 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

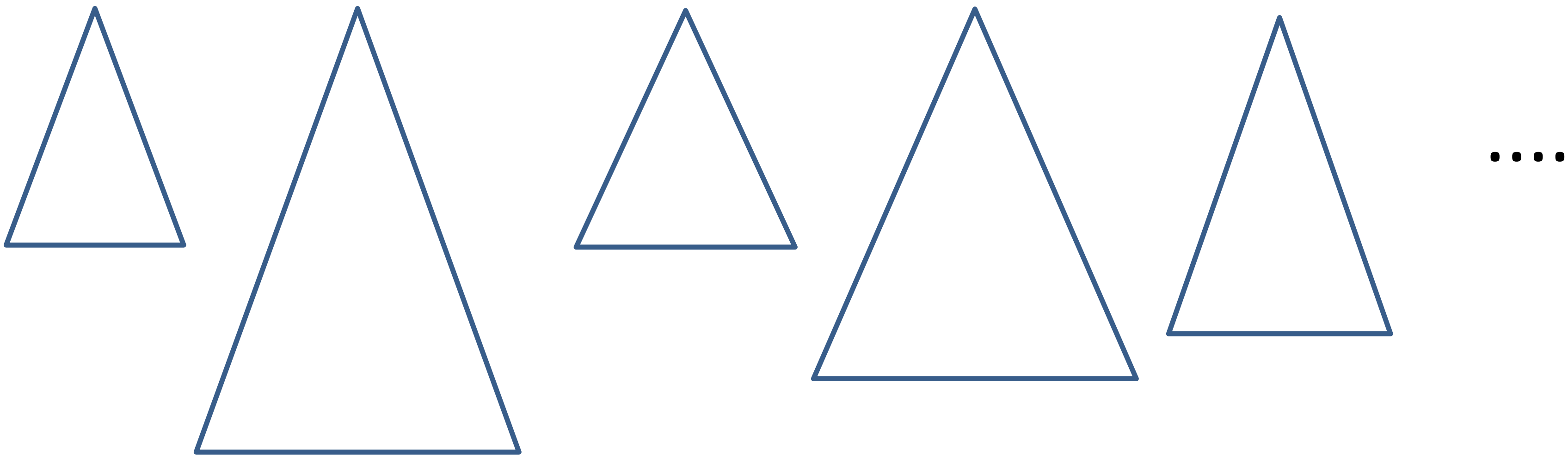
- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements



- 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

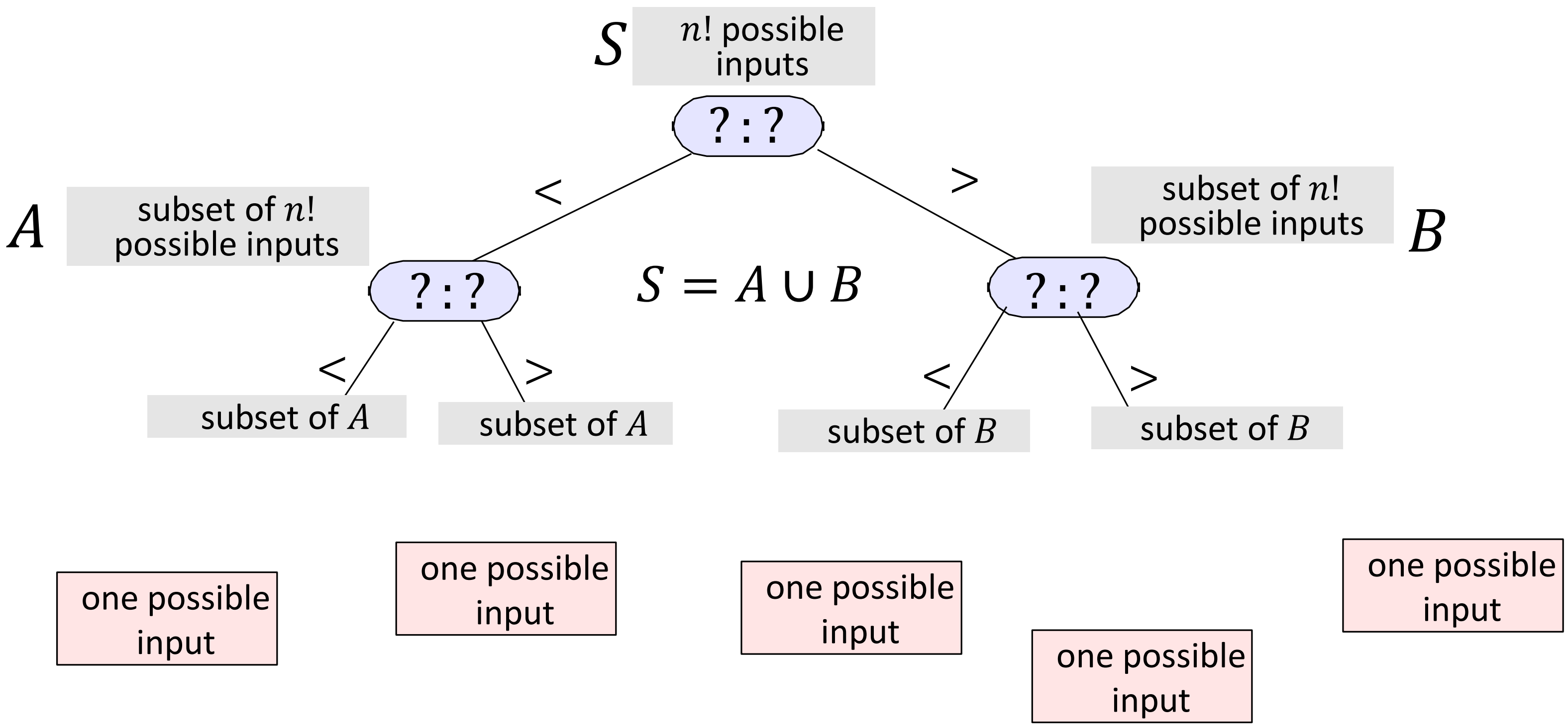
- General case: n non-repeating elements
- Many sorting algorithms, for each one we have its own decision tree
  - decision trees will have various heights



- Smallest height gives us the lower bound on the sorting problem
- Can we reason about the best (smallest) possible height any decision tree must have?

Decision Tree

- Can reason about decision tree for *any* comparison-based sorting algorithm with 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

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

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

Outline

- Sorting, average-case, and Randomization
  - Analyzing average-case run-time
  - Randomized Algorithms
  - QuickSelect
  - QuickSort
  - Lower Bound for Comparison-Based Sorting
  - Non-Comparison-Based Sorting

Non-Comparison-Based Sorting

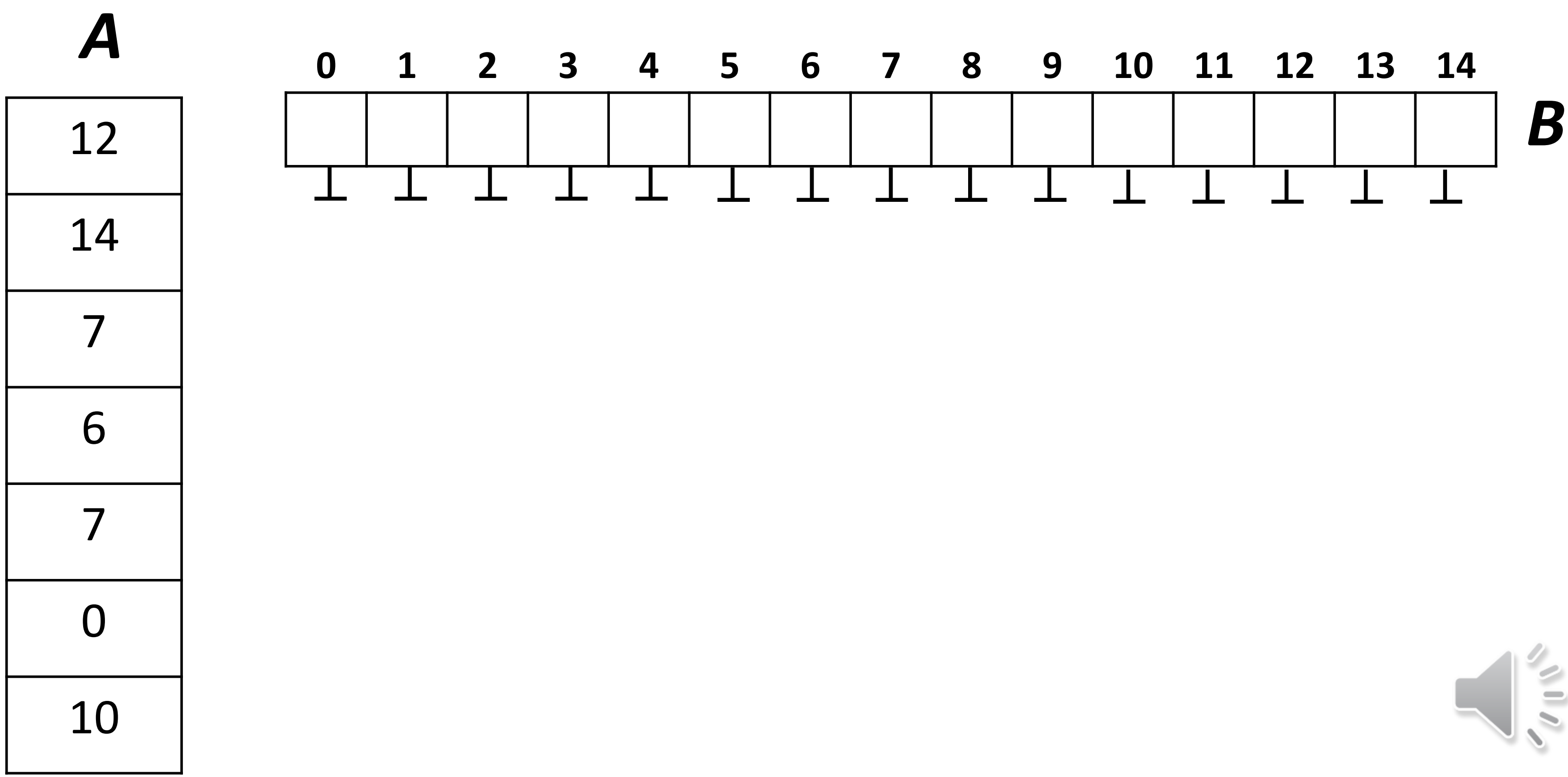
- Sort without comparing items to each other
- Non-comparison based sorting is less general than comparison based
- In particular, we need to make assumptions about items we sort
    - unlike in comparison based sorting, which sorts any data, as long as it can be compared
- Will assume we are sorting 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

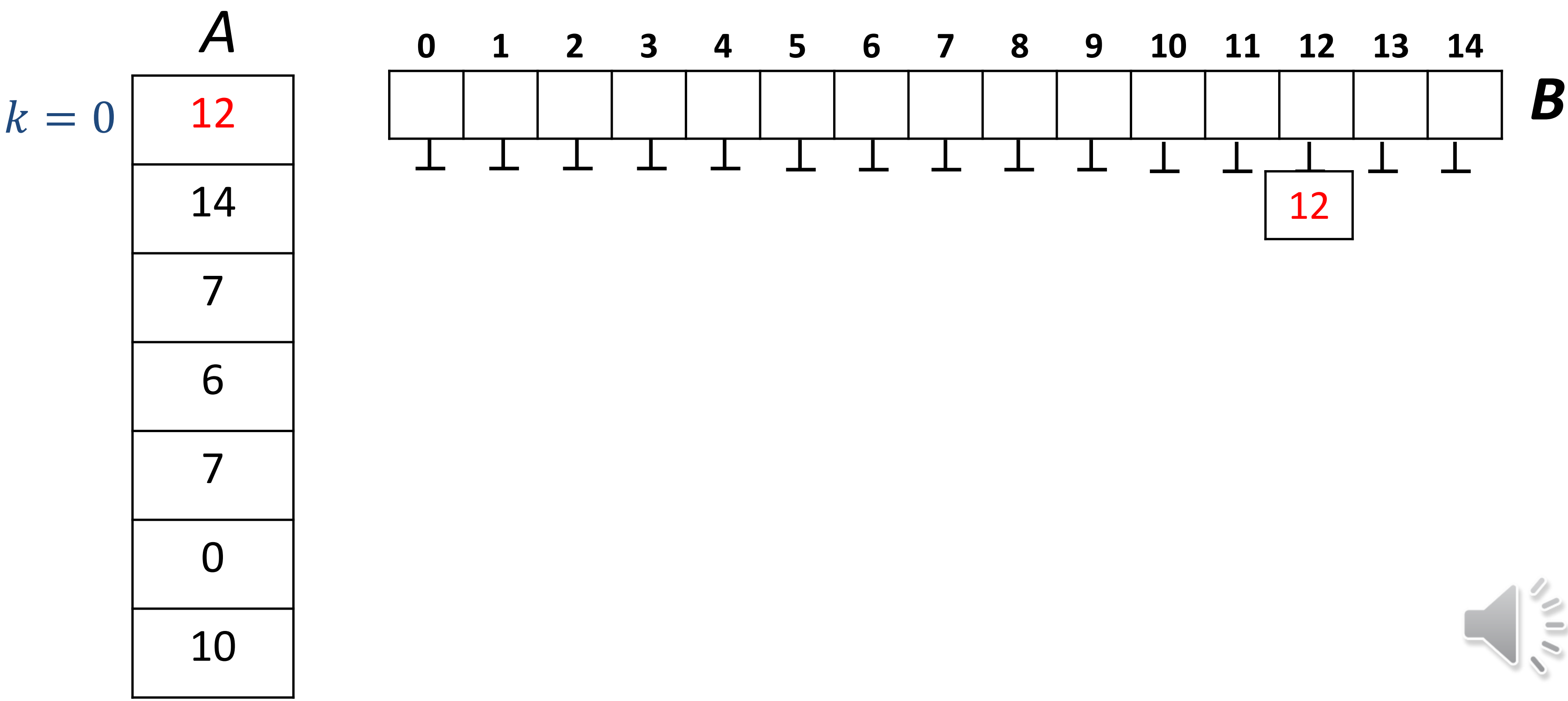
Bucket Sort

- 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 $L = 15$



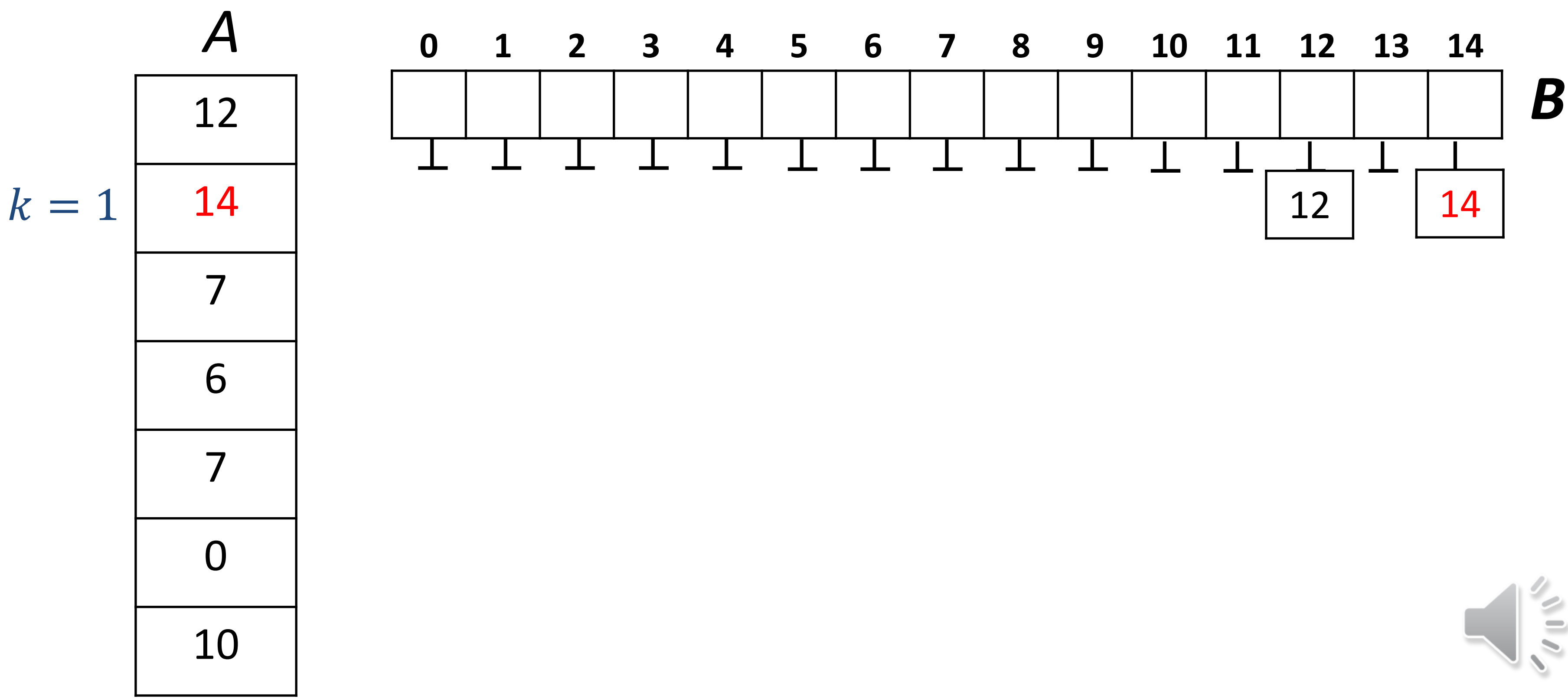
Bucket Sort

- 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 $L = 15$



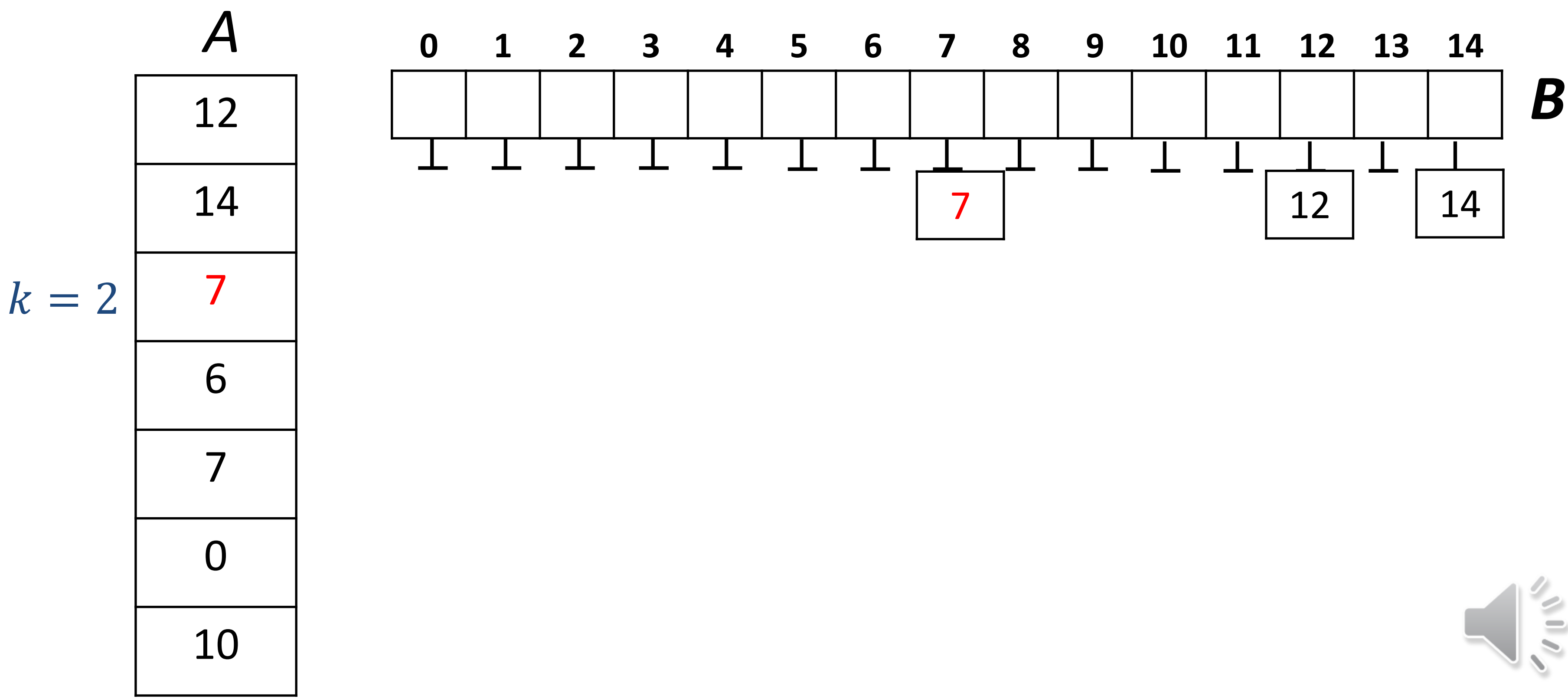
Bucket Sort

- 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 $L = 15$



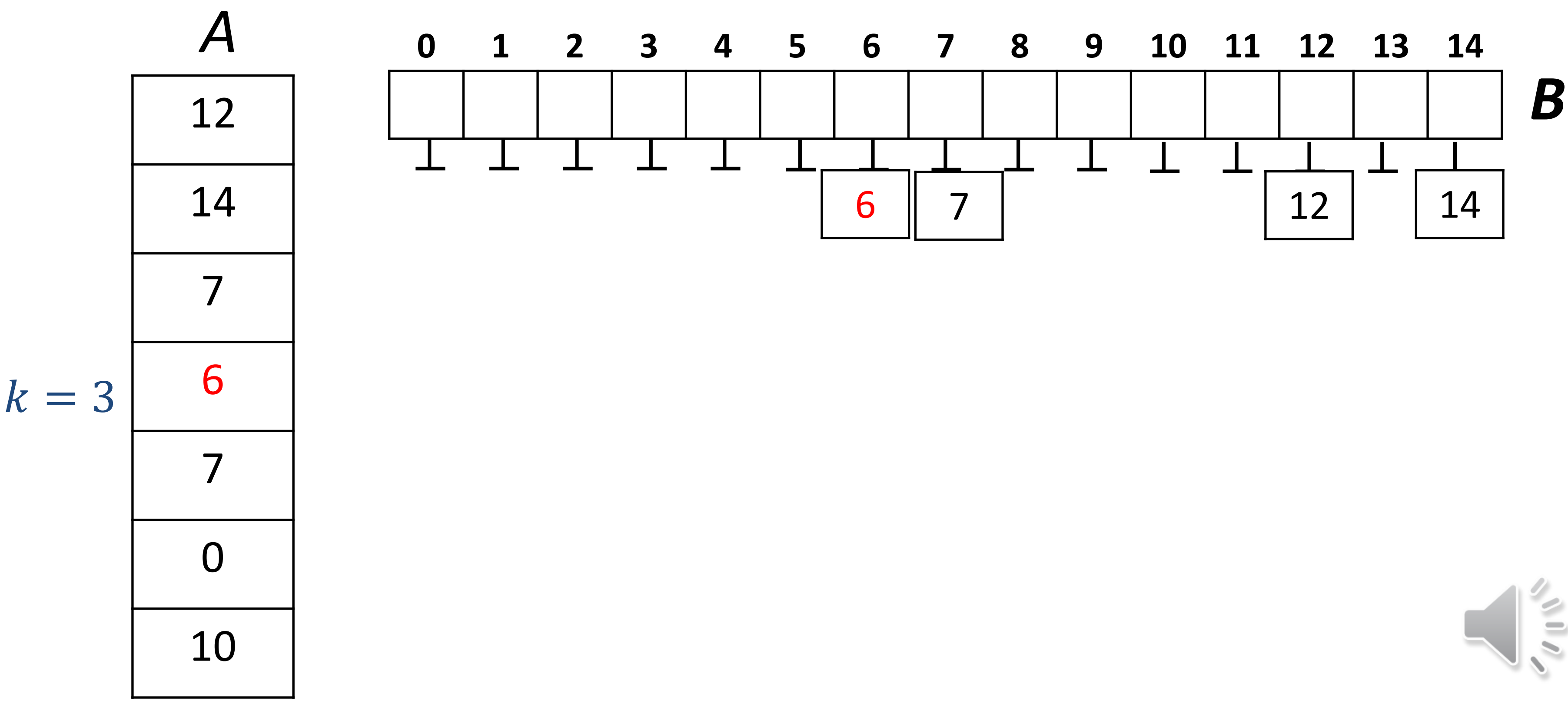
Bucket Sort

- 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 $L = 15$



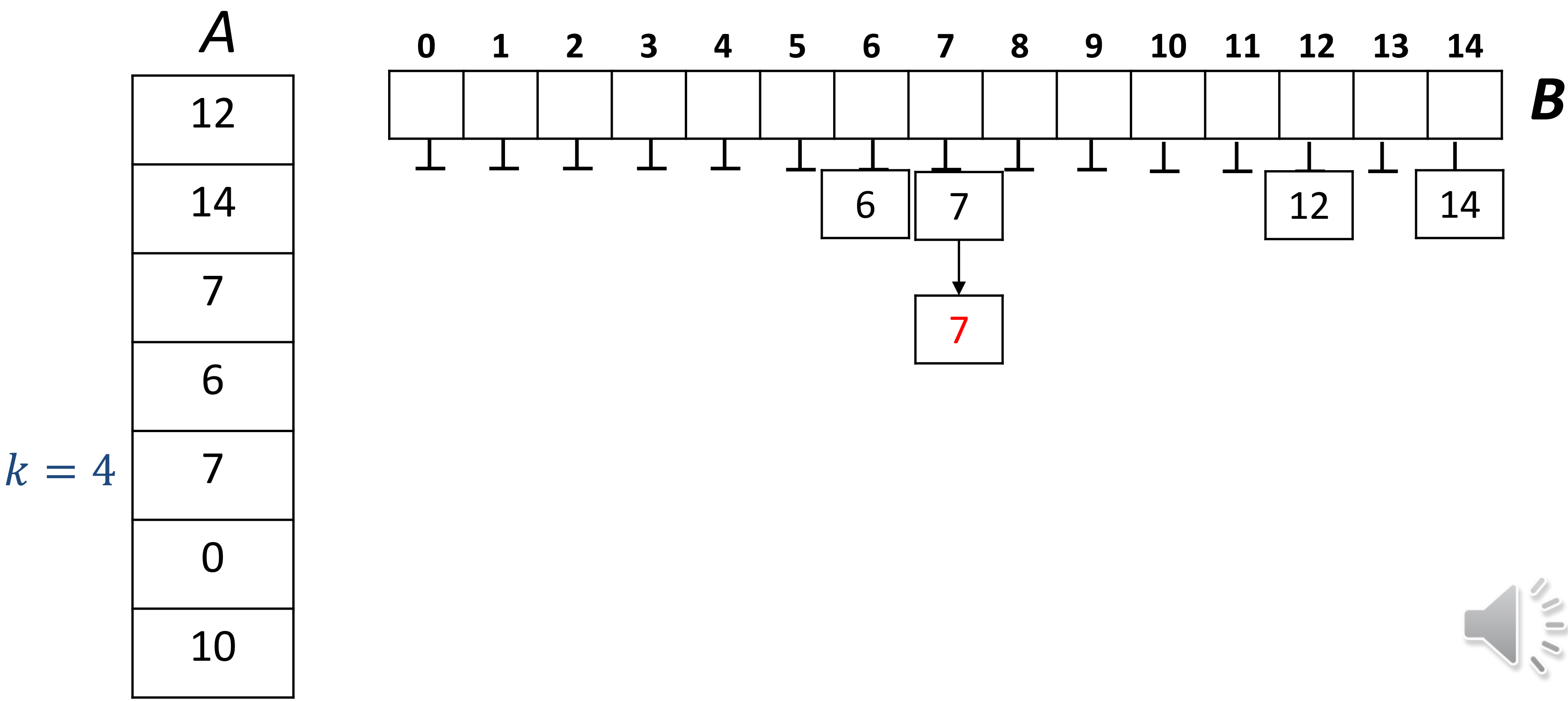
Bucket Sort

- 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 $L = 15$



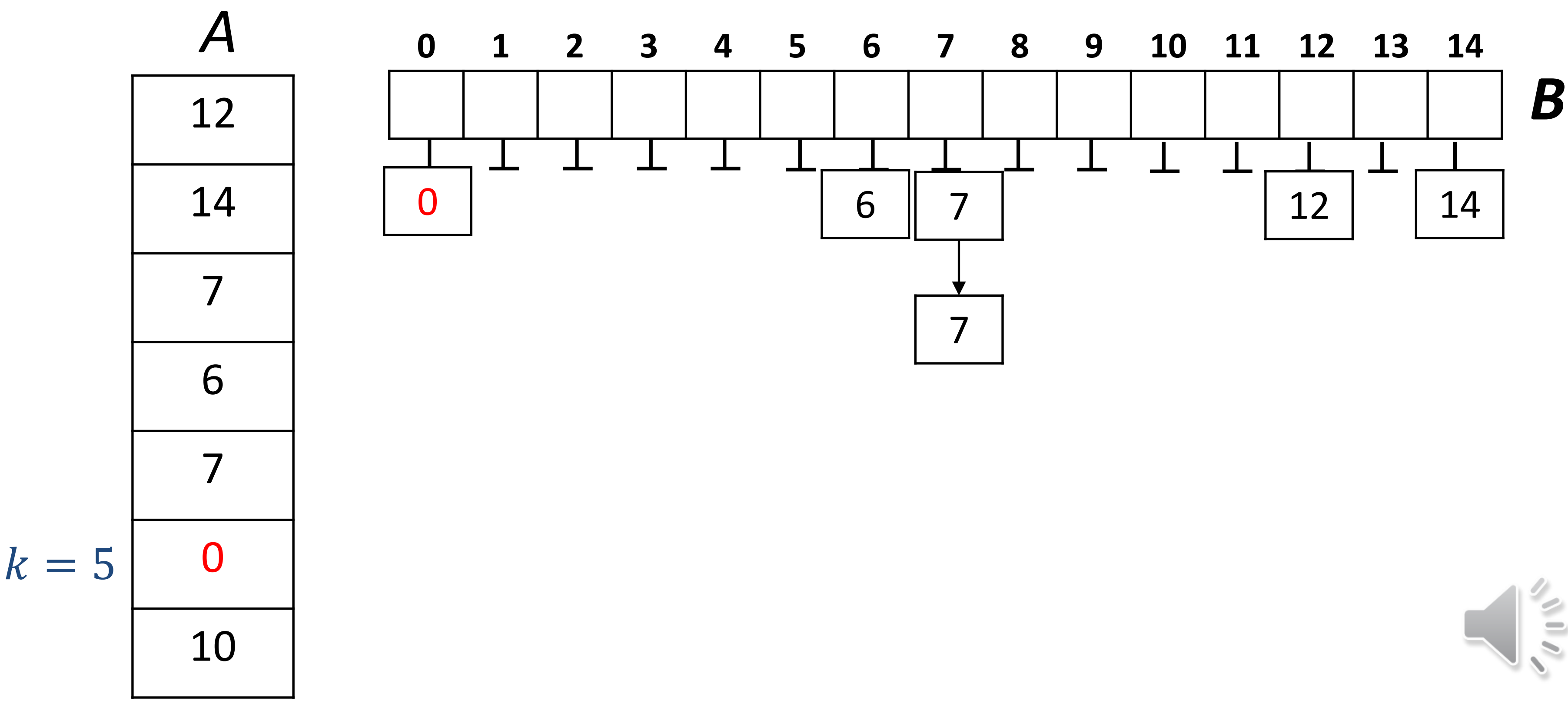
Bucket Sort

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



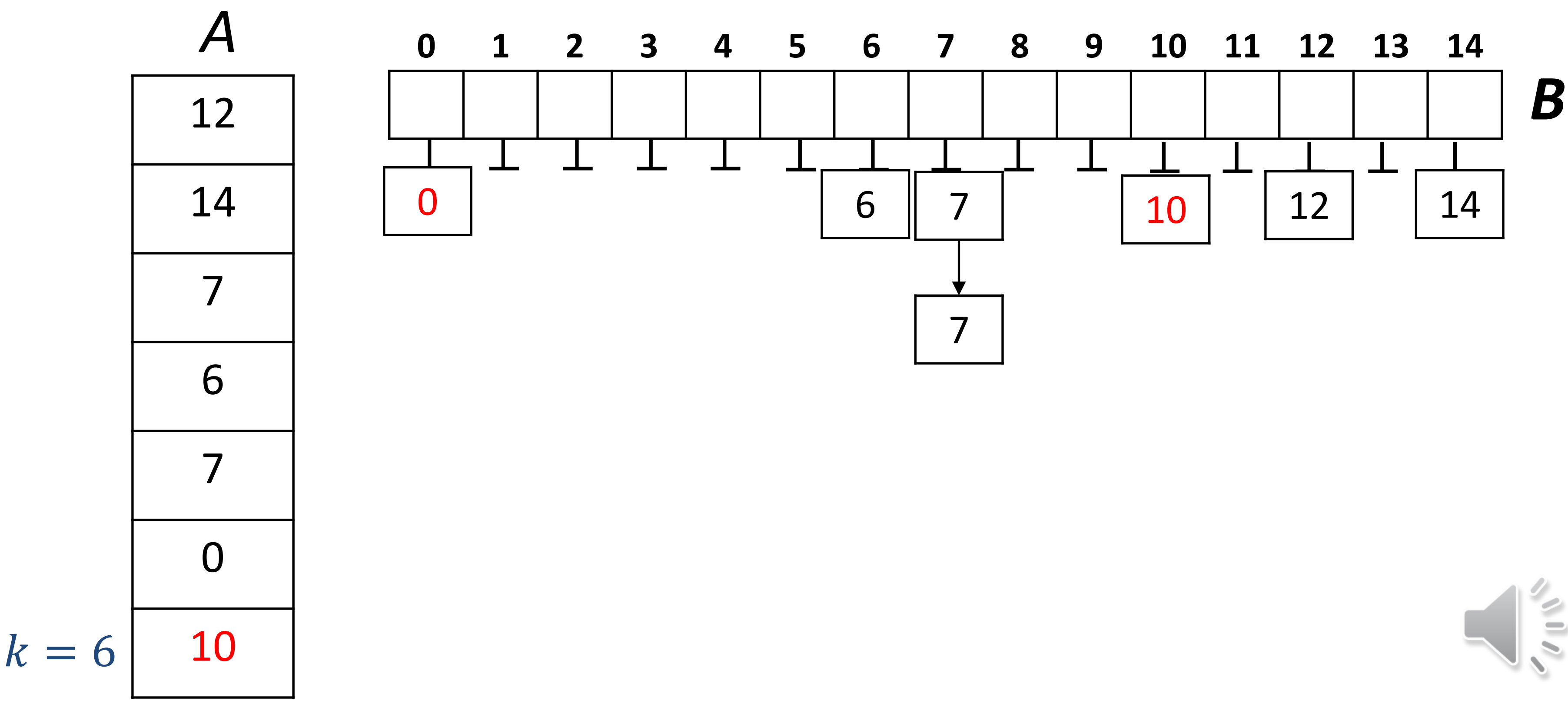
Bucket Sort

- 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 $L = 15$



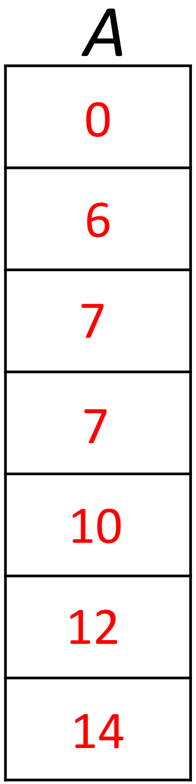
Bucket Sort

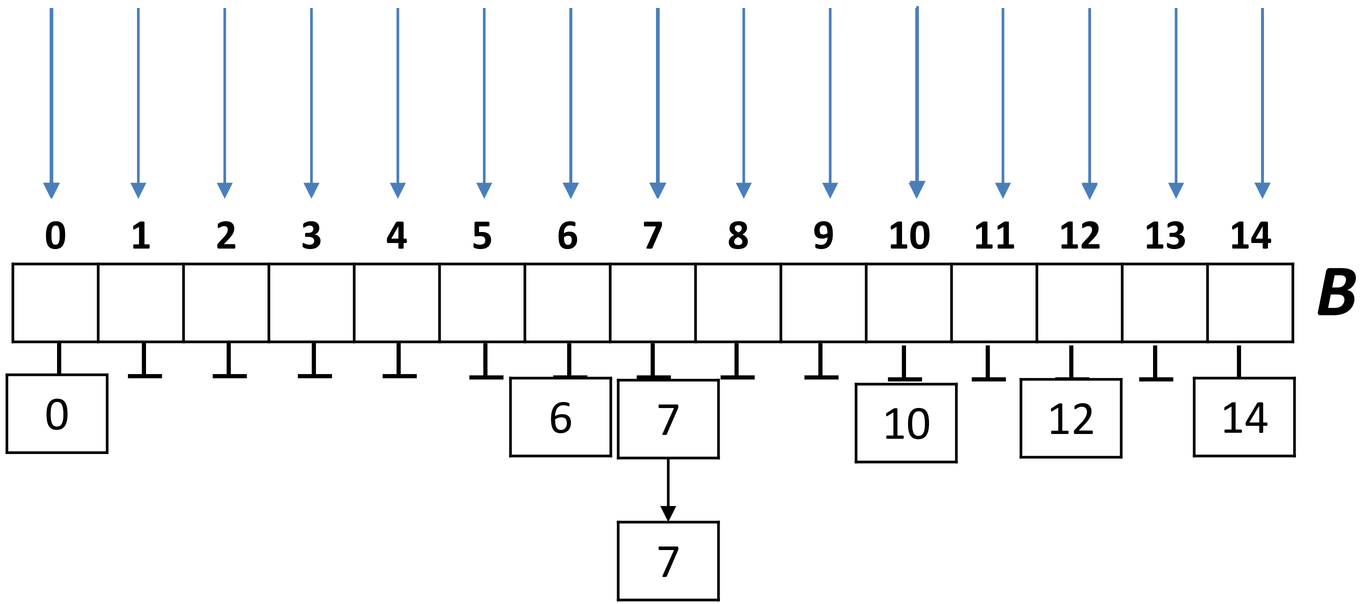
- Suppose all keys in A are integers in range $[0, \ldots, L-1]$
- Use an axillary *bucket array* $B[0, \ldots, L-1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



Bucket Sort

- Suppose all keys in A are integers in range $[0, \ldots, L-1]$
- 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

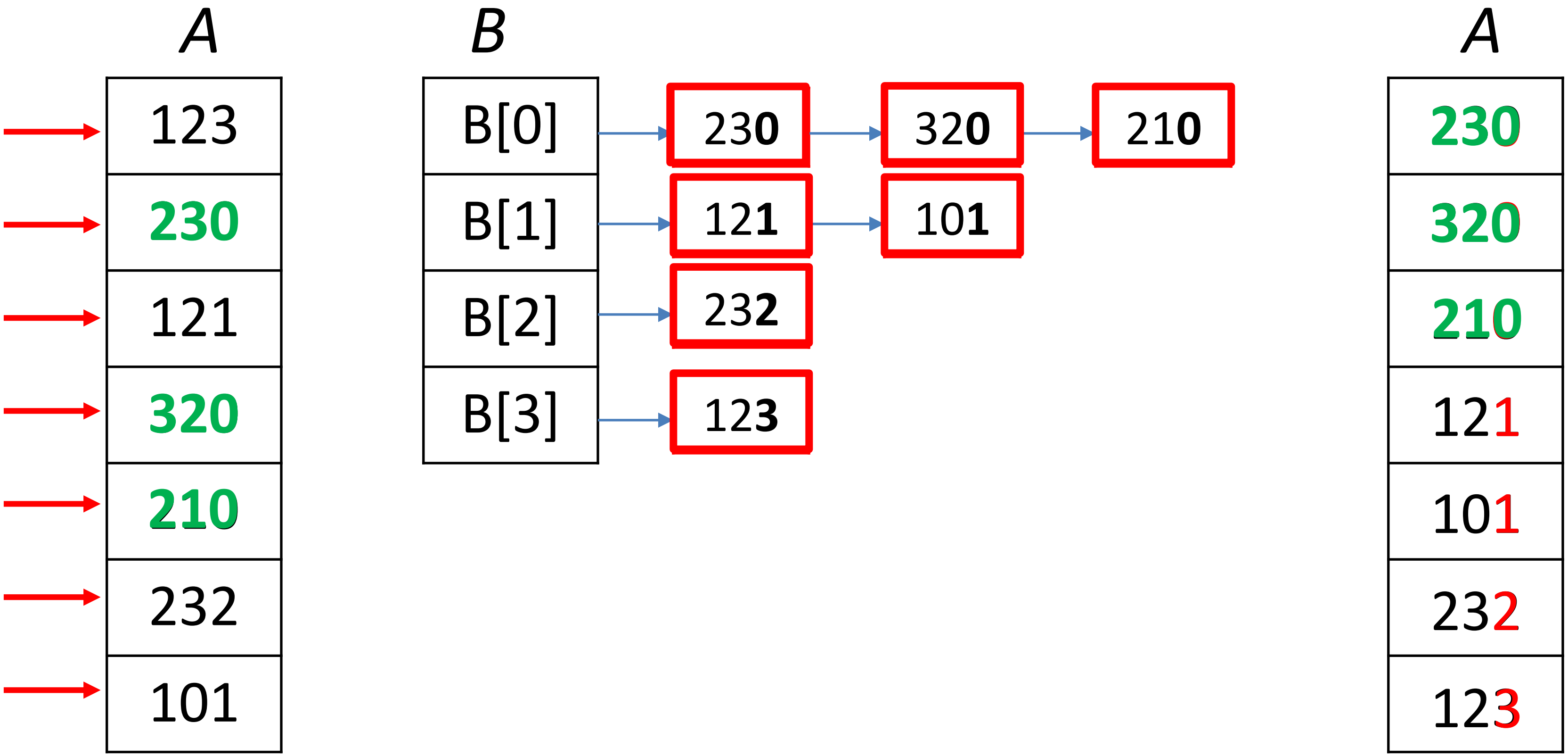
- Running time of bucket sort is $\Theta(L+n)$
  - n is size of A
  - L is range $[0, L)$ of integers in A
- What if L is much larger than n?
  - i.e. A has size 100, range of integers in A is $[0, \ldots, 99999]$
- Assume at most m digits in any key
  - pad with leading 0s

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 $B[0, \ldots, R-1]$ of empty lists (buckets)

for $i \leftarrow 0$ to $n-1$ **do**

$next \leftarrow A[i]$

append $next$ at end of $B[d$th digit of $next]$

$i \leftarrow 0$

for $j \leftarrow 0$ to $R-1$ **do**

while $B[j]$ is non-empty **do**

move first element of $B[j]$ to $A[i{+}{+}]$

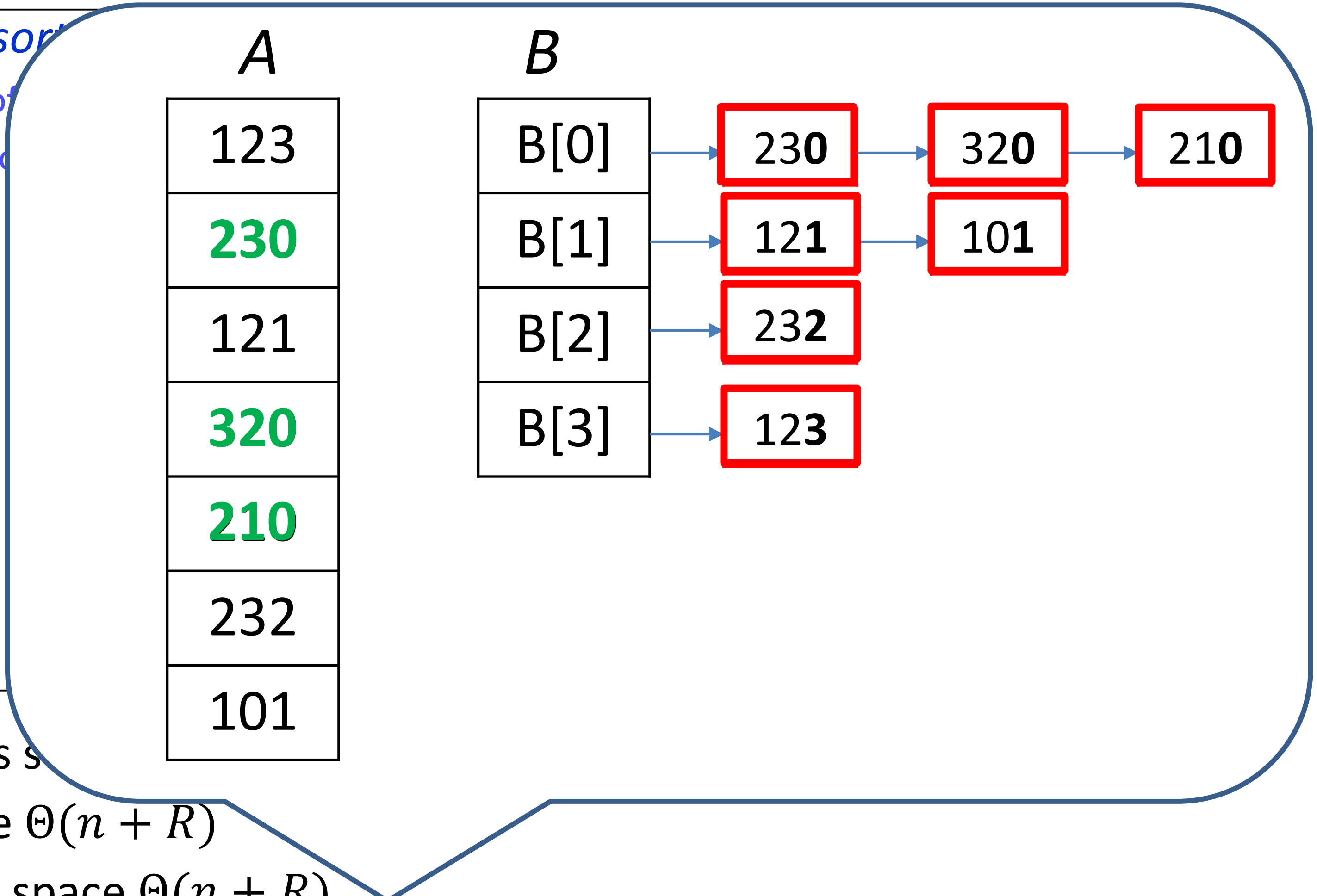
- Sorting is stable: equal items stay in original order
- Run-time $\Theta(n+R)$
- Auxiliary space $\Theta(n+R)$
  - $\Theta(R)$ for array B, and linked lists are $\Theta(n)$

Single Digit Bucket Sort

Bucket-sor

A : array of

d: index



- 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

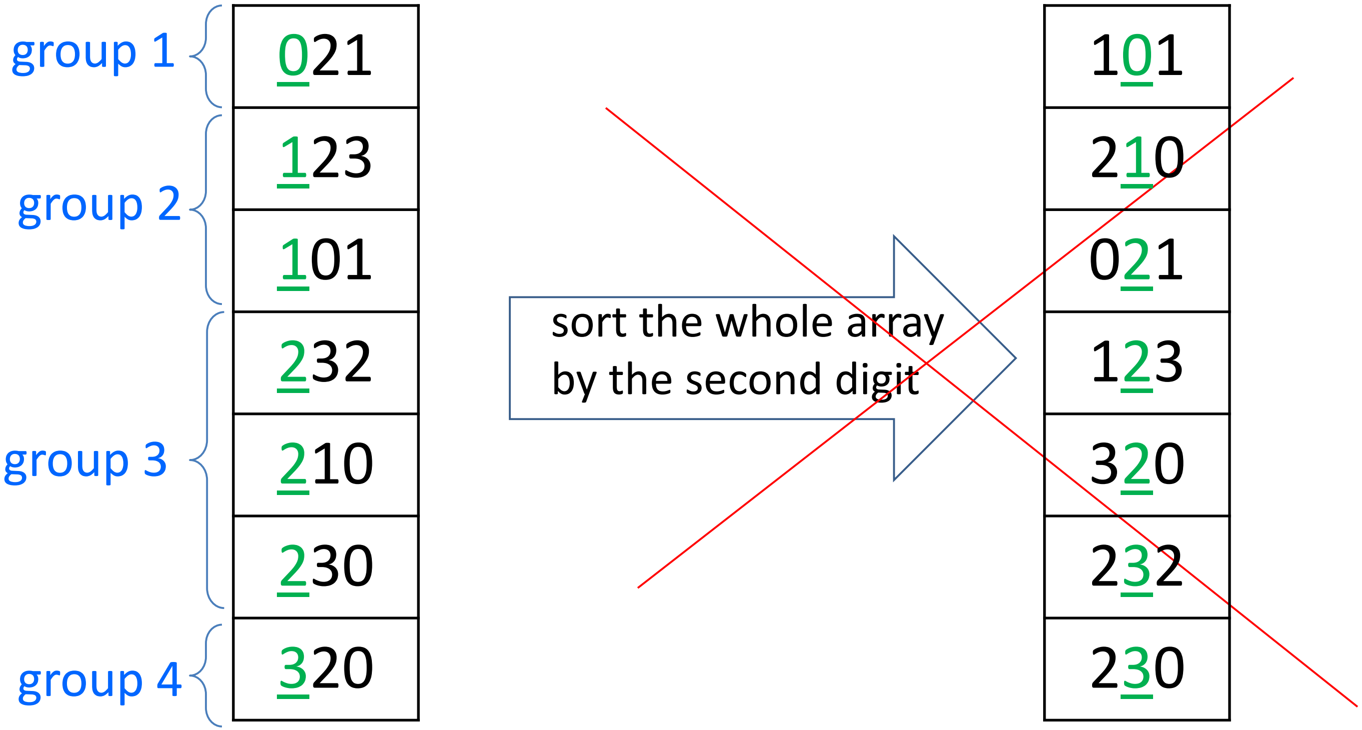
MSD-Radix-Sort

- 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

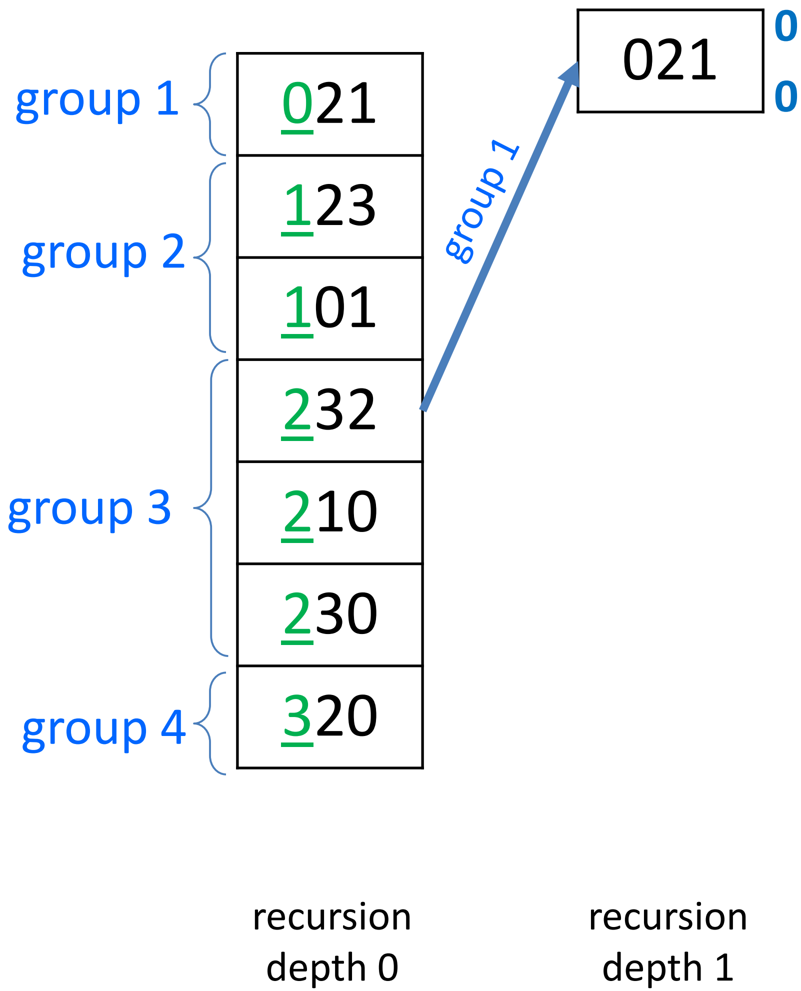
- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit



- 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

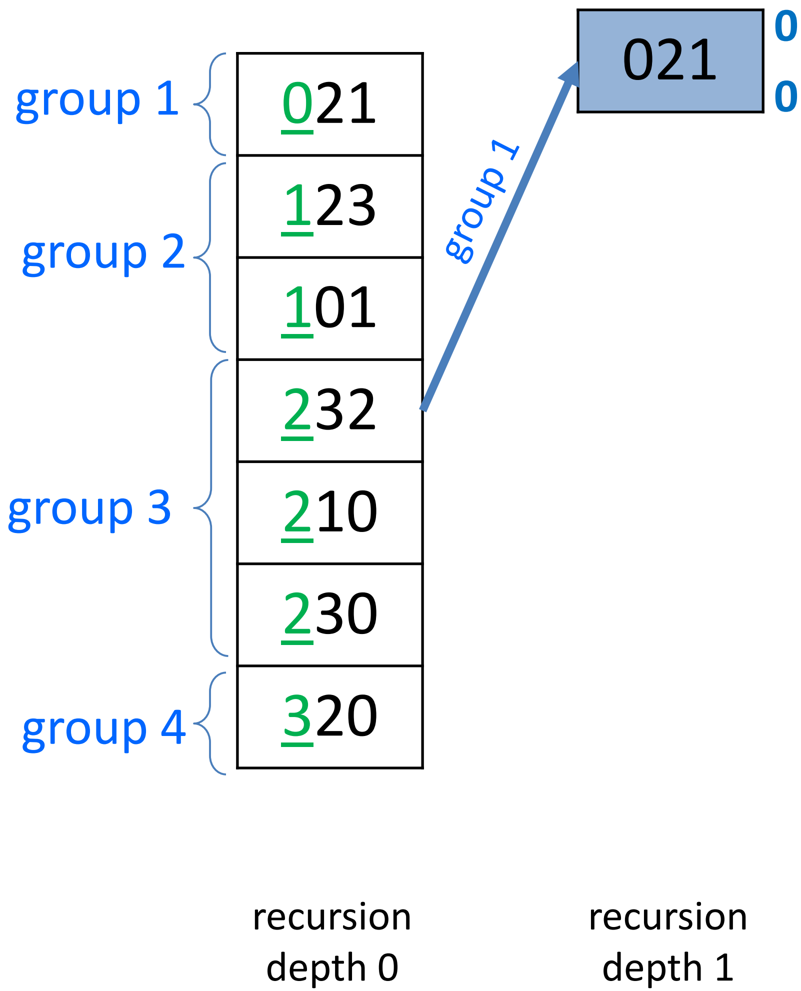
MSD-Radix-Sort

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



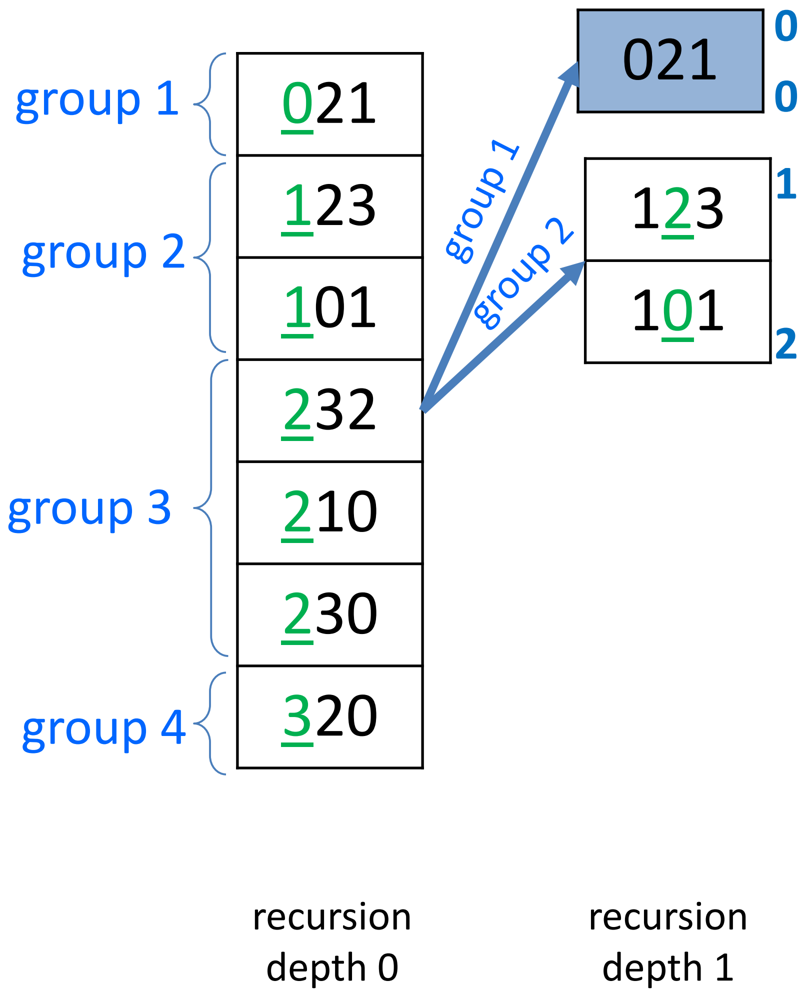
MSD-Radix-Sort

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



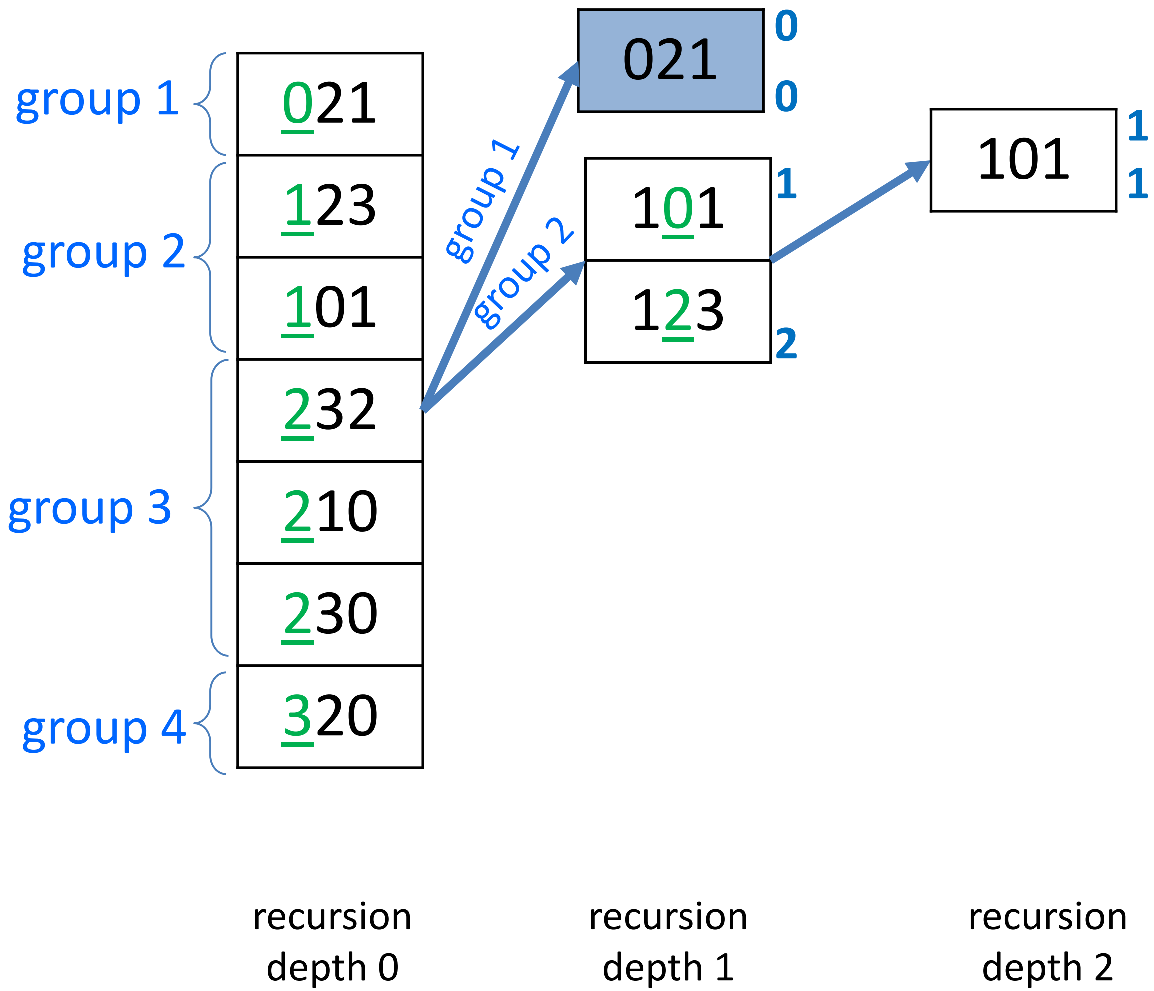
MSD-Radix-Sort

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



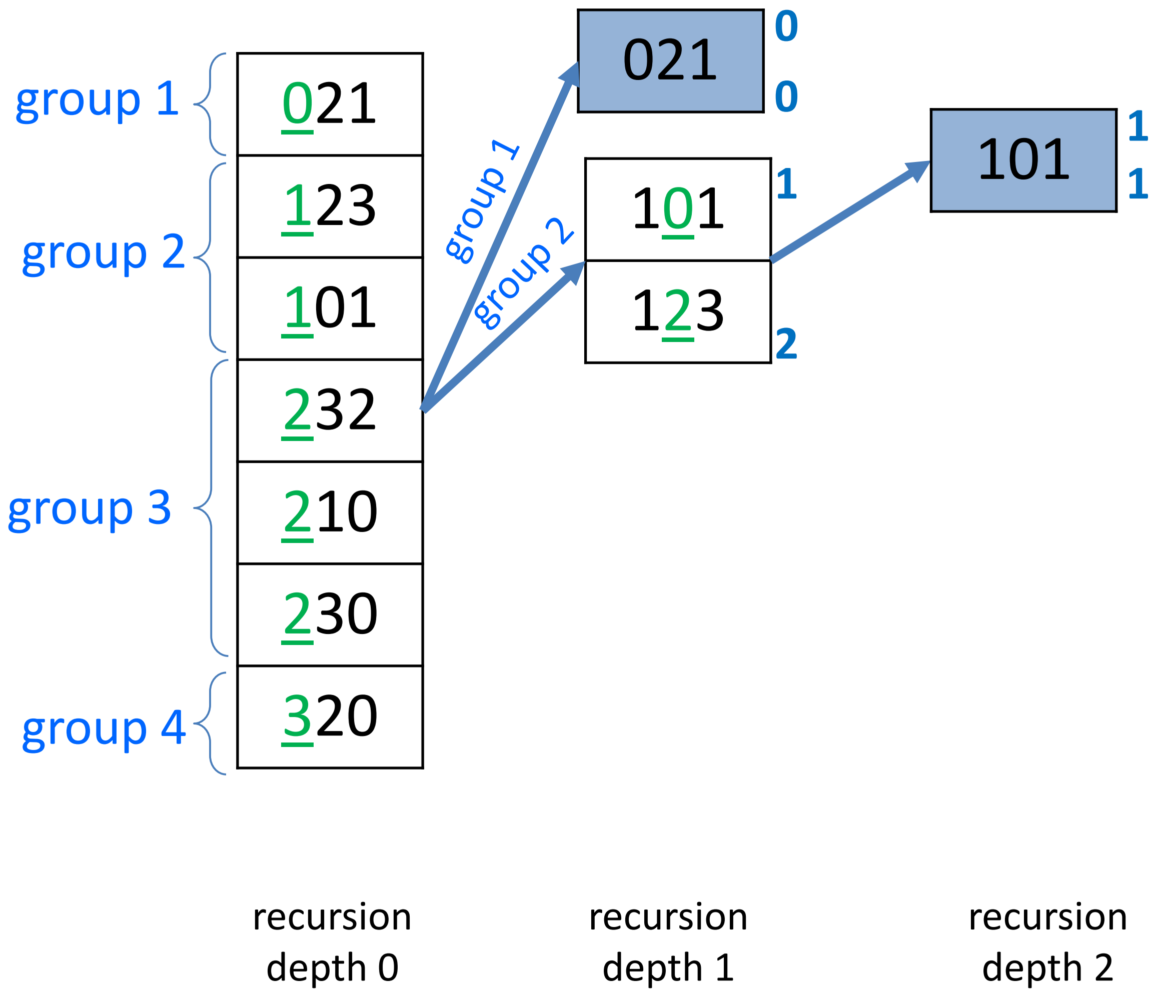
MSD-Radix-Sort

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



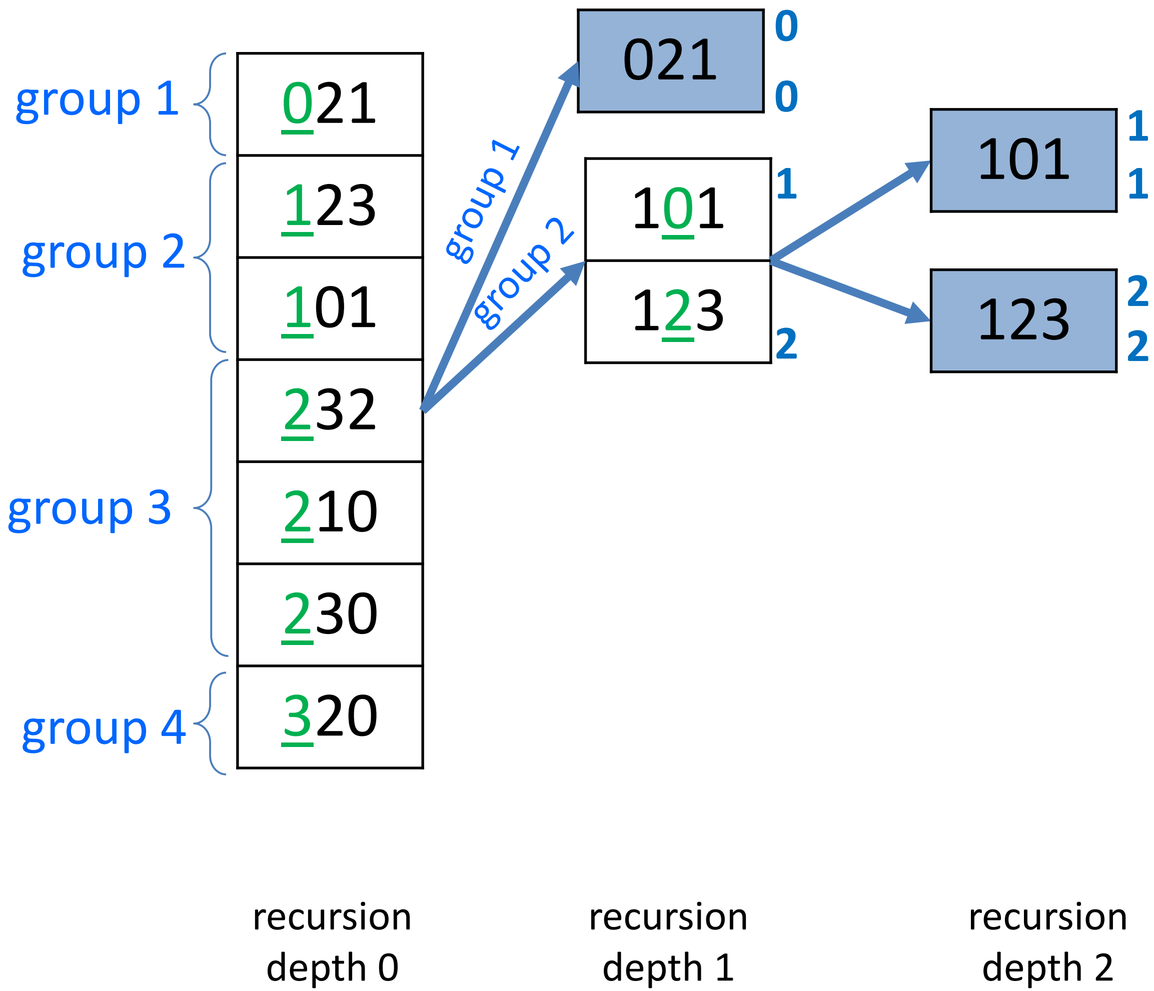
MSD-Radix-Sort

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



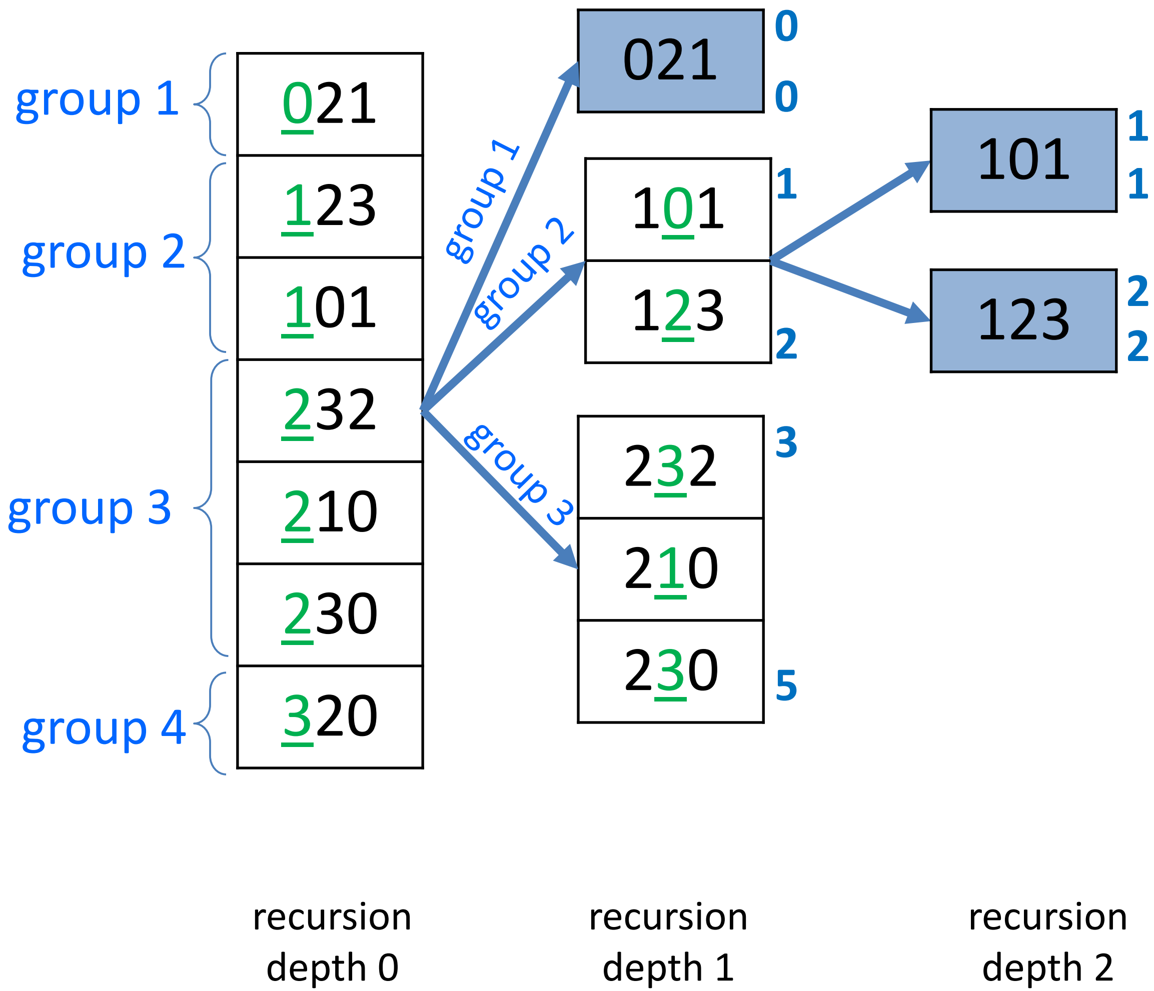
MSD-Radix-Sort

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



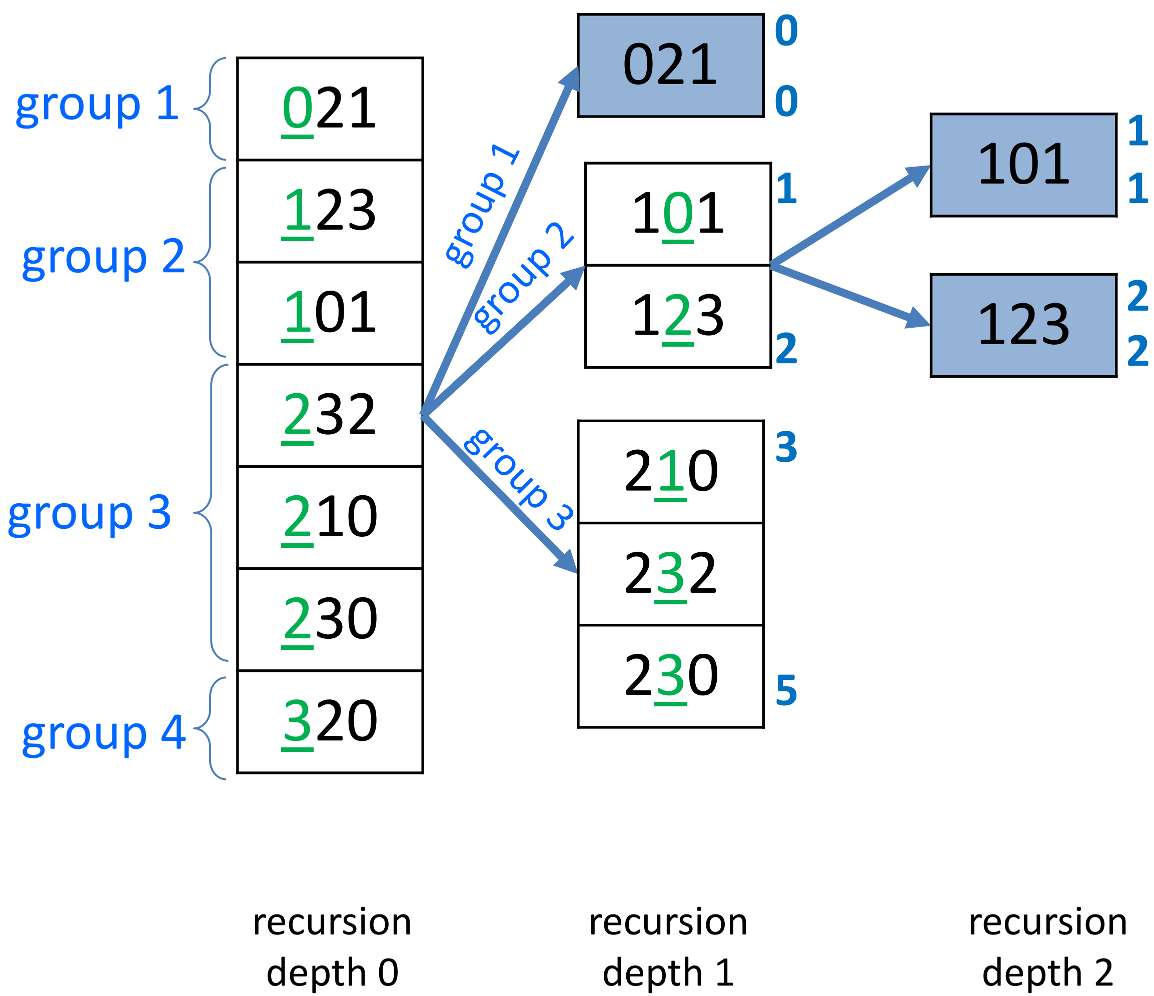
MSD-Radix-Sort

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



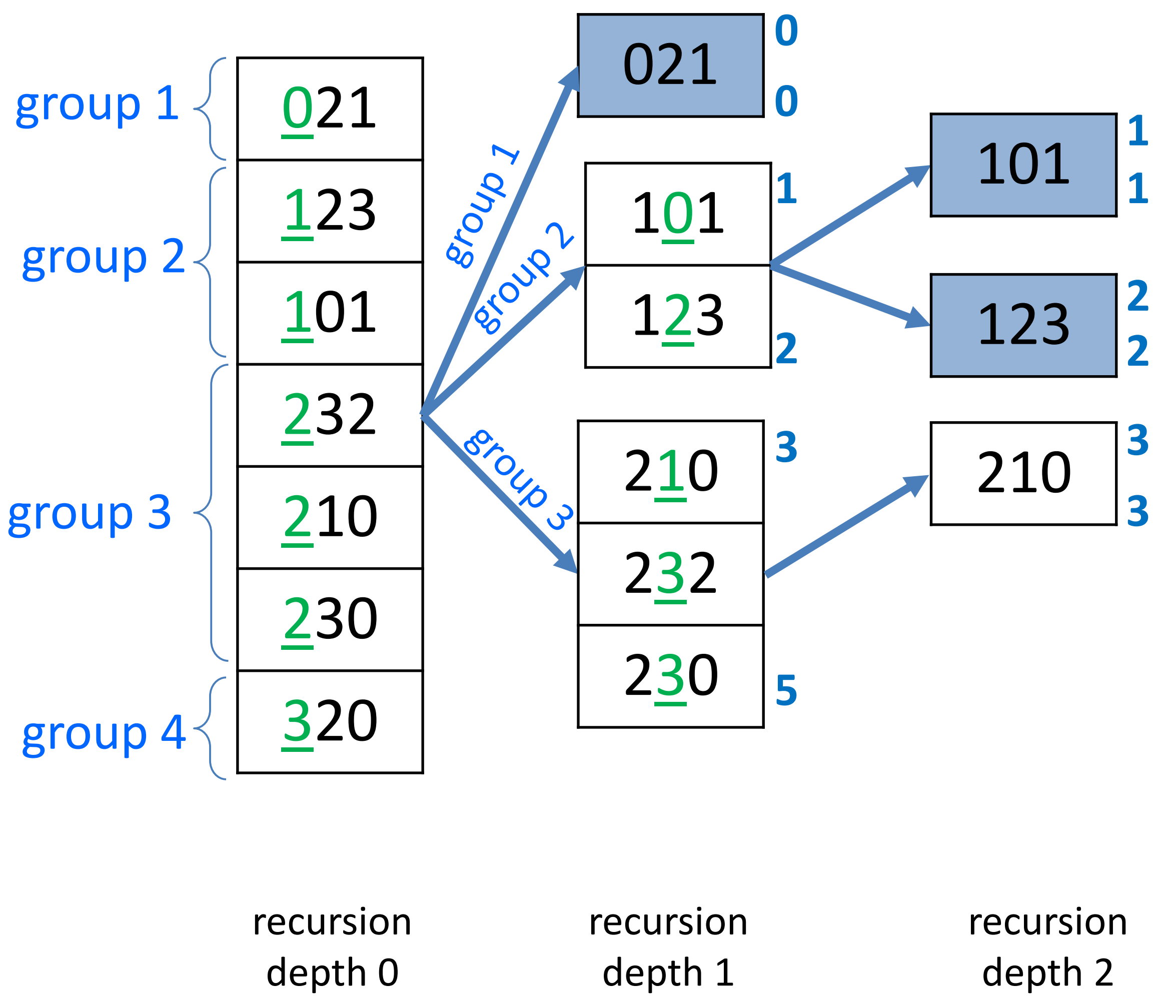
MSD-Radix-Sort

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



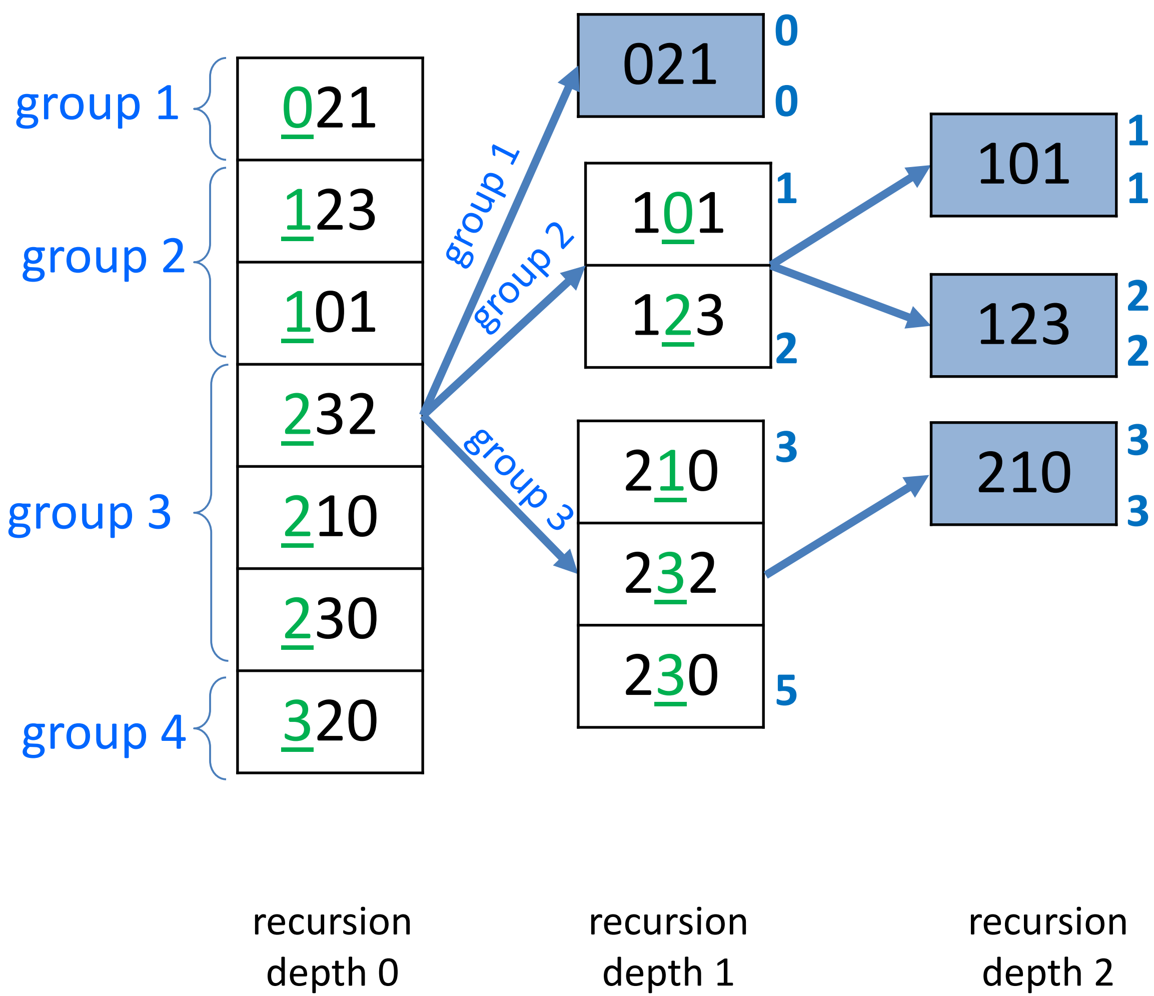
MSD-Radix-Sort

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



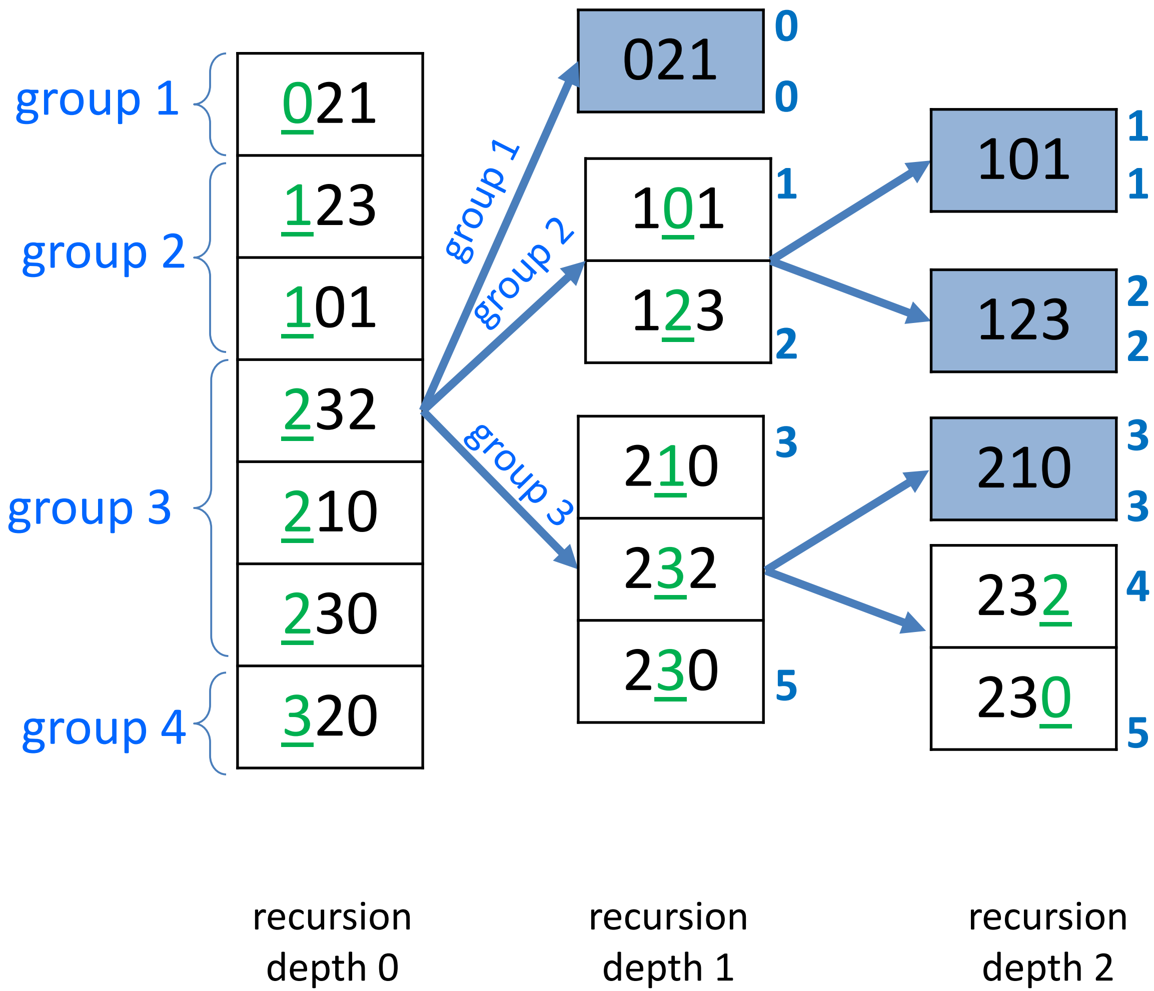
MSD-Radix-Sort

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



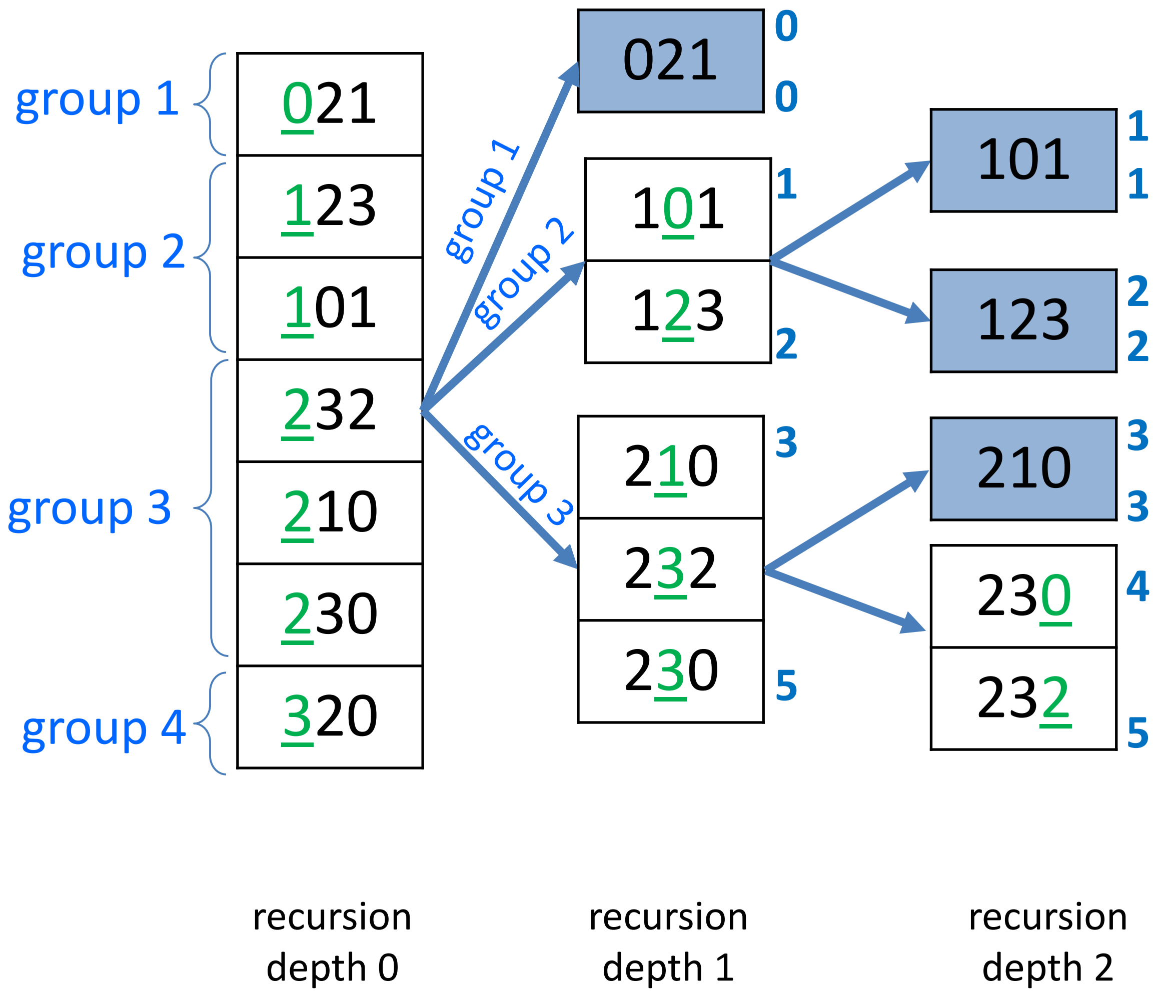
MSD-Radix-Sort

- 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 Pseudocode

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

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

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 $(l' \leq r)$ **do**

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

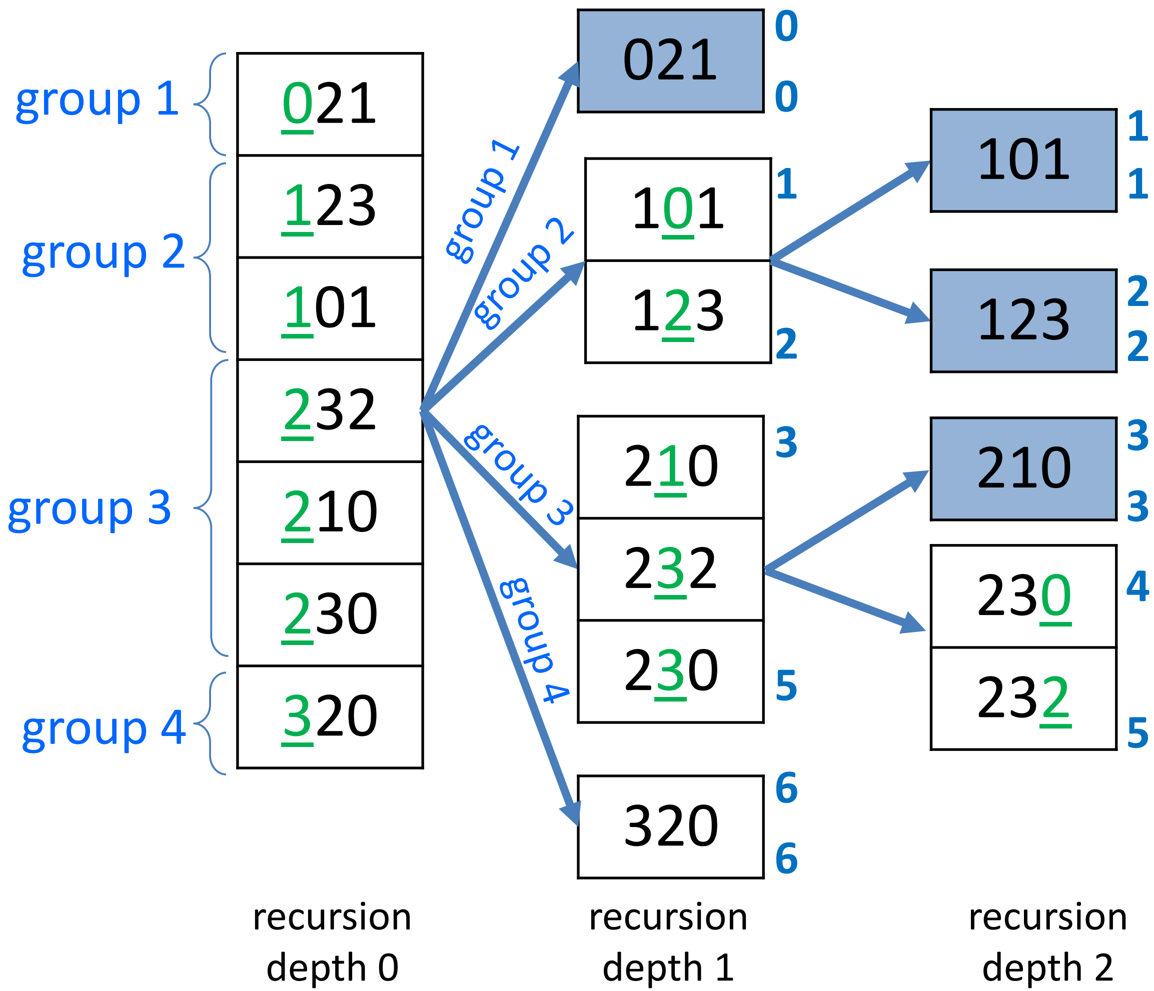
MSD-Radix-sort$(A, l', r', d+1)$

$l' \leftarrow r' + 1$

- 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

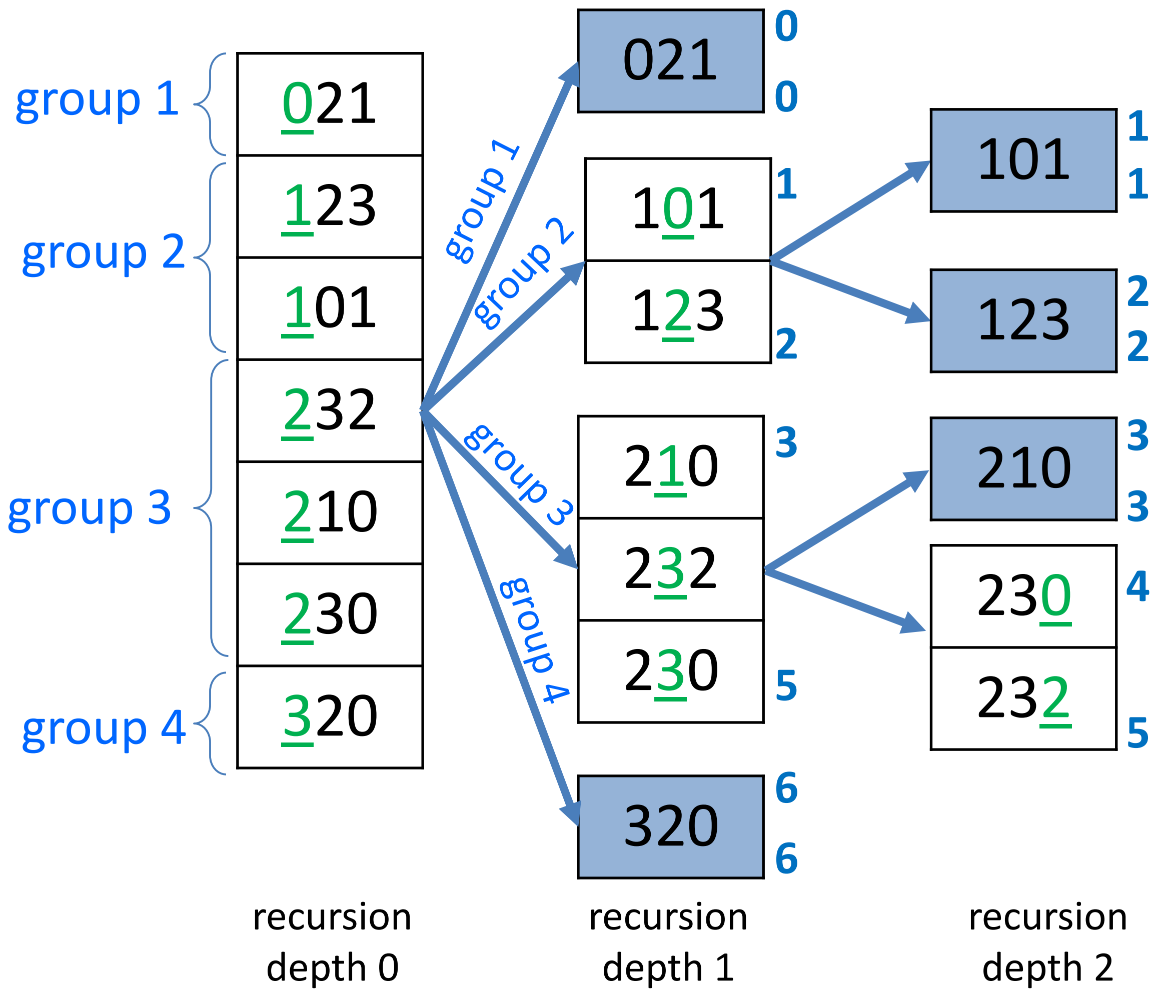
MSD-Radix-Sort

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



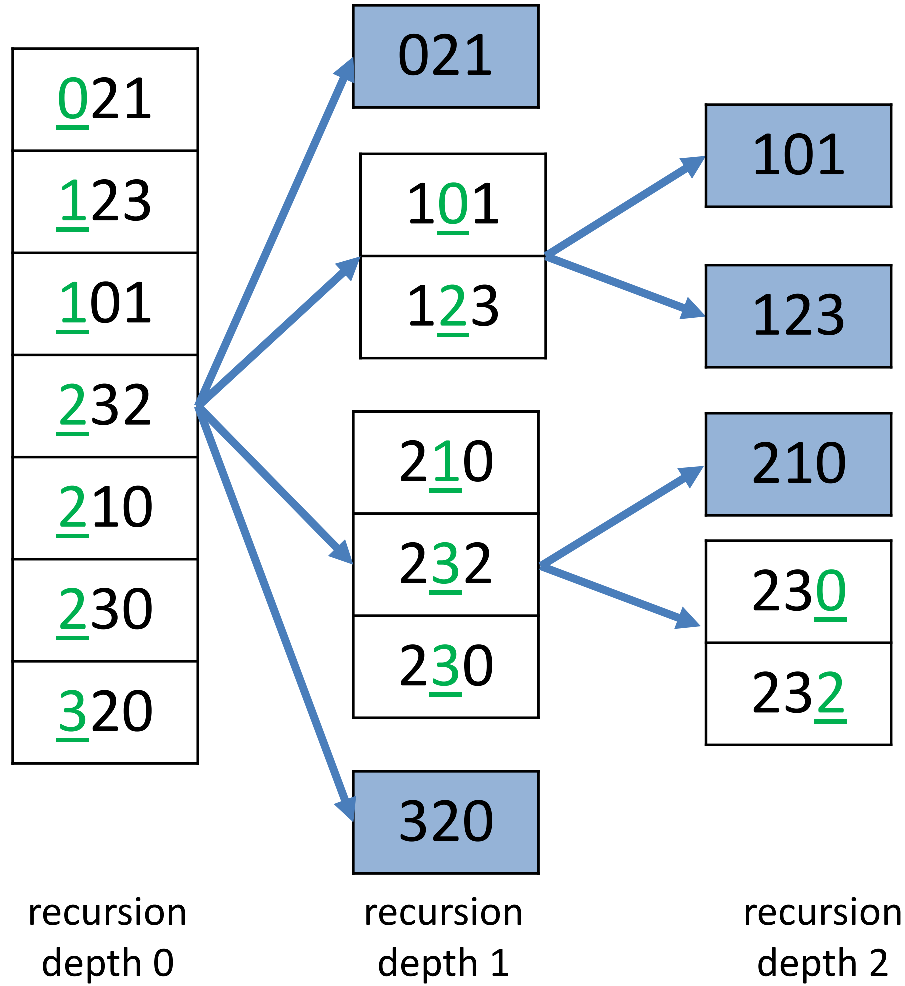
MSD-Radix-Sort

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



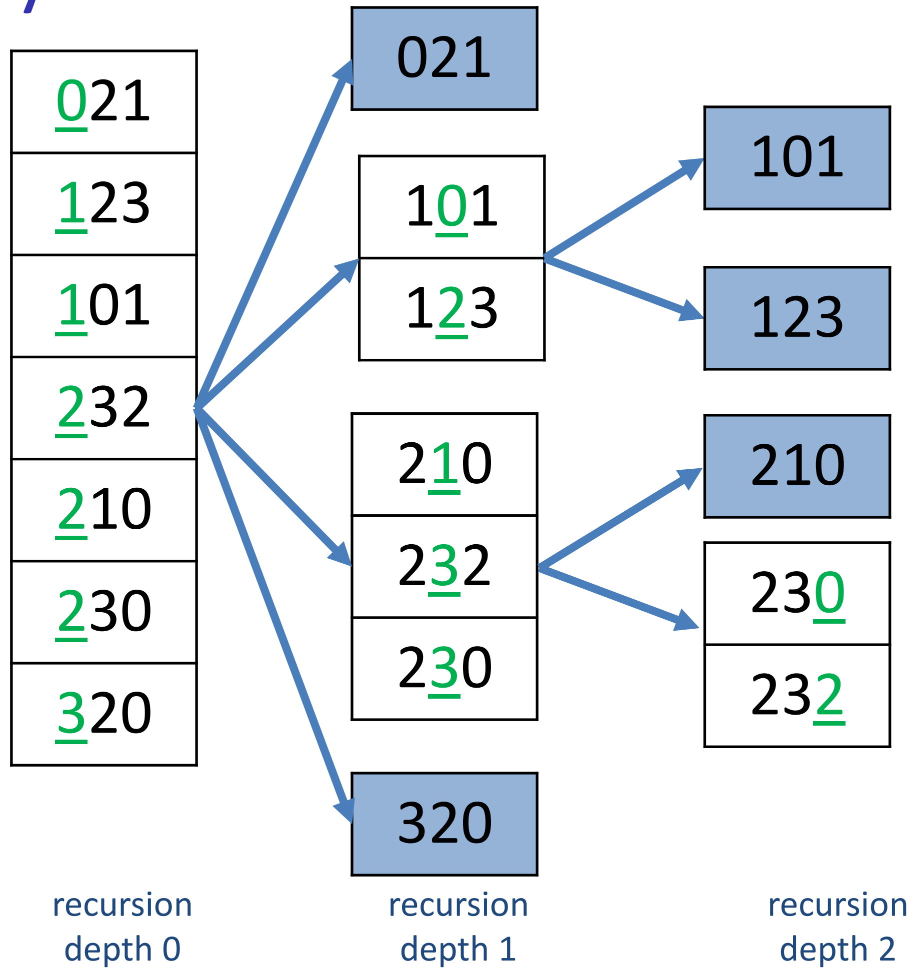
MSD-Radix-Sort Space Analysis

- 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
    - $k \leq n$
      - 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 $n_i + R$
    - $\sum_{i=0}^{k}(n_i+R) = n + \sum_{i=0}^{k} R \leq n + nR$
    - 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

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