CS 240 – Data Structures and Data Management

Module 3: Sorting, Average-case and Randomization

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

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: Motivation

- Worst-case run time is our default for analysis
- Best-case run time is also sometimes useful
- Sometimes, best-case and worst-case runtimes are the same
- But for some algorithms best-case and worst case differ significantly
    - worst-case runtime too pessimistic, best-case too optimistic
    - average-case run time analysis is useful especially in such cases

Average Case Analysis

- Recall average case runtime definition
    - let $\mathbb{I}_n$ be the set of all instances of size n

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

  - assume $|\mathbb{I}_n|$ is finite
  - can achieve 'finiteness' in a natural way for many problems

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

Average Case Analysis: Contrived Example

smallestFirst(A, n)

A: array storing n distinct integers in range $\{0,1,\ldots,n-1\}$

if $A[0] = 0$ **then**

  for $j = 1$ **to** n **do**

    print 'first is smallest'

else print 'first is not smallest'

$\mathbb{I}_3 =$

0	1	2
0	2	1
1	0	2
1	2	0
2	0	1
2	1	0

- Best-case
  - $A[0] \neq 0$
    - runtime is $O(1)$
- Worst case
  - $A[0] = 0$
    - runtime is $\Theta(n)$

Average Case Analysis: Contrived Example

smallestFirst(A, n)

A: array storing n distinct integers in range $\{0,1,\dots,n-1\}$

if $A[0] = 0$ **then**

for $j = 1$ **to** n **do**

print 'first is smallest'

else print 'first is not smallest'

$\mathbb{I}_3 =$

0	1	2
0	2	1
1	0	2
1	2	0
2	0	1
2	1	0

- $n!$ inputs in total
  - $(n-1)!$ inputs have $A[0] = 0$
    - runtime for each is cn
  - $n! - (n-1)!$ inputs have $A[0] \neq 0$
    - runtime for each is c

$$T^{avg}(n) = \frac{1}{|\mathbb{I}_n|}\sum_{I\in\mathbb{I}_n} T(I) = \frac{1}{n!}\Big(\overbrace{cn + \cdots + cn}^{(n-1)!} + \overbrace{c + \cdots c}^{n! - (n-1)!}\Big)$$

$$= \frac{1}{n!}\big(cn(n-1)! + c(n! - (n-1)!)\big) = c + c - \frac{c}{n} \in O(1)$$

Average Case Analysis: Example 2

```
all-0-test(w, n)
        //test if all entries of bitstring w[0..n − 1] are 0
        if (n = 0)  return true
        if (w[n − 1] = 1)  return false
         all-0-test(w, n − 1)
```

- Define $T(n)$ = # bit-comparisons on input w
  - asymptotically the same as runtime
    - runtime is c times # of bit comparisons
  - makes analysis a bit simpler, do not have to carry around constant c
- Best-case runtime
  - $w \;= *** \cdots *** \; 1$
  - $T(n) = 1$
    - return false after the first comparison
  - $\Theta(1)$

Average Case Analysis: Example 2

all-0-test(w, n)

    //test if all entries of bitstring $w[0..n-1]$ are 0

    if $(n = 0)$ **return** *true*

    if $(w[n-1] = 1)$ **return** *false*

    all-0-test$(w, n-1)$

- Worst-case runtime
  - $w = 000 \dots 000$
  - always go into recursion until $n = 0$
  - $T(n) = 1 + T(n-1)$
    - how to solve?

Average Case Analysis: Example 2

$$T(n) = \begin{cases} 1 + T(n-1) & n > 0 \\ 0 & n = 0 \end{cases}$$

- Solving: repeatedly expand until see the pattern

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

after 1 expansion: $T(n) = 2 + \underbrace{T(n-2)}_{1 + T(n-3)}$

after 2 expansions: $T(n) = 3 + T(n-3)$

after i expansions: $T(n) = (i+1) + T(n-(i+1))$

- Stop expanding when get to base case

$$T(n-(i+1)) = T(0) \;\Rightarrow n-(i+1) = 0$$

$$\Rightarrow i = n-1$$

- Thus $T(n) = n - 1 + 1 + T(0) = n \in \Theta(n)$

Average Case Analysis: Example 2

all-0-test(w, n)

//test if all entries of bitstring $w[0..n-1]$ are 0

if $(n = 0)$ **return** *true*

if $(w[n-1] = 1)$ **return** *false*

all-0-test$(w, n-1)$

- Worst-case runtime
  - $w = 000 \ldots 000$
  - always go into recursion until $n = 0$
  - $T(n) = 1 + T(n-1)$
  - resolves to $\Theta(n)$

Average Case Analysis: Example 2

```
all-0-test(w, n)
        //test if all entries of bitstring w[0..n − 1] are 0
        if (n = 0) return true
        if (w[n − 1] = 1) return false
         all-0-test(w, n − 1)
```

B_3

0	0	1	0	0	0
1	0	1	0	1	0
0	1	1	1	0	0
1	1	1	1	1	0

B_2

- Let B_n be the set of all bitstrings of length n
    - note $|B_n| = 2|B_{n-1}|$
- Average runtime

$$T^{avg}(n) = \frac{1}{|B_n|} \sum_{w \in B_n} T(w)$$

- Recursive formula for one non-empty bitstring w

$$T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ 1 + T(w[0..n-2]) & \text{otherwise} \end{cases}$$

- This formula is for one particular bitstring w, **not** for average case runtime

Average Case Analysis: Example 2

$$T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ 1 + T(w[0..n-2]) & \text{otherwise} \end{cases}$$

$$T^{avg}(n) = \frac{1}{|B_n|} \sum_{w \in B_n} T(w)$$

$$= \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=1}} T(w) \quad + \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=0}} T(w)$$

0	0	1
1	0	1
0	1	1
1	1	1

0	0	0
0	1	0
1	0	0
1	1	0

$$= \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=1}} 1 \quad + \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=0}} (1 + T(w[0..n-2]))$$

$$= \frac{1}{2} \quad + \frac{1}{2} \quad + \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=0}} T(w[0..n-2])$$

Average Case Analysis: Example 2

$$T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ 1 + T(w[0..n-2]) & \text{otherwise} \end{cases}$$

$$T^{avg}(n) = 1 + \frac{1}{|B_n|} \sum_{\substack{w \in B_n \\ w[n-1]=0}} T(w[0..n-2])$$

B_2

0	0	0
0	1	0
1	0	0
1	1	0

$$= 1 + \frac{1}{|B_n|} \sum_{v \in B_{n-1}} T(v)$$

$$= 1 + \frac{|B_{n-1}|}{|B_n|} \frac{1}{|B_{n-1}|} \sum_{v \in B_{n-1}} T(v) = 1 + \frac{1}{2} T^{avg}(n-1)$$

- Recurrence $T^{avg}(n) = 1 + \frac{1}{2} T^{avg}(n-1)$ resolves to $\Theta(1)$

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

```
simple(A, n)
  A: array storing n numbers
    sum ← 0
    if random(3) = 0 then return sum
    else
       for i ← 0 to n − 1 do
          sum ← sum + A[i]
       return sum
```

$random(m)$ returns integer sampled uniformly from $\{0, 1, \ldots, m-1\}$, so $\Pr(random(3) = 0) = 1/3$

- A *randomized algorithm* is one which relies on random numbers for some steps
- Runtime depends on both input I and random numbers R used
- Side note: computers cannot generate truly random numbers
  - assume there is pseudo-random number generator (PRNG), a deterministic program that uses initial *seed* to generate 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?
    - depends on input I and on R, sequence of random numbers algorithm choses
- Define $T(I,R)$ to be running time of randomized algorithm for instance I and R
- *Expected runtime 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 define best-case and average-case expected running time, but usually consider only worst-case expected runtime
- Sometimes also want to know running time if get really unlucky with random numbers R, i.e. worst case (or worst instance and worst random numbers case)

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

Expected Running Time Example

$simple(A, n)$

A: array storing n numbers

$sum \leftarrow 0$

if $random(3) = 0$ **then return** sum

else

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

$sum \leftarrow sum + A[i]$

return sum

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

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

- *simple* needs only one random number: $\Pr(0) = \Pr(1) = \Pr(2) = \frac{1}{3}$

$$T^{exp}(I) = T(I,0) \cdot \Pr(0) + T(I,1) \cdot \Pr(1) + T(I,2) \cdot \Pr(2)$$

$$= T(I,0) \cdot \frac{1}{3} + T(I,1) \cdot \frac{1}{3} + T(I,2) \cdot \frac{1}{3}$$

$$= c \cdot \frac{1}{3} + c \cdot n \cdot \frac{1}{3} + c \cdot n \cdot \frac{1}{3} \in \Theta(n)$$

- All instances have the same expected runtime, so $T^{exp}(n) \in \Theta(n)$

Randomized Algorithm: *Simple2*

```
simple2(A, n)
  A: array storing n numbers
    sum ← 0
    r1 ← to random(n), r2 ← to random(n)
     for i ← 1 to r1 do
        for j ← 1 to r2 do
           sum ← sum + A[j]A[i]
```

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

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

- Uses 2 random numbers $R = \langle r_1, r_2 \rangle$: $\Pr(r_1 = 0) = \cdots = \Pr(r_1 = n-1) = \frac{1}{n}$

$$\Pr[\langle 0,0 \rangle] = \Pr[\langle 0,1 \rangle] = \cdots = \Pr[\langle n-1, n-1 \rangle] = \left(\frac{1}{n}\right)^2$$

$$T^{exp}(I) = \sum_{\langle r_1, r_2 \rangle} T(I, \langle r_1, r_2 \rangle) \cdot \left(\frac{1}{n}\right)^2 = \left(\frac{1}{n}\right)^2 \sum_{r_1 \in \{0,1,\ldots,n-1\}} c \cdot r_1 \sum_{r_2 \in \{0,1,\ldots,n-1\}} r_2$$

$$= \left(\frac{1}{n}\right)^2 \sum_{r_1} c \cdot r_1 \frac{n(n-1)}{2} = \left(\frac{1}{n}\right)^2 c \frac{n(n-1)}{2} \frac{n(n-1)}{2}$$

- All instances have the same running time, so $T^{exp}(n) \in \Theta(n^2)$

Why Use Randomized Algorithms

1) improved running time
   - often design a randomized algorithm so that all instances of size n have the same expected runtime
2) improved solution
   - not studied in this course

Randomized Algorithms to Improve Runtime

```
all-0-test(w, n)
    //test if all entries of bitstring w[0..n − 1] are 0
    if (n = 0) return true
    if (w[n − 1] = 1) return false
     all-0-test(w, n − 1)
```

- Average case $O(1)$
- Worst-case $O(n)$
- Would hope that in practice, time averaged over **different runs** is $O(1)$
- However, average-cases analysis averages over **instances**, not **runs**
  - cannot average over runs, do not know the instances the user will choose
- Suppose all instances are equally likely to occur in practice
  - then averaging over **different runs** is equivalent to averaging over **instances**
  - so can expect *all-0-test* to have $O(1)$ runtime averaged over runs
- However humans often generate instances that are far from equally likely
  - if user calls *all-0-test* on almost reverse sorted arrays, runtime averaged over **different runs** is $\Theta(n)$ in practice
  - real-life example: humans invoke sorting algorithm most often on arrays that are already almost sorted

Randomized Algorithms to Improve Runtime

randomized-all-0-test(w, n)
  //test if all entries of bitstring $w[0..n-1]$ are 0
  if $(n = 0)$ **return** *true*
  if $(random(2) = 0)$ **then**
    $w[n-1] = 1 - w[n-1]$
  if $(w[n-1] = 1)$ **return** *false*
  randomized-all-0-test$(w, n-1)$

- Randomization can improve runtime in practice if instances are not equally likely
  - makes sense to employ when average case runtime is better than worst case runtime
- Randomization can shift dependence from what we cannot control (user) to what we can control (random number generation)
  - improved runtime in practice
  - no more bad instances!
  - could still have unlucky numbers
    - if running time is long on some run, it is because we generated unlucky random numbers, not because of the instance itself
    - exceedingly rare, think of chances of creating a string containing all zeros by performing random flips on w

Randomized Algorithm *randomized-all-0-test*

```
randomized-all-0-test(w, n)
    //test if all entries of bitstring w[0..n − 1] are 0
    if (n = 0) return true
    if (random(2) = 0) then
        w[n − 1] = 1 − w[n − 1]
    if (w[n − 1] = 1) return false
    randomized-all-0-test(w, n − 1)
```

- Running time depends **both** on input w **and** sequence R of generated random
  - $w = 0110,\ R = \langle 1,0,1 \rangle$
  - Step 1:
    $w = 0110 \quad R = \langle 1,0,1 \rangle \quad \Rightarrow w = 0110 \quad \Rightarrow$ make recursive call
  - Step 2:
    $w = 011 \quad R = \langle 1,0,1 \rangle \Rightarrow w = 010 \quad \Rightarrow$ make recursive call
  - Step 3:
    $w = 01 \quad R = \langle 1,0,1 \rangle \Rightarrow w = 01 \quad \Rightarrow$ return *false*
- Recursion if $w[n-1] \neq random\ number$, return *false* otherwise

Expected Runtime of *randomized-all-0-test*

```
randomized-all-0-test(w, n)
    //test if all entries of bitstring w[0..n − 1] are 0
    if (n = 0)  return true
    if (random(2) = 0)  then
        w[n − 1] = 1 − w[n − 1] // the only change
    if (w[n − 1] = 1)  return false
    randomized-all-0-test(w, n − 1)
```

- Let $T(w, R)$ be # of bit-comparisons on input w if the random outcomes are R
  - this is proportional to runtime
- $R = \langle x,\ R' \rangle$
  - x is the first random number
  - R' are the other random numbers (if any) for the recursions
- By random number independence, $\Pr(R) = \Pr(x)\Pr(R')$
- Recursive formula for an arbitrary instance w (any bitstring)

$$T(w,R) = T(w, < x, R' >) = \begin{cases} 1 & \text{if } x = w[n-1] \\ 1 + T(w[0..n-2], R') & \text{otherwise} \end{cases}$$

Expected Runtime of *randomized-all-0-test*

$$T^{exp}(w) = \sum_{R} \Pr(R) \cdot T(w, R) \quad = \sum_{<x,R'>} \Pr(R')\Pr(x) \cdot T(w, < x, R' >)$$

$$= \frac{1}{2} \sum_{<x,R'>} \Pr(R') \cdot T(w, < x, R' >)$$

0	0	0
0	0	1
0	1	0
0	1	1

1	0	0
1	0	1
1	1	0
1	1	1

$$= \frac{1}{2} \sum_{R'} \Pr(R') \cdot T(w, < x = w[n-1], R' >) + \frac{1}{2} \sum_{R'} \Pr(R') \cdot T(w, < x \neq w[n-1], R' >)$$

0	0	0
0	0	1
0	1	0
0	1	1

if $w[n-1] = 0$

1	0	0
1	0	1
1	1	0
1	1	1

Expected Runtime of *randomized-all-0-test*

$$T^{exp}(w) = \sum_{R} \Pr(R) \cdot T(w,R) = \sum_{<x,R'>} \Pr(R')\Pr(x) \cdot T(w, < x, R' >)$$

$$= \frac{1}{2} \sum_{<x,R'>} \Pr(R') \cdot T(w, < x, R' >)$$

0	0	0
0	0	1
0	1	0
0	1	1

1	0	0
1	0	1
1	1	0
1	1	1

$$= \frac{1}{2} \sum_{R'} \Pr(R') \cdot T(w, < x = w[n-1], R' >) + \frac{1}{2} \sum_{R'} \Pr(R') \cdot T(w, < x \neq w[n-1], R' >)$$

1	0	0
1	0	1
1	1	0
1	1	1

if $w[n-1] = 1$

0	0	0
0	0	1
0	1	0
0	1	1

Expected Runtime of *randomized-all-0-test*

$$T(w, < x, R' >) = \begin{cases} 1 & \text{if } x = w[n-1] \\ 1 + T(w[0..n-2], R') & \text{otherwise} \end{cases}$$

$$T^{exp}(w) = \frac{1}{2}\sum_{R'} \Pr(R') \cdot T(w, < x = w[n-1], R' >) + \frac{1}{2}\sum_{R'} \Pr(R') \cdot T(w, < x \neq w[n-1], R' >)$$

$$= \frac{1}{2}\sum_{R'} \Pr(R') \cdot 1 + \frac{1}{2}\sum_{R'} \Pr(R') \cdot (1 + T(w[0..n-2], R'))$$

$$= \frac{1}{2} + \frac{1}{2}\sum_{R'} \Pr(R') \cdot 1 + \frac{1}{2}\sum_{R'} \Pr(R') \cdot T(w[0..n-2], R')$$

$$= \frac{1}{2} + \frac{1}{2} + \frac{1}{2}\sum_{R'} \Pr(R') \cdot T(w[0..n-2], R')$$

Expected Runtime of *randomized-all-0-test*

$$T(w, <x, R'>) = \begin{cases} 1 & \text{if } x = w[n-1] \\ 1 + T(w[0..n-2], R') & \text{otherwise} \end{cases}$$

$$T^{exp}(w) = \sum_{R} \Pr(R) \cdot T(w, R) = 1 + \frac{1}{2} \sum_{R'} \Pr(R') \cdot T(w[0..n-2], R')$$

$$= 1 + \frac{1}{2} T^{exp}(\text{some instance of size } n-1)$$

$$C \leq \max\{A, B, C, \ldots, Z\}$$

Expected Runtime of *randomized-all-0-test*

$$T(w, < x, R' >) = \begin{cases} 1 & \text{if } x = w[n-1] \\ 1 + T(w[0..n-2], R') & \text{otherwise} \end{cases}$$

$$T^{exp}(w) = \sum_{R} \Pr(R) \cdot T(w, R) = 1 + \frac{1}{2} \sum_{R\prime} \Pr(R') \cdot T(w[0..n-2], R')$$

$$= 1 + \frac{1}{2} T^{exp}(\text{some instance of size } n-1)$$

$$\leq 1 + \frac{1}{2} \max_{v \in B_{n-1}} T^{exp}(v)$$

$$= 1 + \frac{1}{2} T^{exp}(n-1)$$

- $T^{exp}(w) \leq 1 + \frac{1}{2} T^{exp}(n-1)$ is true **for all** w
- Therefore $T^{exp}(n) = \max_{w \in B_n} T^{exp}(w) \leq 1 + \frac{1}{2} T^{exp}(n-1)$

Expected Running Time of *randomized-all-0-test*

- Recurrence $T^{exp}(n) \leq \frac{1}{2} T^{exp}(n-1)$
  - recurrence inequality solved just as equality by expansion
  - resolves to $\Theta(1)$
- Expected running time is $O(1)$
- Same recurrence as for average case *all-0-test*
  - $T^{avg}(n) = 1 + \frac{1}{2} T^{avg}(n-1)$
- Recall *randomized-all-0-test* is very similar to *all-0-test*
  - the only difference is a random bit flip
- Is expected time of randomized version always the same as average case time of non-randomized version?
  - no in general (depends on randomization)
  - yes if randomization is a shuffle
    - choose instance randomly with equal probability

Average-case vs. Expected runtime

AlgoritmShuffled(n)
among all instances I of size $\boldsymbol{n}$ for *Algorithm*
choose I randomly and uniformly
Algorithm(I, n)

- Ignoring time needed for the first two lines

$$T^{exp}(n) = \sum_{I \in \mathbb{I}_n} \Pr(I \text{ is chosen})\, T(I) = \sum_{I \in \mathbb{I}_n} \frac{1}{|\mathbb{I}_n|} T(I) = T^{avg}(n)$$

- Expected runtime of *AlgorithmShuffled* is equal to the average case time of *Algorithm*
- Computing expected runtime of *AlgorithmShuffled* is usually easier than computing average case time of *Algorithm*
    - this gives a different way to compute average case runtime

Average-case vs. Expected runtime

```
shuffle-all-0-test(n)
for (i = n − 1; i ≥ 0, i −−) do
    w[i] = random(2)
for (i = n − 1; i ≥ 0, i −−) do
    if (w[n − 1] = 1) then return false
return true
```

```
randomized-all-0-test(w, n)
for (i = n − 1; i ≥ 0, i −−) do
    if (random(2) = 0) then
        w[i] = 1 − w[i]
    if (w[n − 1] = 1) then return false
return true
```

- Example: randomized *all-0-test*, rephrased with for-loops
- These algorithms are not quite the same, but this does not matter for the expected number of bit comparisons
  - either way, at the time of comparison, the bit is 1 with probability ½
- Therefore, the average time of *all-0-test* can be deduced without analyzing $T^{avg}_{all-0-test}(n)$ directly

$$T^{avg}_{all-0-test}(n) = T^{exp}_{shuffle-all-0-test}(n) = T^{exp}_{rand-all-0-test}(n) \in \Theta(1)$$

Average-case vs. Expected runtime

- Average case runtime and expected runtime are different concepts!

average case	expected
$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{\lvert \mathbb{I}_n \rvert}$	$T^{exp}(I) = \sum_{\text{outcomes } R} T(I,R) \cdot \Pr(R)$
sum is over instances	sum is over random outcomes
	applied only to a randomized algorithm

- There is a relationship only if the randomization of a deterministic algorithm effectively achieves ‘choose the input instance randomly’

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
  - k elements are smaller or equal, $n - 1 - k$ elements are larger or equal
  - select(k) returns $k + 1$ smallest element
  - k is also called **rank**

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

select$(2) = 20$

- Special case: *MedianFinding* = select$\left(k = \left\lfloor \frac{n}{2} \right\rfloor\right)$
- Selection can be done with heaps in $\Theta(n + k \log n)$ time
  - this is $\Theta(n \log n)$ for median finding, not better than sorting
- **Question**: can we do selection in linear time?
  - yes, with *quick-select* (average case analysis)
  - subroutines for *quick-select* also useful for sorting algorithms

Two Crucial Subroutines for *Quick-Select*

- *choose-pivot*(A)
  - return an index p in A
  - $v = A[p]$ is called *pivot value*

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

- *partition* (A, p) uses $v = A[p]$ to rearranges A so that

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

  - items in $A\ [0, \dots, i-1]$ are $\leq v$
  - $A[i] = v$
  - items in $A\ [i+1, \dots, n-1]$ are $\geq v$
  - index i is called *pivot-index* i
    - we have no control over value of i
  - *partition*(A, p) returns *pivot-index* i
    - i is a correct location of v in sorted A
    - v would be the answer if $i = k$

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. $0 \le p < n$

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

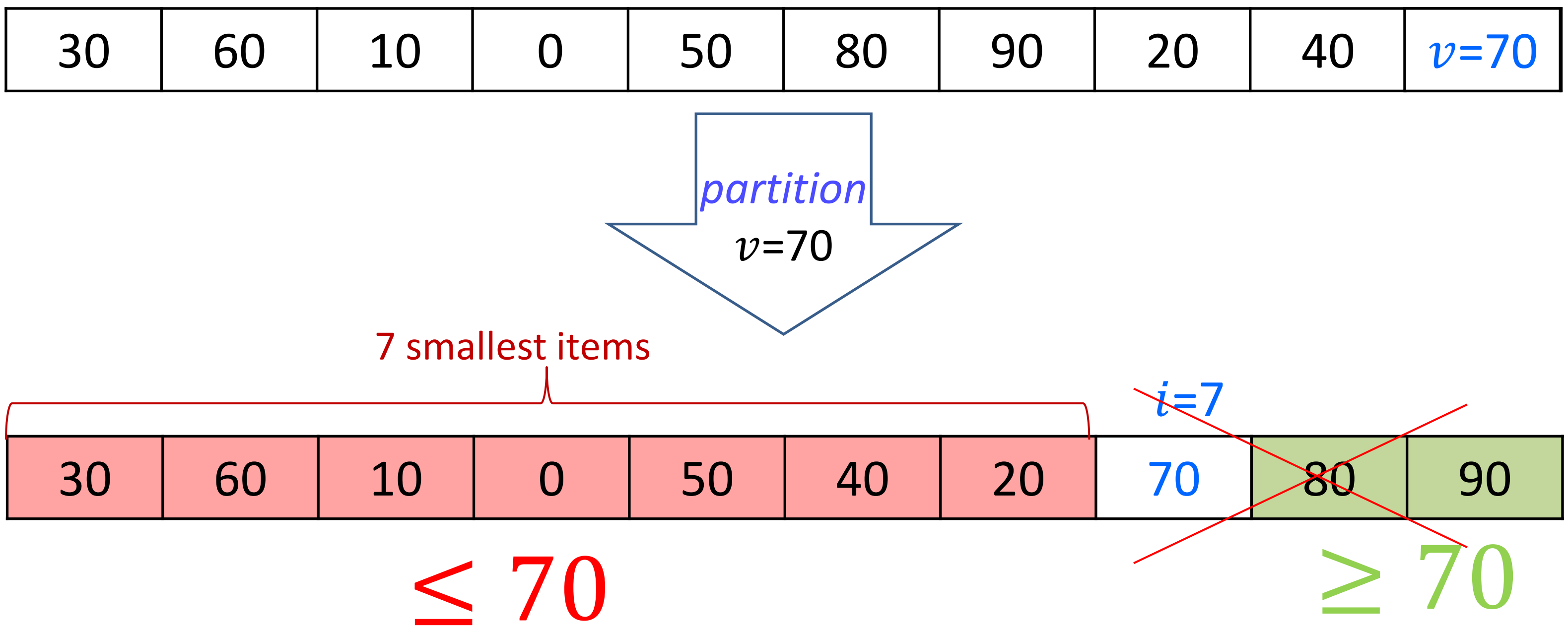
Efficient In-Place partition (Hoare)

```
partition (A, p)
    A: array of size n
    p: integer s.t. 0 ≤ p < n
        swap(A[n − 1], A[p])  // put pivot at the end
        i ← −1,      j ← n − 1,      v ← A[n − 1]
        loop
            do i ← i + 1 while A[i] < v
            do j ← j − 1 while j ≥ i and A[j] > v
            if i ≥ j then break
            else swap(A[i], A[j])
        end loop
        swap(A[n − 1], A[i])   // put pivot in correct position
        return i
```

- Running time is $\Theta(n)$

Quick Select Algorithm

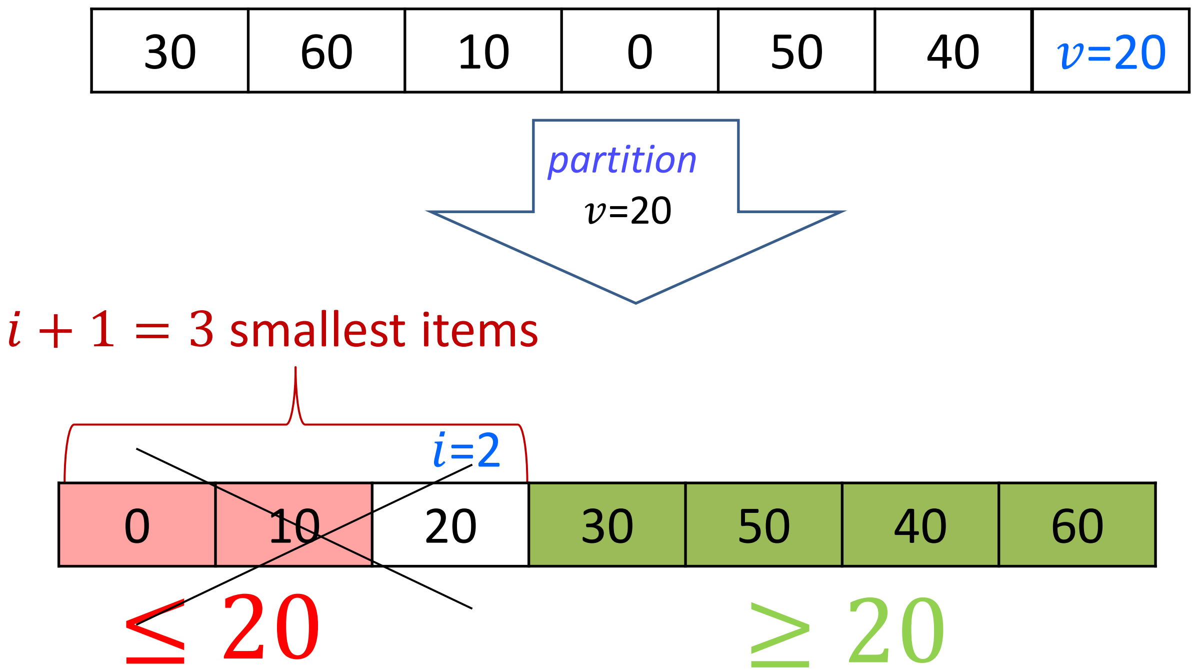
- Find item that would be in $A[k]$ if A was sorted
- Similar to quick-sort, but recurse only on one side ("quick-sort with pruning")
- Example: select($k = 4$)



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

Quick Select Algorithm

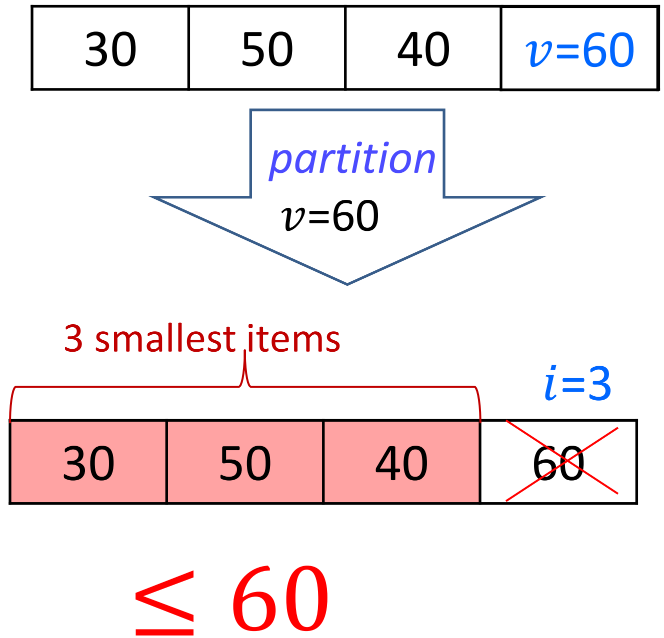
- Example continued: select($k = 4$)



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

Quick Select Algorithm

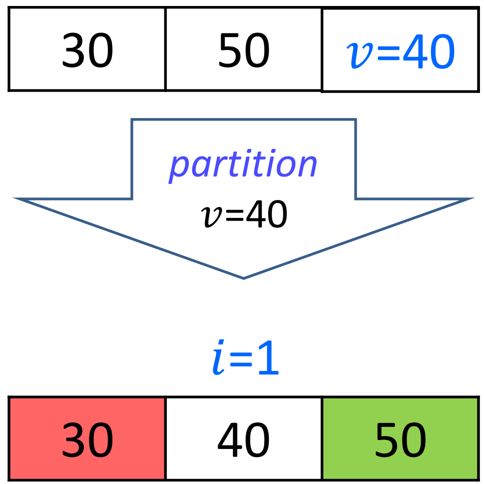
- Example continued: select($k = 1$)



- $i > k$, 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

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) //running time $\Theta(n)$

if $i = k$ **then return** $A[i]$

else if $i > k$ **then return** *QuickSelect*$(A[0, 1, \ldots, i-1],\ k)$

else if $i < k$ **then return** *QuickSelect*$(A[i+1, \ldots, n-1],\ k-(i+1))$

- Let $T(n, k)$ be # of comparisons in array of size n with parameter k
  - this is asymptotically the same as run-time
- **Best case**
  - first chosen pivot could have pivot-index k
  - no recursive calls, total cost $\Theta(n)$
- **Worst case**
  - pivot-value is always the largest and $k = 0$

$$T(n, 0) = \begin{cases} n + T(n-1, 0) & n > 1 \\ 1 & n = 1 \end{cases}$$

  - recurrence equation resolves to $\Theta(n^2)$

Average Case Analysis

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

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

- Observe: *QuickSelect* 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

- For simplicity, will assume array A stores unique numbers
- Characterize input by its sorting permutation $\boldsymbol{\pi}$
  - 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

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

$\pi(0)$ $\pi(1)$ $\pi(2)$ $\pi(6)$

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

$$1 \le 2 \le 3 \le 5 \le 7 \le 11 \le 14$$ sorted!

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

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

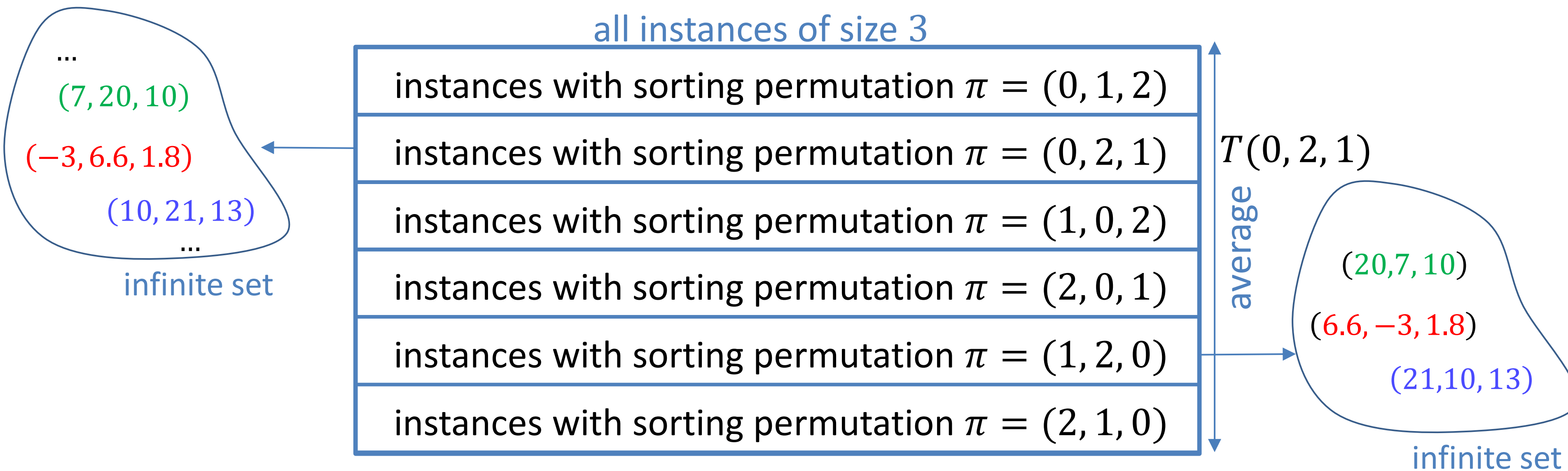
$$\pi = (4, 1, 2, 3, 6, 5, 0)$$

Average Time with Sorting Permutations

- There are $n!$ sorting permutations for arrays with distinct numbers 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 through permutations

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

- Intuitively, since all instances with sorting permutation π have exactly the same running time, we group them together



Average-Case Analysis of *QuickSelect*

- For analyzing average case run-time, we assume all input items are distinct
    - this can be forced by tie-breakers
- Can show (complicated) that average-case runtime is $\Theta(n)$
- Instead, we will randomize *QuickSelect*
    - when randomization is done with shuffling, the expected time of randomized *QuickSelect* is the same as average case runtime of non-randomized *QuickSelect*
    - expected runtime of randomized *QuickSelect* is easier to derive
    - In addition, randomized *QuickSelect* is the fastest algorithm for the selection problem in practice

Randomized QuickSelect: Shuffling

- **First idea** for randomization
- Shuffle the input then run *quickSelect*

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

- Can show that every permutation of A is equally likely after *shuffle*
- As shown before, expected time of *quickSelectShuffled* is the same as average case time of *quickSelect*

Randomized QuickSelect Algorithm

- **Second idea**: change pivot selection

RandomizedQuickSelect(A, k)

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

$p \leftarrow$ *random*$(A.size)$

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

if $i = k$ **then return** $A[i]$

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

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

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

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

- Just one line change from *QuickSelect*
- It is possible to prove that *RandomizedQuickSelect* has the same expected runtime as *quickSelectShuffled* (no details)
- Therefore expected time for *RandomizedQuickSelect* is the same as the average case runtime of *QuickSelect*
  - easier to compute

Randomized QuickSelect: Analysis

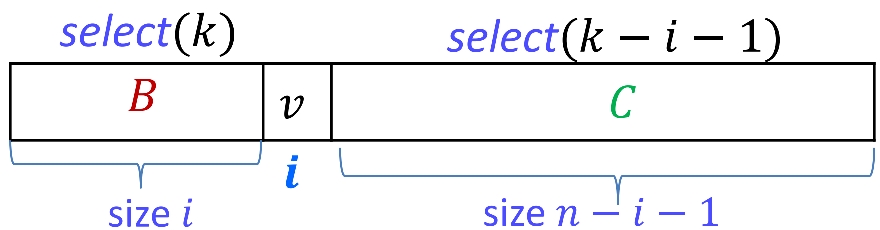
RandomizedQuickSelect(A, k)

$p \leftarrow$ *random*$(A.size)$

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

…

- Let $T(A, k, R)$ be number of *key-comparisons* on array A of size n, selecting kth element, using random numbers R
    - asymptotically the same as running time
- Identify numbers p generated by *random* with pivot indexes i
    - one-one correspondence between generated numbers and pivot indexes
- So R is a sequence of randomly generated pivot indexes, $R = \langle$first, the rest of $R\rangle = \langle i, R' \rangle$
- Assume array elements are distinct
    - probability of any pivot-index i equal to $1/n$
- Structure of array A after partition



- Recurse in array B or C or algorithms stops

$$T(A, k, \langle i, R' \rangle) = n + \begin{cases} T(B, k, R') & \text{if } i > k \\ T(C, k - i - 1, R') & \text{if } i < k \\ 0 & \text{otherwise} \end{cases}$$

Randomized QuickSelect: Analysis

- Runtime of $RandomizedQuickSelect(A, k)$ also depends on k

$$T^{exp}(n) = \max_{A \in \mathbb{I}_n} \max_{k \in \{0, \ldots n-1\}} \sum_{R} T(A, k, R) \Pr(R)$$

Randomized QuickSelect: Analysis

$$T(A,k,\langle i,R'\rangle) = n + \begin{cases} T(B,k,R') & \text{if } i > k \\ T(C,k-i-1,R') & \text{if } i < k \\ 0 & \text{otherwise} \end{cases}$$

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

$$= \sum_{R=\langle i,R'\rangle} T(A,k,\langle i,R'\rangle)\Pr(i)\Pr(R')$$

$$= \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} T(A,k,\langle i,R'\rangle)\Pr(R') \quad + \frac{1}{n}\cdot n \quad + \frac{1}{n}\sum_{i=k+1}^{n-1}\sum_{R'} T(A,k,\langle i,R'\rangle)\Pr(R')$$

$i < k$: recurse on C base case $i > k$: recurse on B

$$= \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'}[n + T(C,k-i-1,R')]\Pr(R') + 1 + \frac{1}{n}\sum_{i=k+1}^{n-1}\sum_{R'}[n + T(B,k,R')]\Pr(R')$$

$$\le n \ + \frac{1}{n}\sum_{i=0}^{k-1} \max_{D\in\mathbb{I}_{n-i-1},\, w\in\{0,\ldots k-1\}} \sum_{R'} T(D,w,R')\Pr(R') \ + \frac{1}{n}\sum_{i=k+1}^{n-1} \max_{D\in\mathbb{I}_{i},\, w\in\{k+1,\ldots,n-1\}} \sum_{R'} T(D,w,R')\Pr(R')$$

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

- Since above bound works for any A and k, it will work for the worst A and k

$$T^{exp}(n) = \max_{A\in\mathbb{I}_n} \max_{k\in\{0,\ldots n-1\}} \sum_R T(A,k,R)\Pr(R) \le n + \frac{1}{n}\sum_{i=0}^{n-1} \max\{T^{exp}(i), T^{exp}(n-i-1)\}$$

Randomized QuickSelect: Analysis

- In summary, expected runtime for *RandomizedQuickSelect*

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

Randomized QuickSelect: Solving Recurrence

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

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

Proof:

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

induction hypothesis applies to each one of these

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

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

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

Randomized QuickSelect: Solving Recurrence

exactly what we need for the proof

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

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

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

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

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

$$\left(\frac{3n}{2}-1\right)\frac{n}{4} \qquad \left(\frac{3n}{2}-1\right)\frac{n}{4}$$

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

Summary of Selection

Expected runtime of *RandomizedQuickSelect* is $\Theta(n)$

- the bound is tight since partition takes $\Omega(n)$
- if unlucky with random numbers, then runtime is $\Omega(n^2)$
  - worst case: worst instance, worst luck

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

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
- Also used them to *sort* based on partitioning

QuickSort(A)

Input: array *A* of size n

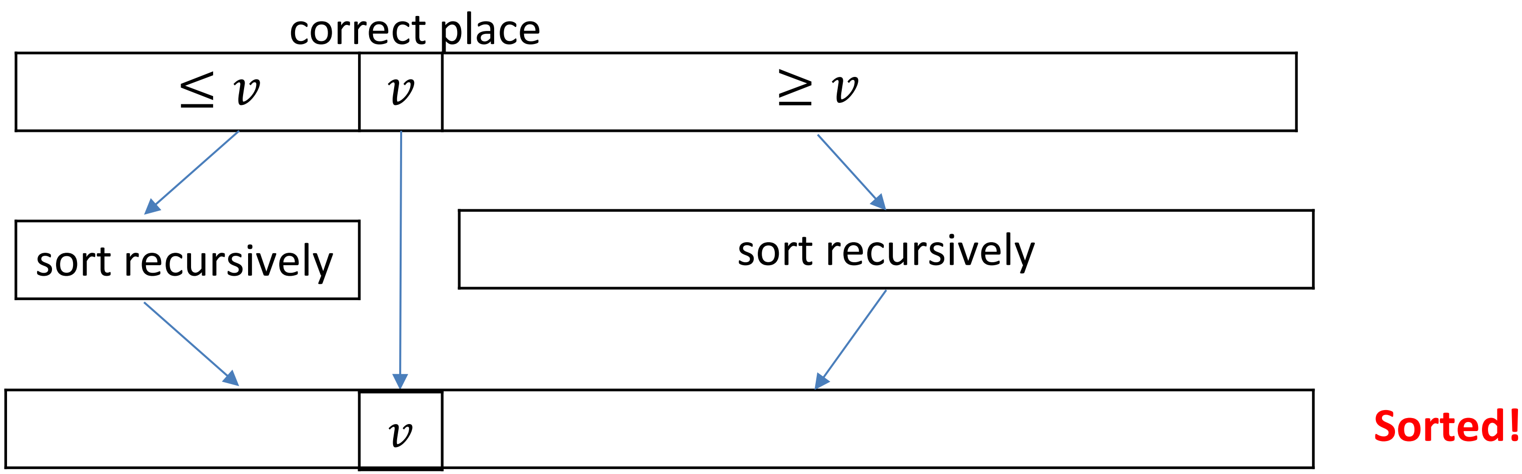
if $n \leq 1$ **then return**

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

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

QuickSort$(A[0, 1, \ldots, i-1])$

QuickSort$(A[i+1, \ldots, n-1])$



QuickSort

QuickSort(A)

Input: array *A* of size n

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

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

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

QuickSort$(A[0, 1, \dots, i-1])$

QuickSort$(A[i+1, \dots, n-1])$

- Let $T(n)$ to be the number of comparisons on size n array
  - running time is Θ(number of comparisons)
- Recurrence for pivot-index i: $T(n) = n + T(i) + T(n-i-1)$
- Worst case $T(n) = T(n-1) + n$
  - recurrence solved in the same way as *quickSelect*, $O(n^2)$
- Best case $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$
  - solved in the same way as *mergeSort*, $\Theta(n \log n)$
- Average case?
  - through randomized version of *QuickSort*

Randomized QuickSort: Random Pivot

```
RandomizedQuickSort(A)
   Input: array A of size n
         if n ≤ 1  then return
         p ← random(A.size)
         i ← partition (A ,p)
         RandomizedQuickSort(A[0, 1, … , i − 1])
         RandomizedQuickSort(A[i + 1, … , n − 1])
```

- Let $T^{exp}(n) =$ number of comparisons
- Analysis is similar to that of *RandomizedQuickSelect*
  - but recurse both in array of size i and array of size $n - i - 1$
- *Expected running time for RandomizedQuickSort*
  - derived similarly to *RandomizedQuickSelect*

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

Randomized QuickSort: Expected Runtime

- Simpler recursive expression for $T^{exp}(n)$

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

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

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

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

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

Randomized QuickSort: Solve Recurrence Relation

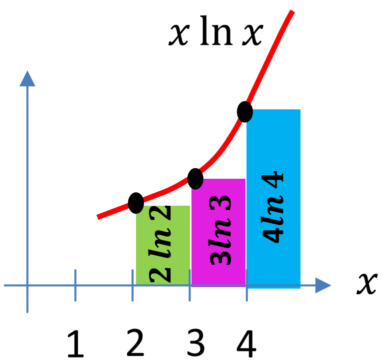
$$T(1) = 0 \text{ and } T(n) \le n + \frac{2}{n}\sum_{i=2}^{n-1} T(i)$$

- Claim $T(n) \le 2n \ln n$ for all $n > 0$
- Proof (by induction on n):
  - $T(1) = 0$ (no comparisons)
  - Suppose true for $2 \le m < n$
  - Let $n \ge 2$

induction hypothesis

$$T(n) \le n + \frac{2}{n}\sum_{i=2}^{n-1} T(i) \le n + \frac{2}{n}\sum_{i=2}^{n-1} 2i \ln i = n + \frac{4}{n}\sum_{i=2}^{n-1} i \ln i$$

- Upper bound by integral, since is $x \ln x$ is monotonically increasing for $x > 1$



$$\sum_{i=2}^{n-1} i \ln i \le \int_2^n x \ln x \, dx = \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 \underbrace{- 2\ln 2 + 1}_{\le 0}$$

$$\le \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2$$

Randomized QuickSort: Solve Recurrence Relation

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

- Claim $T(n) \le 2n \ln n$ for all $n > 0$
- Proof (by induction on n):
  - $T(1) = 0$ (no comparisons)
  - Suppose true for $2 \le m < n$
  - Let $n \ge 2$

induction hypothesis

$$\le \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2$$

$$T(n) \le n + \frac{2}{n}\sum_{i=2}^{n-1} T(i) \le n + \frac{2}{n}\sum_{i=2}^{n-1} 2i \ln i = n + \frac{4}{n}\sum_{i=2}^{n-1} i \ln i$$

$$T(n) \le n + \frac{4}{n}\left(\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2\right) = 2n \ln n$$

- Expected running time of *RandomizedQuickSort* is $O(n \log n)$
  - This is tight since best-case run-time is $\Omega(n \log n)$
- Average case runtime of *QuickSort* 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) ← S.pop()              // get the next subproblem
        while r − l + 1 > 10          // work on it if it's larger than 10
            p ← choose-pivot(A, l, r)
            i ← partition (A, l, r, p)
            if i − l > r − i do       // is left side larger than right?
                S.push((l, i − 1))    // store larger  problem in S for later
                l ← i + 1             // next work on the right side
            else
                S.push((i + 1, r))    // store larger problem in S for later
                r ← 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

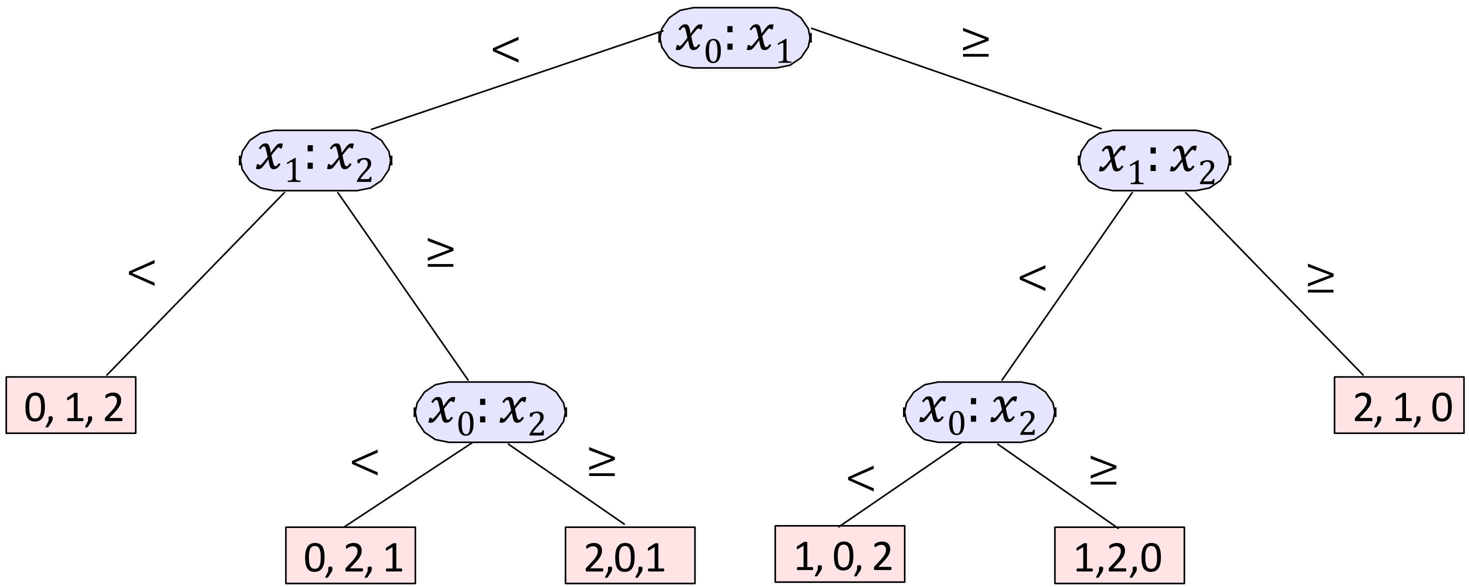
The Comparison Model

- 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
- 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 decisions that are taken during the execution of an algorithm and the resulting outcome
- For each comparison-based sorting algorithm we can construct a corresponding decision tree
- Given decision tree, we can deduce the algorithm
- Can create decision trees for any comparison-based algorithm, not just sorting

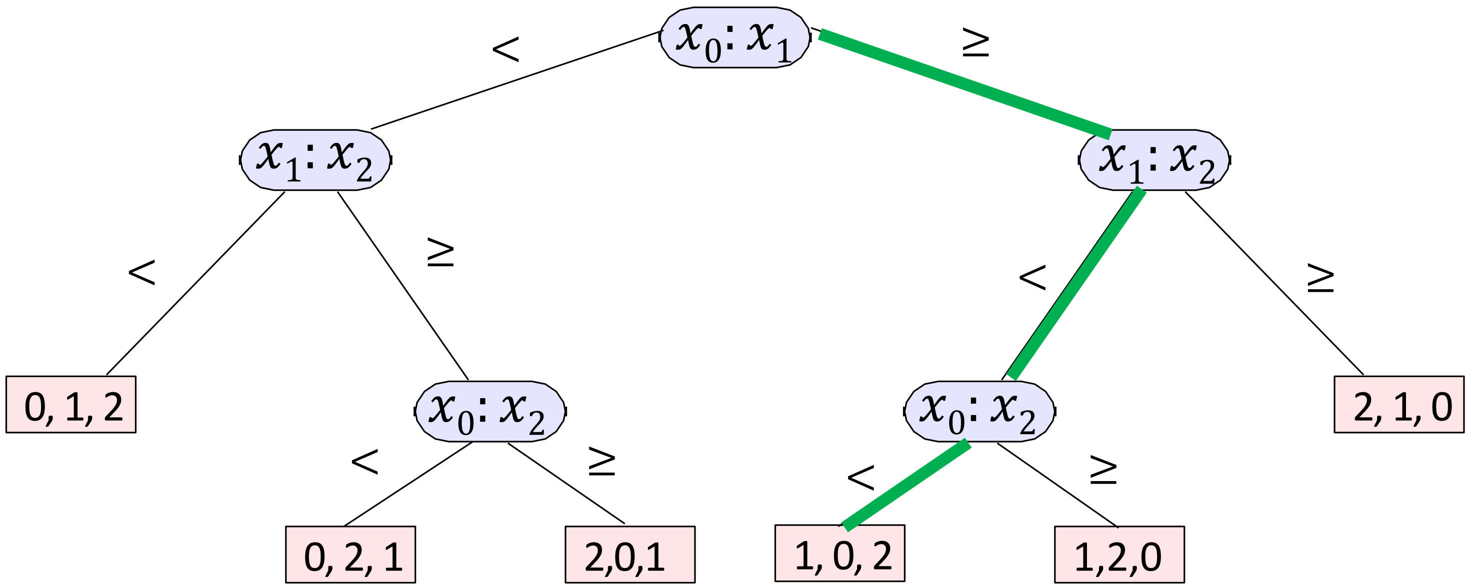
Decision Tree for Concrete Algorithm Sorting 3 items



if $x_0 < x_1$ **then**
 if $x_1 < x_2$ **then** print(x_0, x_1, x_2)
 else if $x_0 < x_2$ **then** print(x_0, x_2, x_1)
 else print(x_2, x_0, x_1)
else
 if $x_1 < x_2$ **then**
  if $x_0 < x_2$ **then** print(x_1, x_0, x_2)
  else print(x_1, x_2, x_0)
 else print(x_2, x_1, x_0)

Decision Tree: Sorting Example

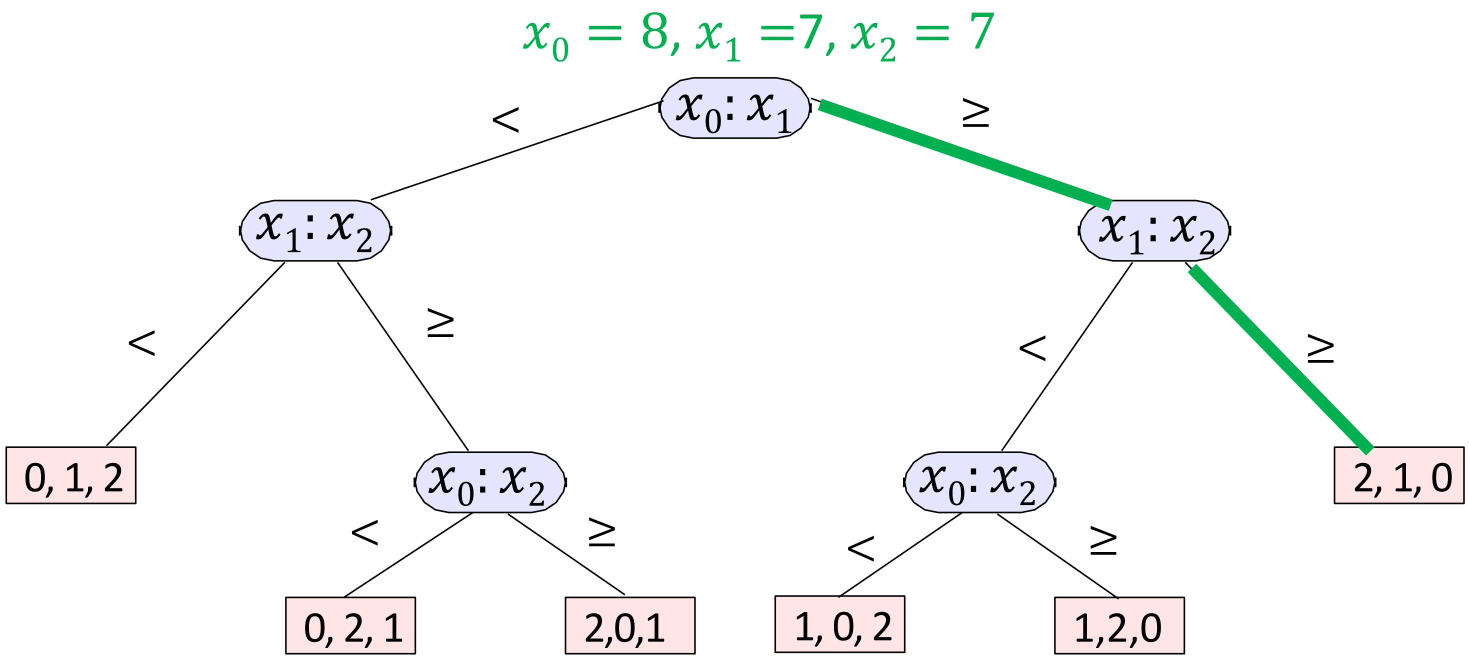
$x_0 = 4, x_1 = 2, x_2 = 7$



$x_1 = 2 \leq x_0 = 4 \leq x_2 = 7$

3 comparisons

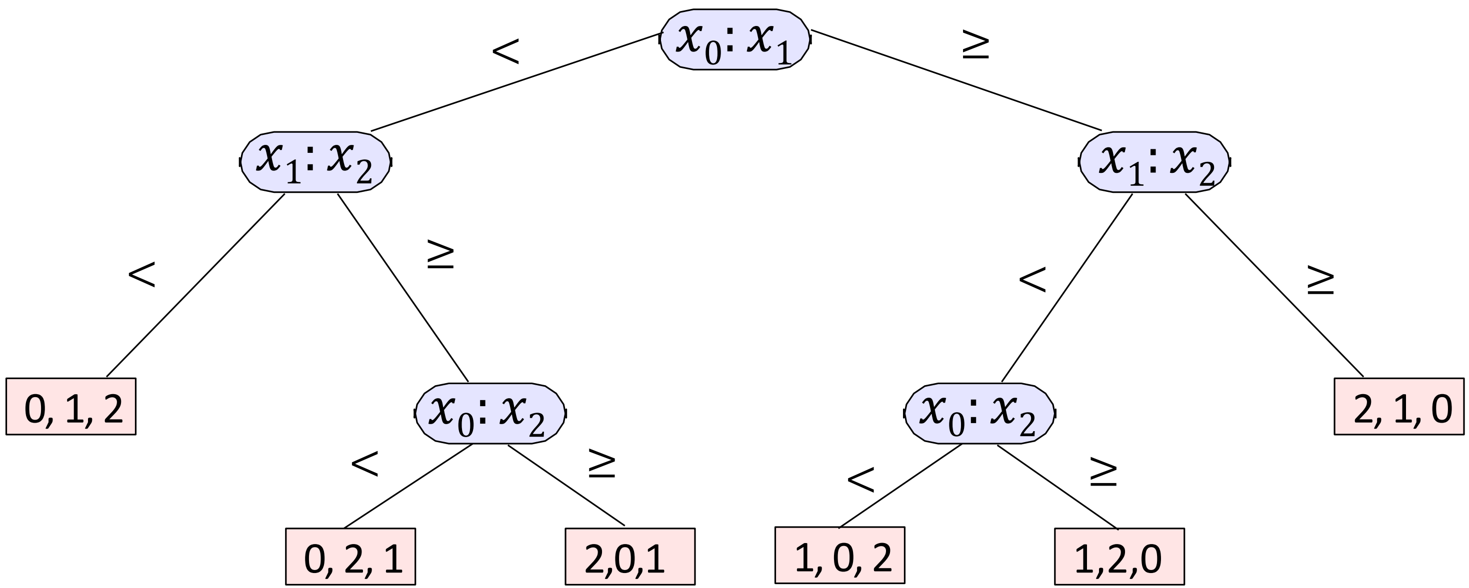
Decision Tree: Sorting Example



$x_2 = 7 \leq x_1 = 7 \leq x_0 = 8$

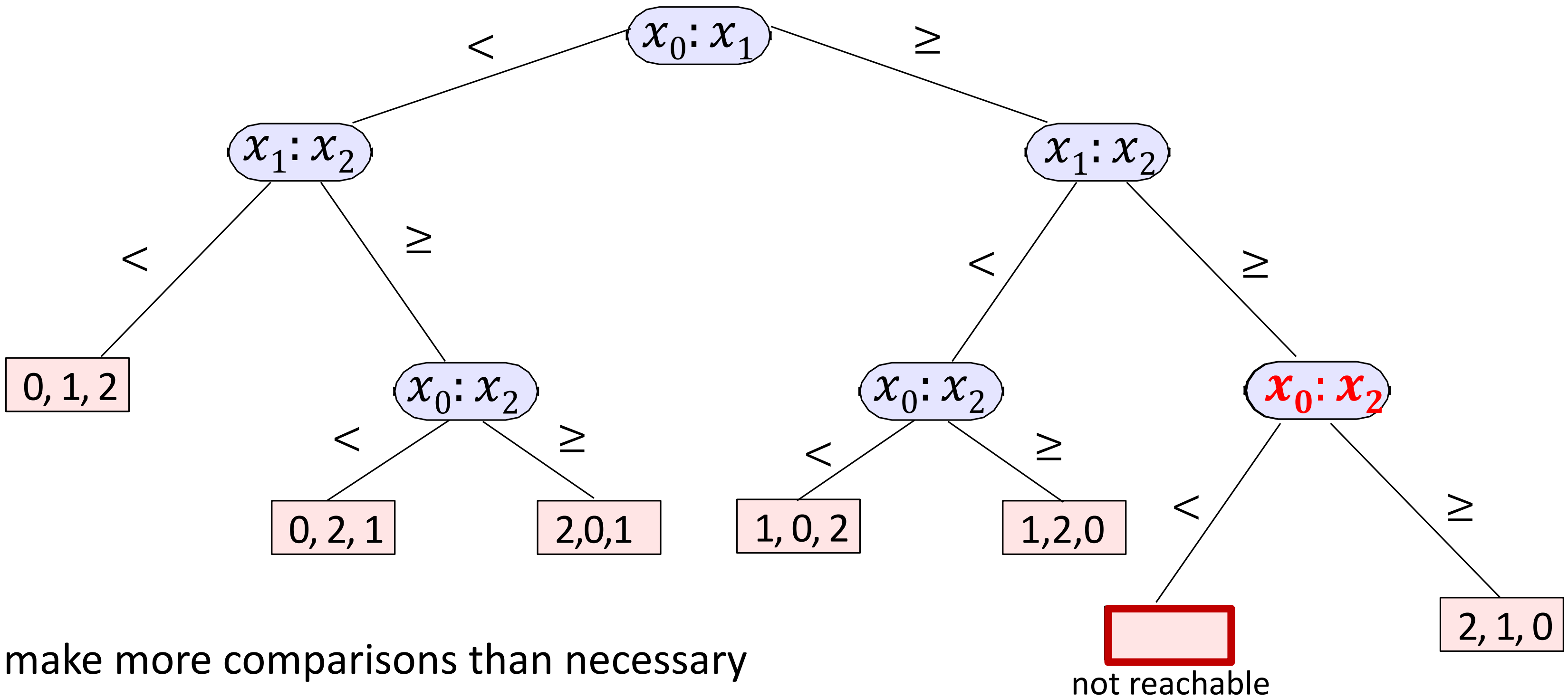
2 comparisons

Decision Tree



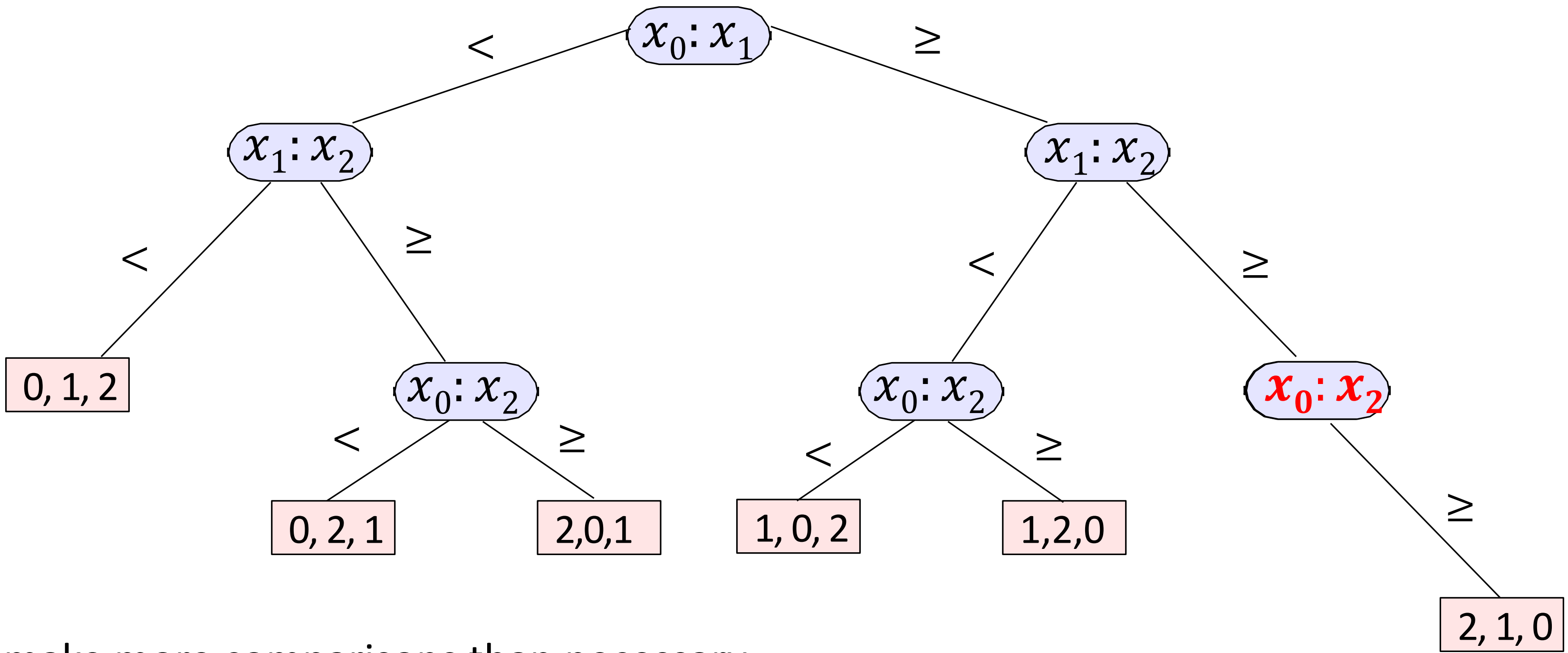
- Interior nodes are comparisons
    - root corresponds is the first comparison
- Each comparison has two outcomes: $<$ and $\geq$
- Each interior node has two children, links to the children are labeled with outcomes
- When algorithm makes no more comparisons, that node becomes a leaf
    - *sorting permutation* has been determined once we reach a leaf
    - label the leaf with the corresponding sorting permutation, if reachable

Decision Tree



- Can make more comparisons than necessary
- Can have leaves which are never reached
- Can have unreachable branches
- Unreachable branches/leaves make no difference for the runtime
  - algorithm never goes into unreachable structure
- So assume everything is reachable (i.e. prune unreachable branches from decision tree)

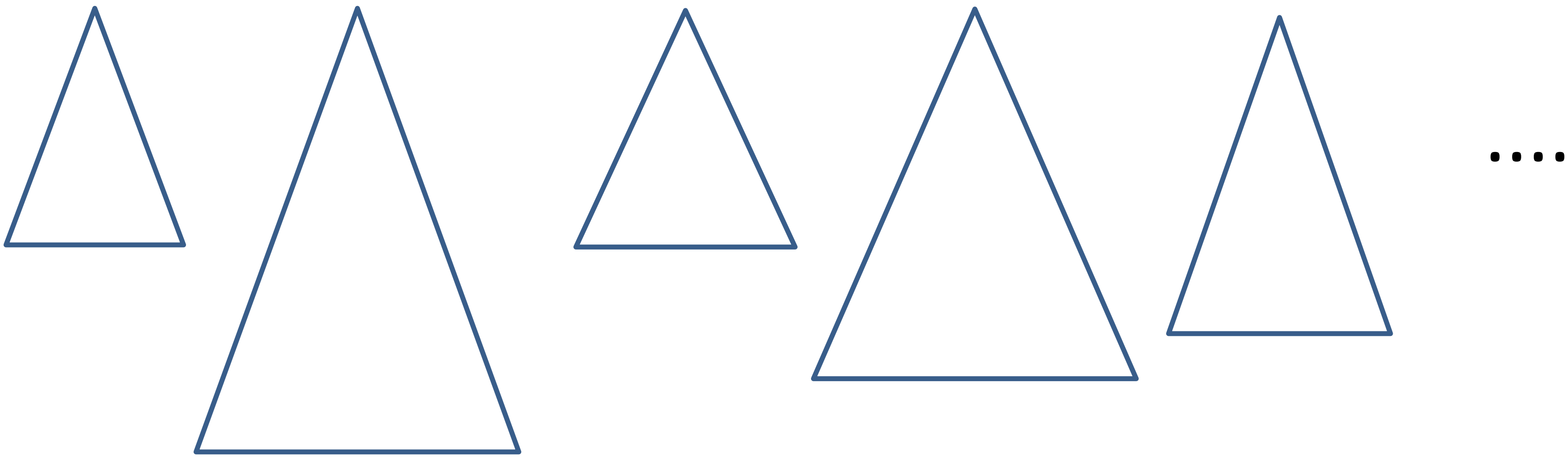
Decision Tree



- Can make more comparisons than necessary
- Can have leaves which are never reached
- Can have unreachable branches
- Unreachable branches/leaves make no difference for the runtime
  - algorithm never goes into unreachable structure
- So assume everything is reachable (i.e. prune unreachable branches from decision tree)
- Tree height h is the worst case number of comparisons

Decision Tree

- General case: comparison-based sort for n elements
- Many sorting algorithms, for each one we have its own decision tree



- Can prove that the height of ***any*** decision tree is at least $cn\log n$
    - which is $\Omega(n\log n)$

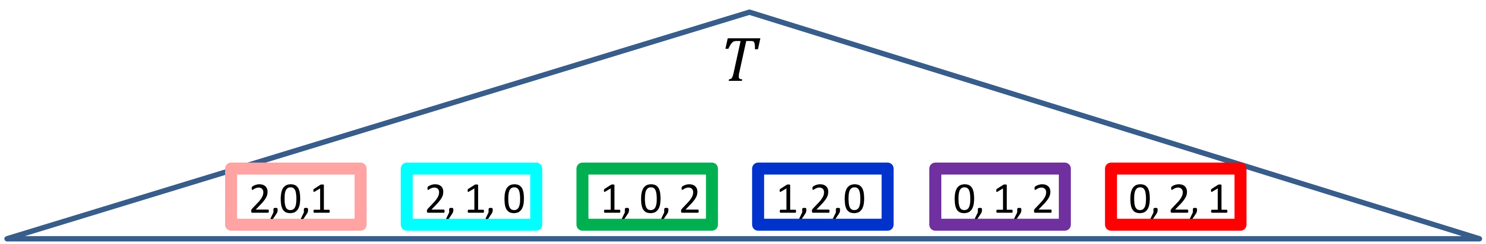
Lower bound for sorting in the comparison model

Theorem: Comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons

Proof:

- Let *SortAlg* be ***any*** comparison based sorting algorithm
- Since *SortAlg* is comparison based, it has a decision tree

$$S_3 = \{[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]\}$$



- *SortAlg* must sort correctly *any* array of n elements
- Let S_n = set of arrays storing not-repeating integers $1, \dots, n$
- $|S_n| = n!$
- Let π_x denote the sorting permutation of $x \in S_n$
- When we run x through T, we **must** end up at a leaf labeled with π_x
- $x, y \in S_n$ with $x \neq y$ have sorting permutations $\pi_x \neq \pi_y$
- $n!$ instances in S_n must go to distinct leaves $\Rightarrow$ **tree must have at least $n!$ leaves**

Lower bound for sorting in the comparison model

Proof: (cont.)

- Therefore, the 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

$$h \geq \log(n!) = \log(n(n-1)\ldots\cdot 1) = \underbrace{\log n + \cdots + \log(\tfrac{n}{2}+1)}_{> \log\frac{n}{2}} + \log\frac{n}{2} + \cdots + \log 1$$

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

□

- Notes about the proof
  - proof does not assume the algorithm sorts only distinct elements
  - proof does not assume the algorithms sorts only integers in range $\{1, \ldots, n\}$
  - poof is based on finding $n!$ input instances that must go to distinct leaves
    - total number of inputs is infinite

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

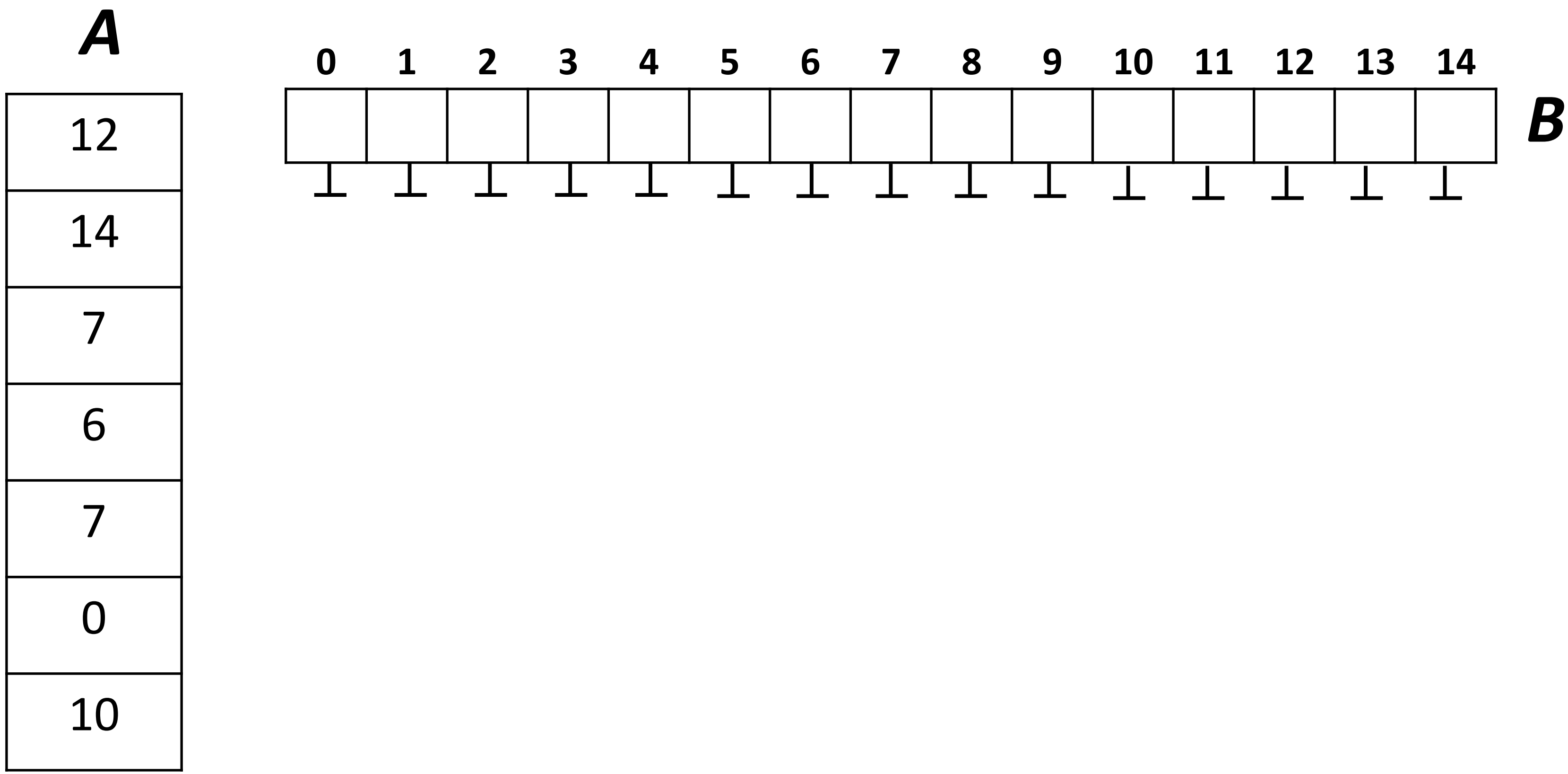
- Sort without comparing items to each other
- **Non-comparison based sorting is less general than comparison based sorting**
- In particular, need to make assumptions about items we sort
  - unlike in comparison based sorting, which sorts any data, as long as it can be compared
- Will assume we are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

- Suppose all keys in A of size n are integers in range $[0, \ldots, L-1]$
- How would you sort if L is not too large?
  - say $L < n$

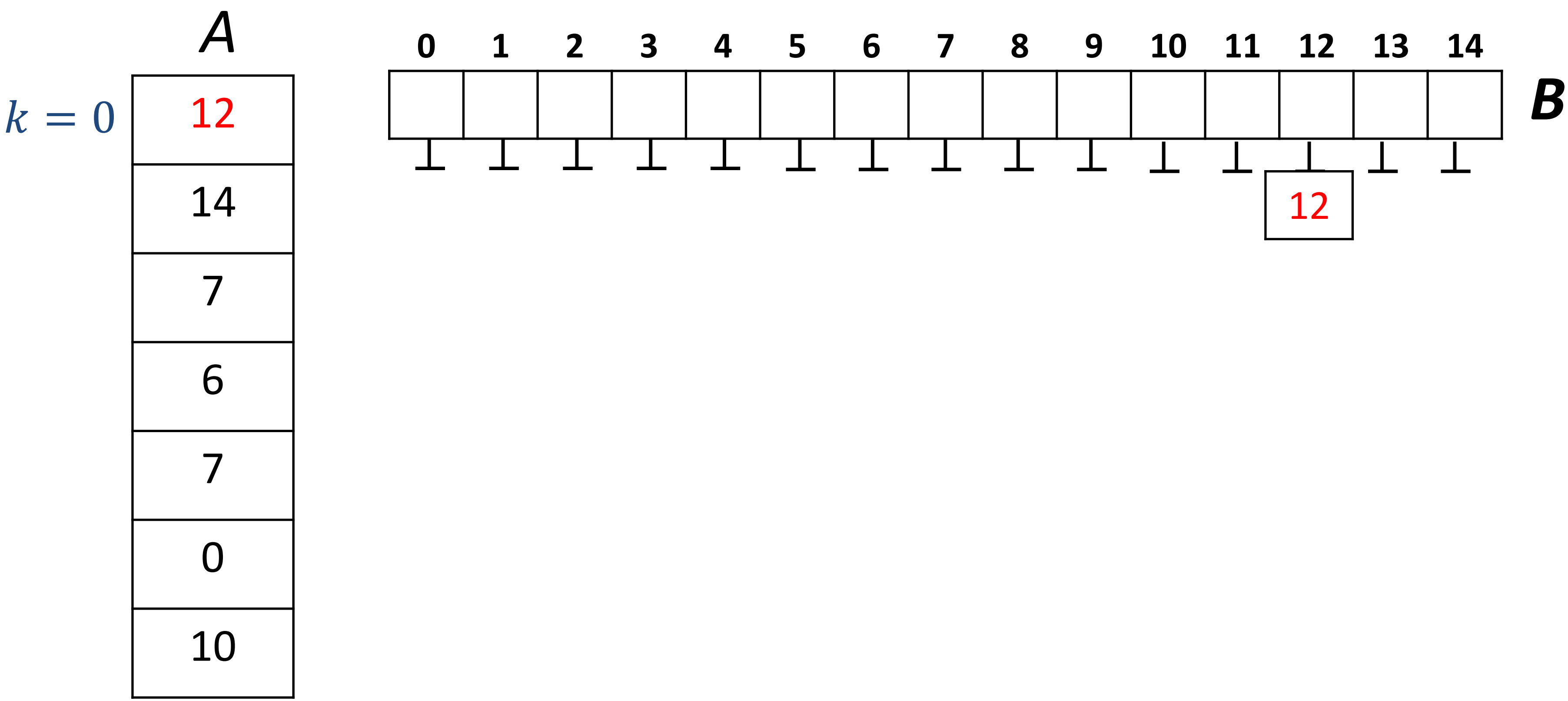
Bucket Sort

- Suppose all keys in A of size n are integers in range $[0, \dots, L-1]$
- How would you sort if L is not too large?
- Use an axillary *bucket array* $B[0, \dots, L-1]$ to sort
  - i.e. array of initially empty linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



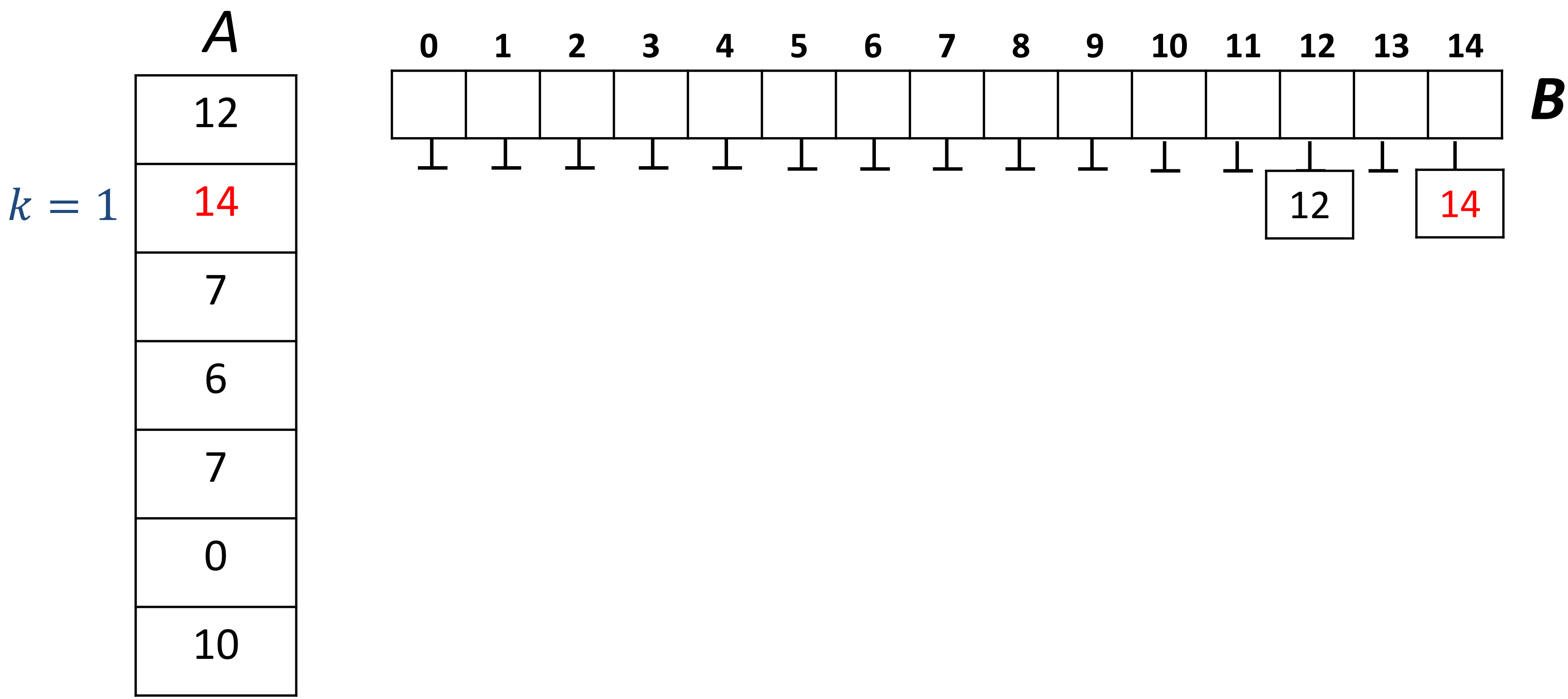
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



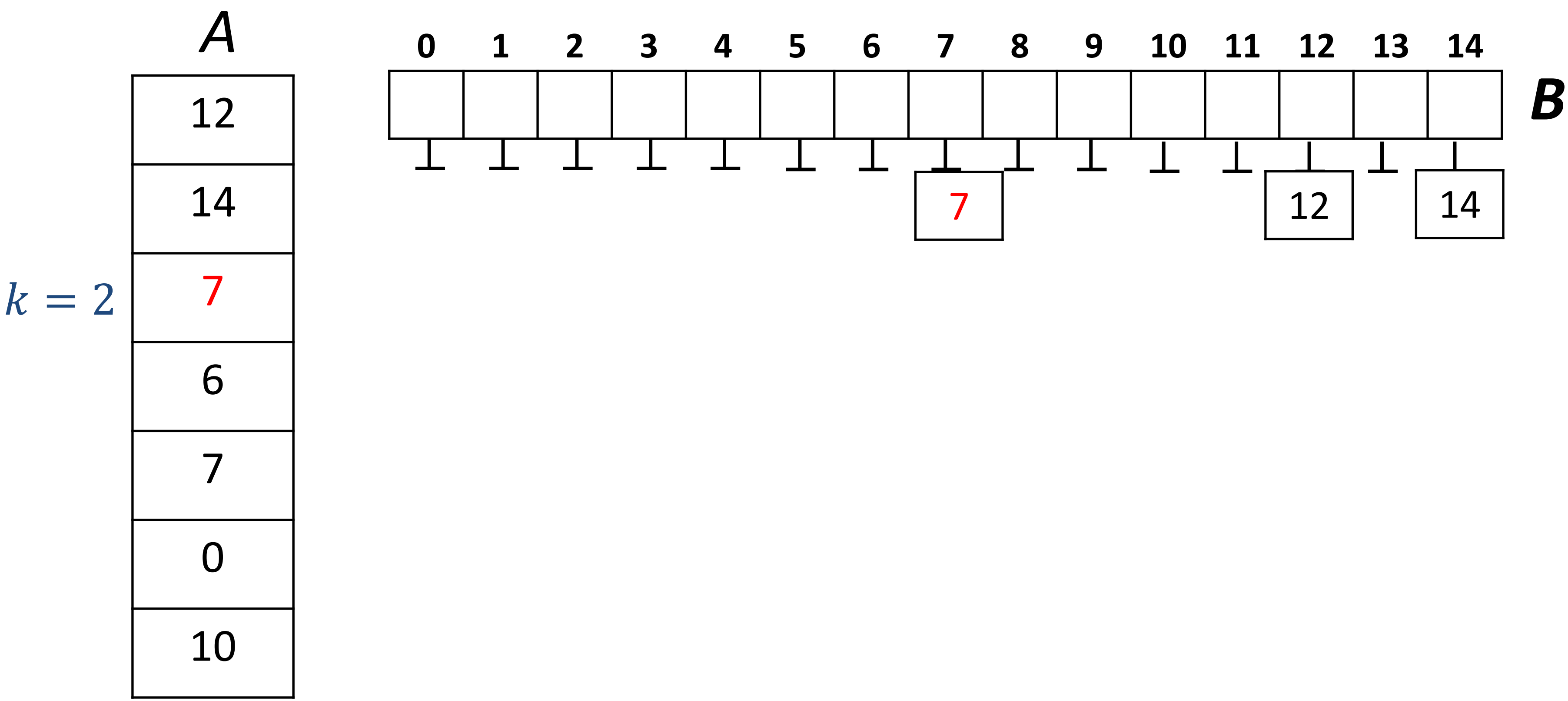
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



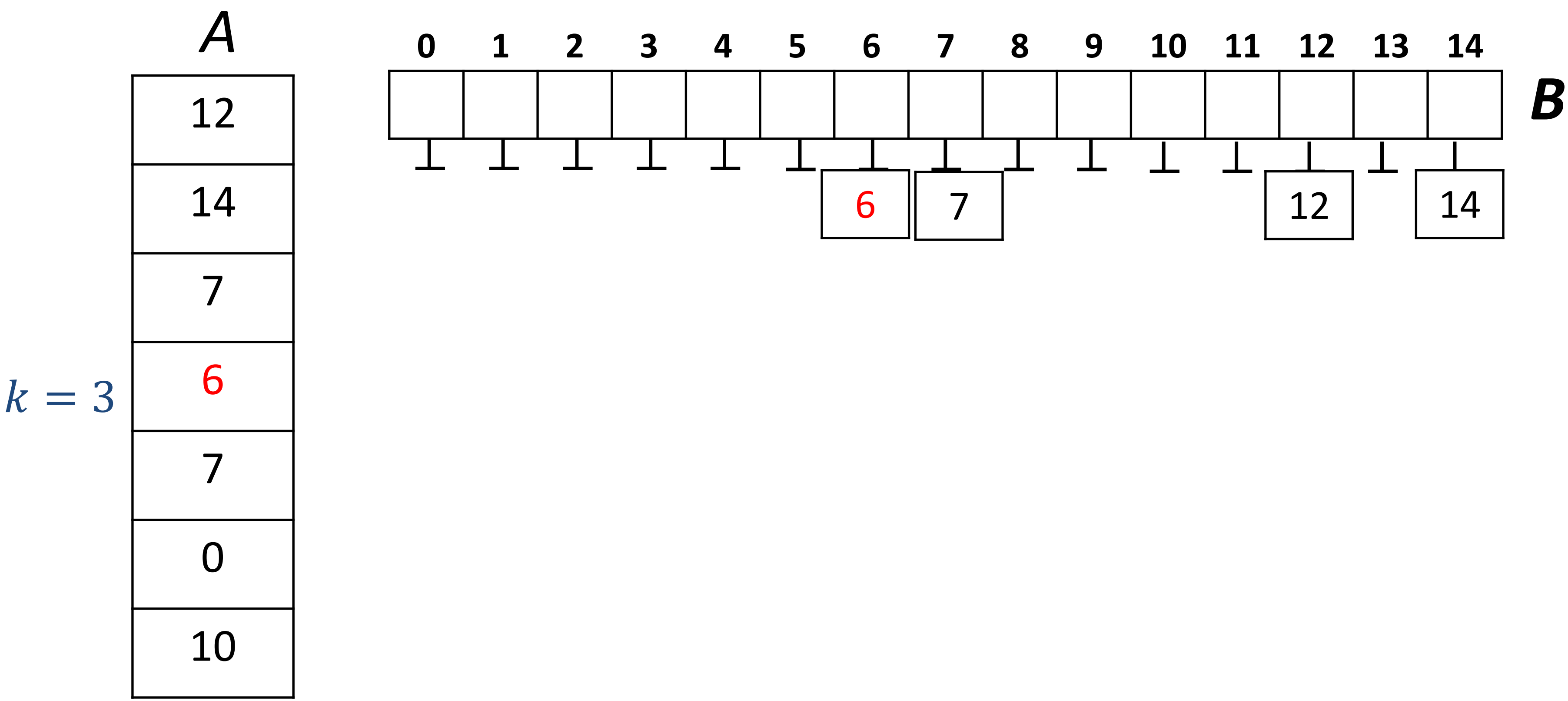
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



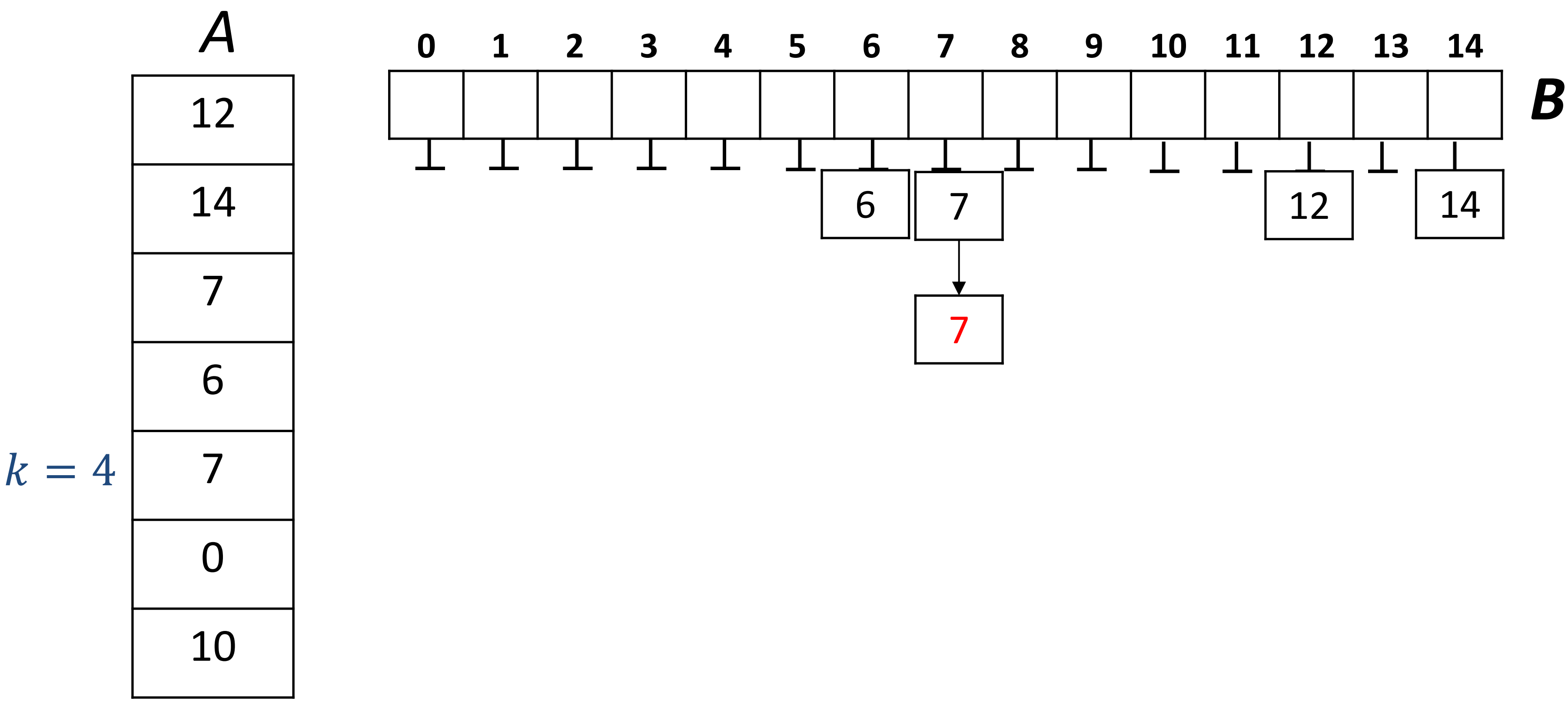
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



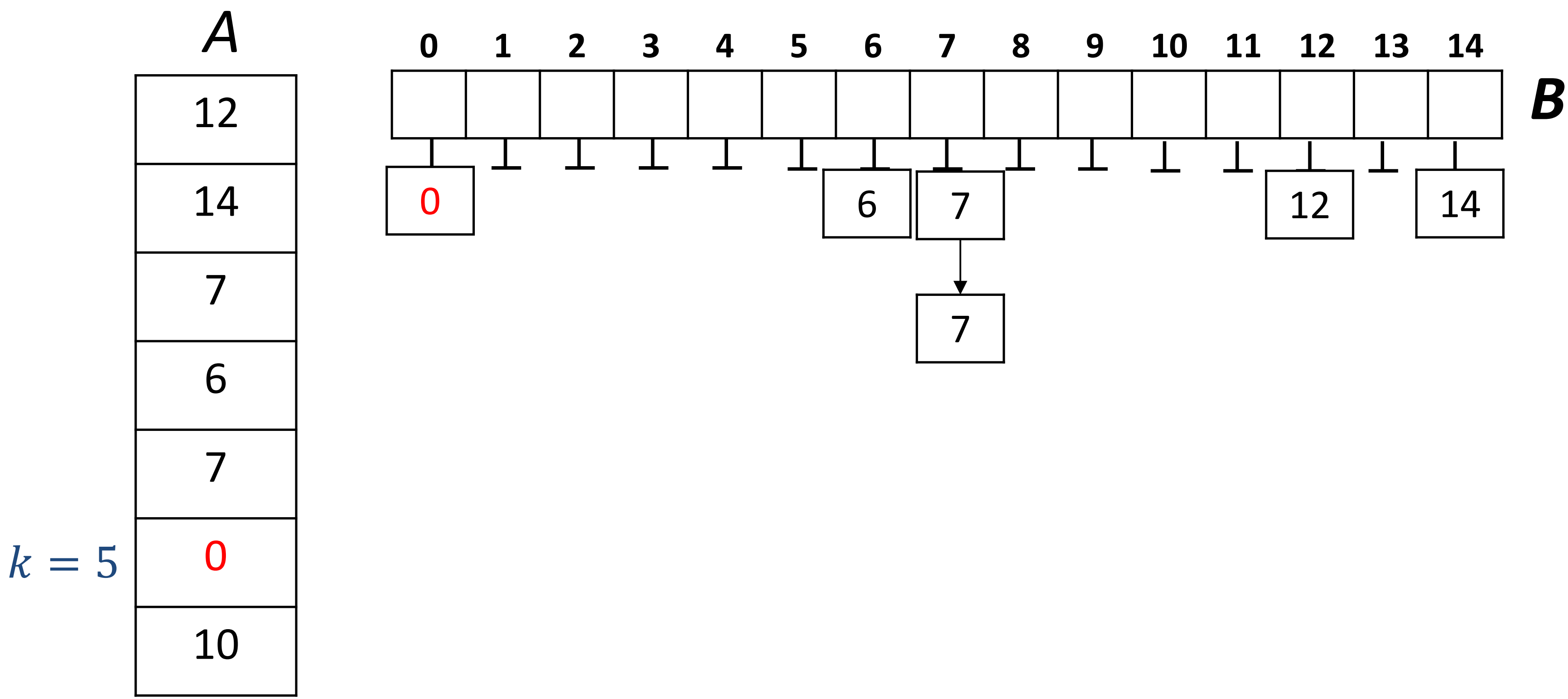
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



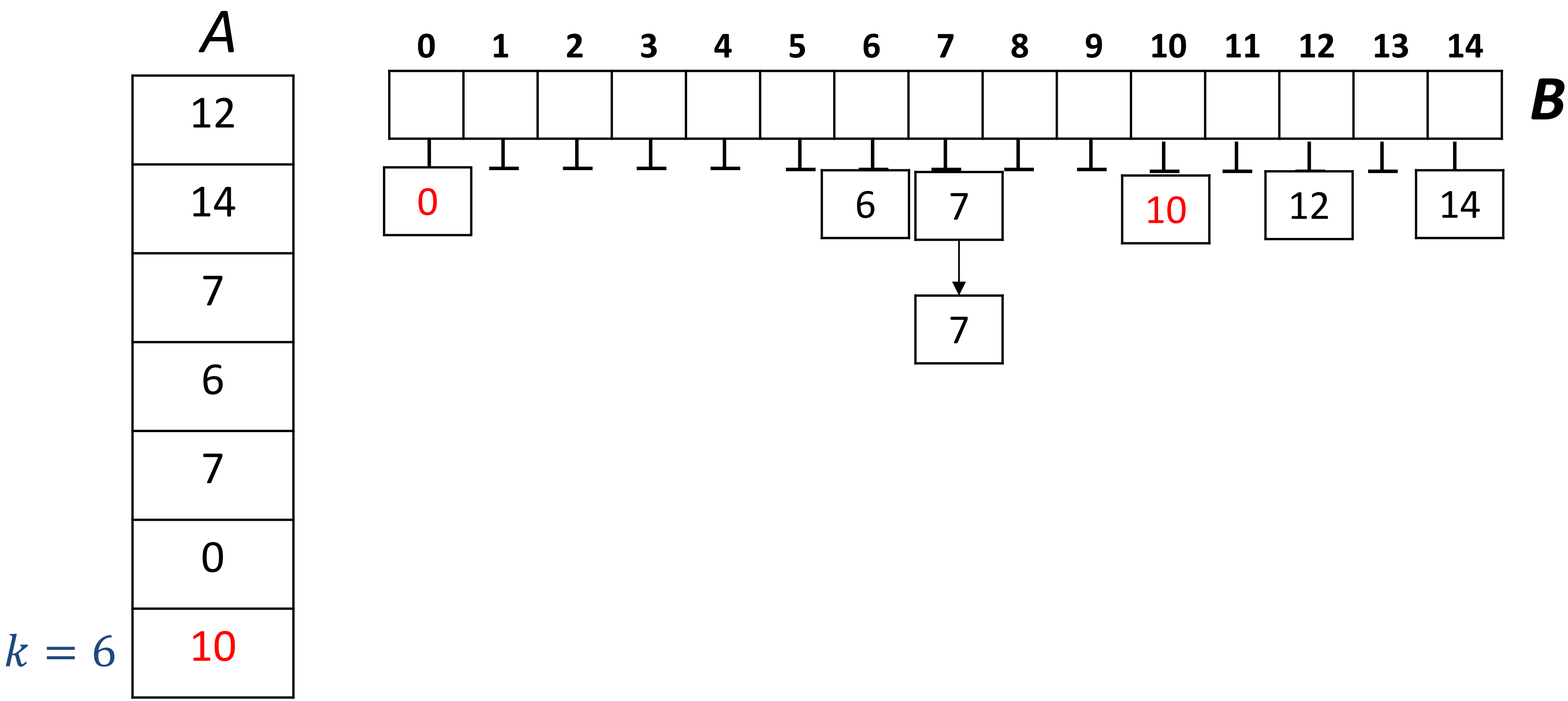
Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$



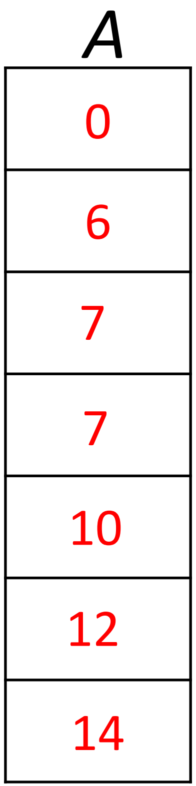
Bucket Sort

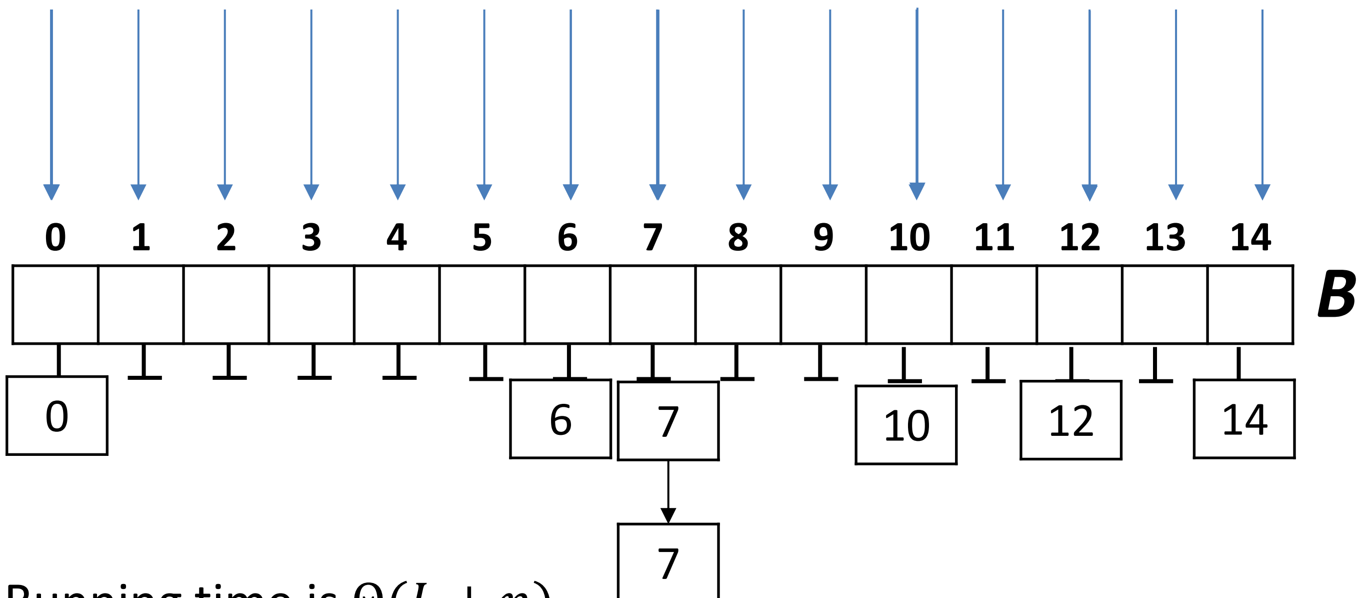
- Suppose all keys in A of size n are integers in range $[0, \dots, L-1]$
- Use an axillary *bucket array* $B[0, \dots, L-1]$ to sort
    - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$



Bucket Sort

- Suppose all keys in A of size n 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 $L = 15$
- Now iterate through B and copy non-empty buckets to A

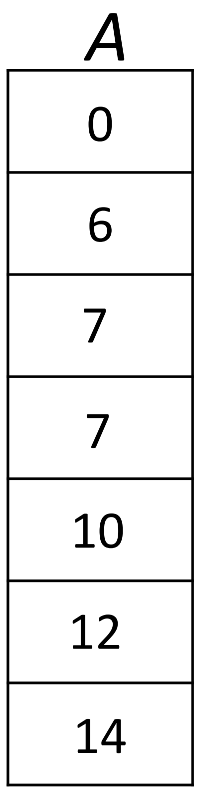


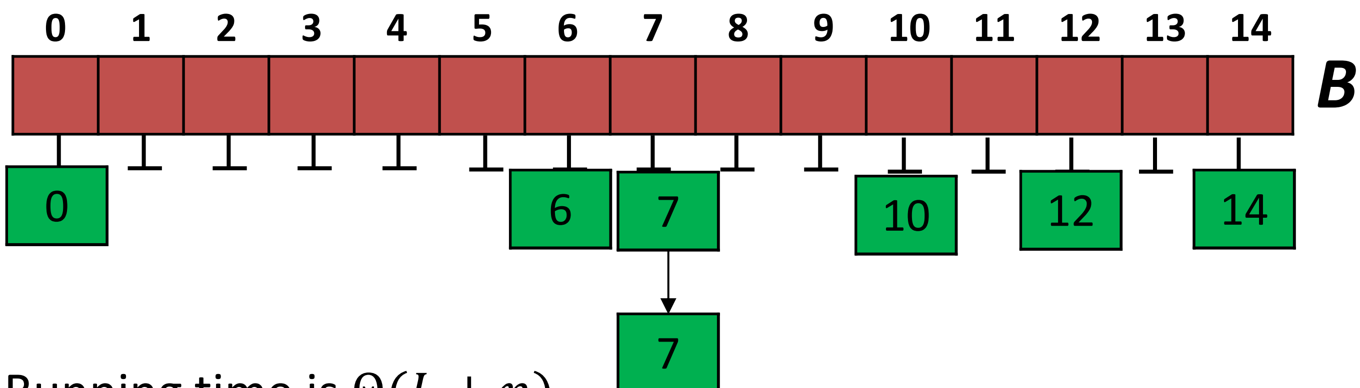


- Running time is $\Theta(L + n)$
    - runtime depends on **both** n and L
- Auxiliary space is $\Theta(L + n)$

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





- Running time is $\Theta(L + n)$
  - n is size of A, runtime depends on both n and L
- Auxiliary space is $\Theta(\boldsymbol{L} + n)$

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 keys have length of m digits
  - pad with leading 0s to get keys of equal length m

123	230	021	320	210	232	101

- Can sort 'digit by digit'

123
230
021
320

$1 \rightarrow m$

MSD-Radix-Sort: forward

123
230
021
320

$1 \leftarrow m$

LSD-Radix-Sort: backward

- Bucketsort is perfect for sorting 'by digit'

Base R number representation

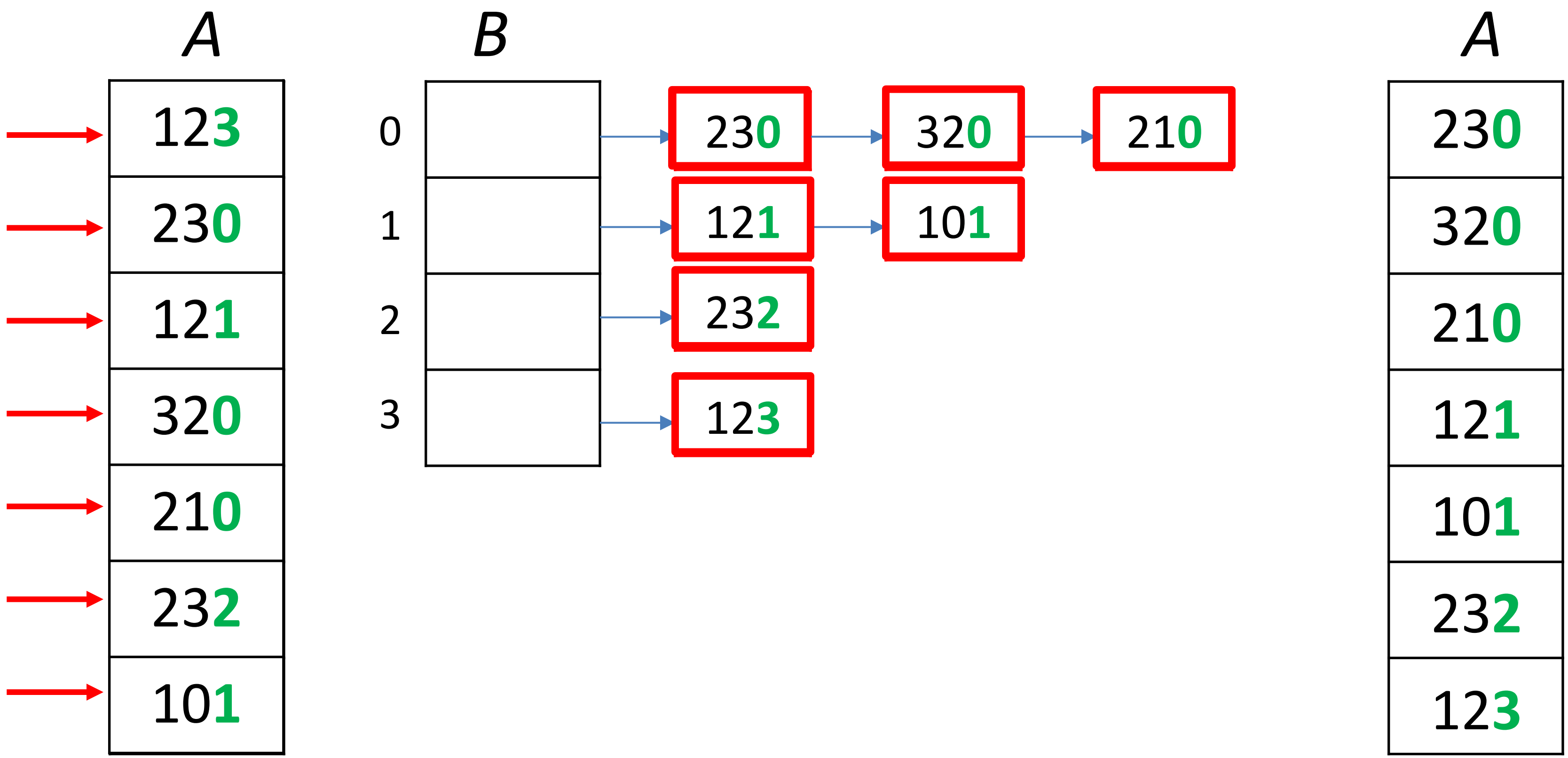
- Can represent numbers in any base R representation
    - digits go from 0 to $R - 1$
    - R buckets
    - numbers are in the range $\{0, 1, \dots, R^m - 1\}$
- Number of distinct digits gives the number of buckets R
- Useful to control number of buckets
    - larger $R \Rightarrow$ smaller m
        - less iterations but more work per iteration (larger bucket array)
        - $(100010)_2 = (34)_{10}$
- 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

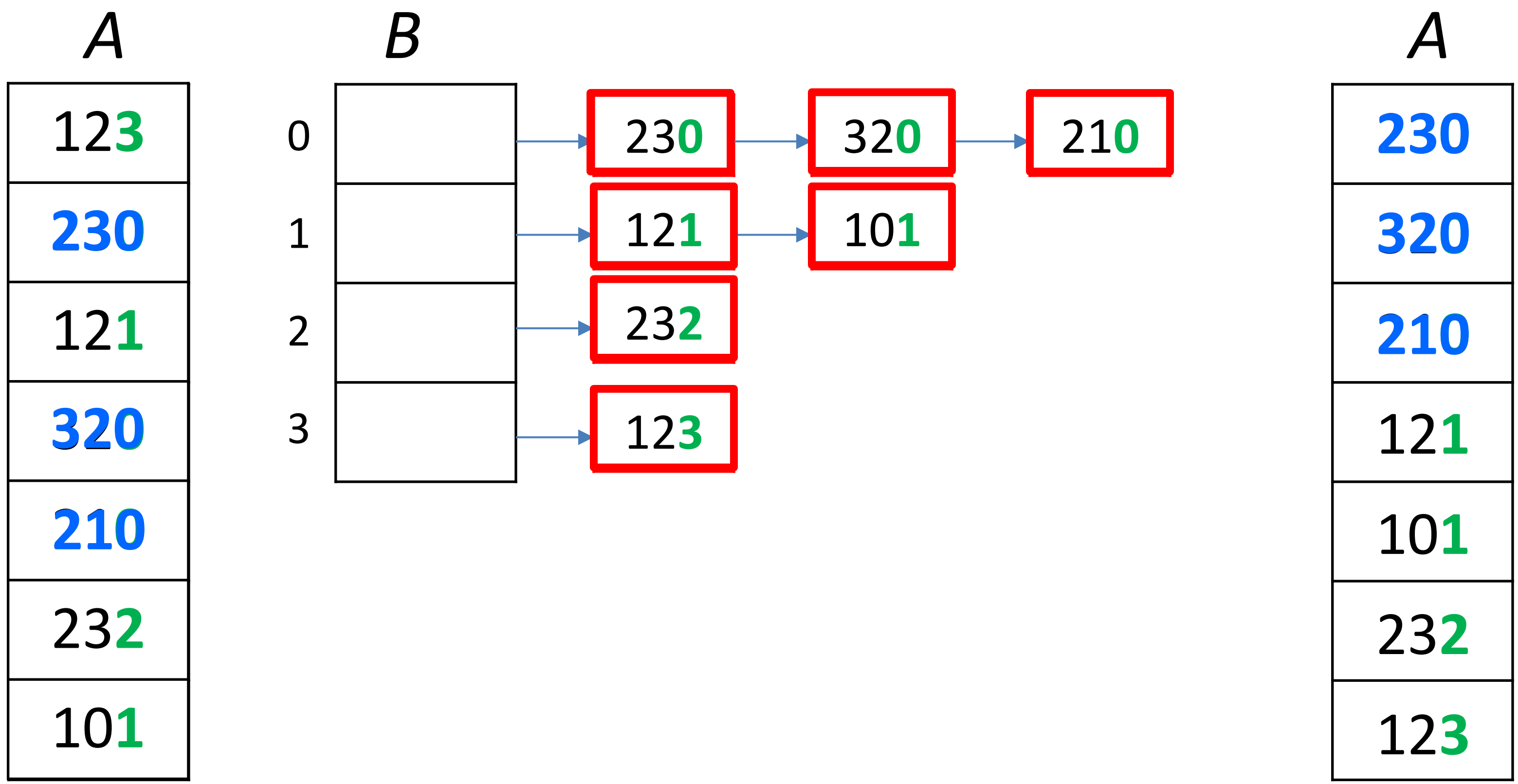
Bucket Sort on Last Digit

- 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
  - example: $211 < 123$



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
  - example: $211 < 123$



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

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 $\Theta(n)$

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

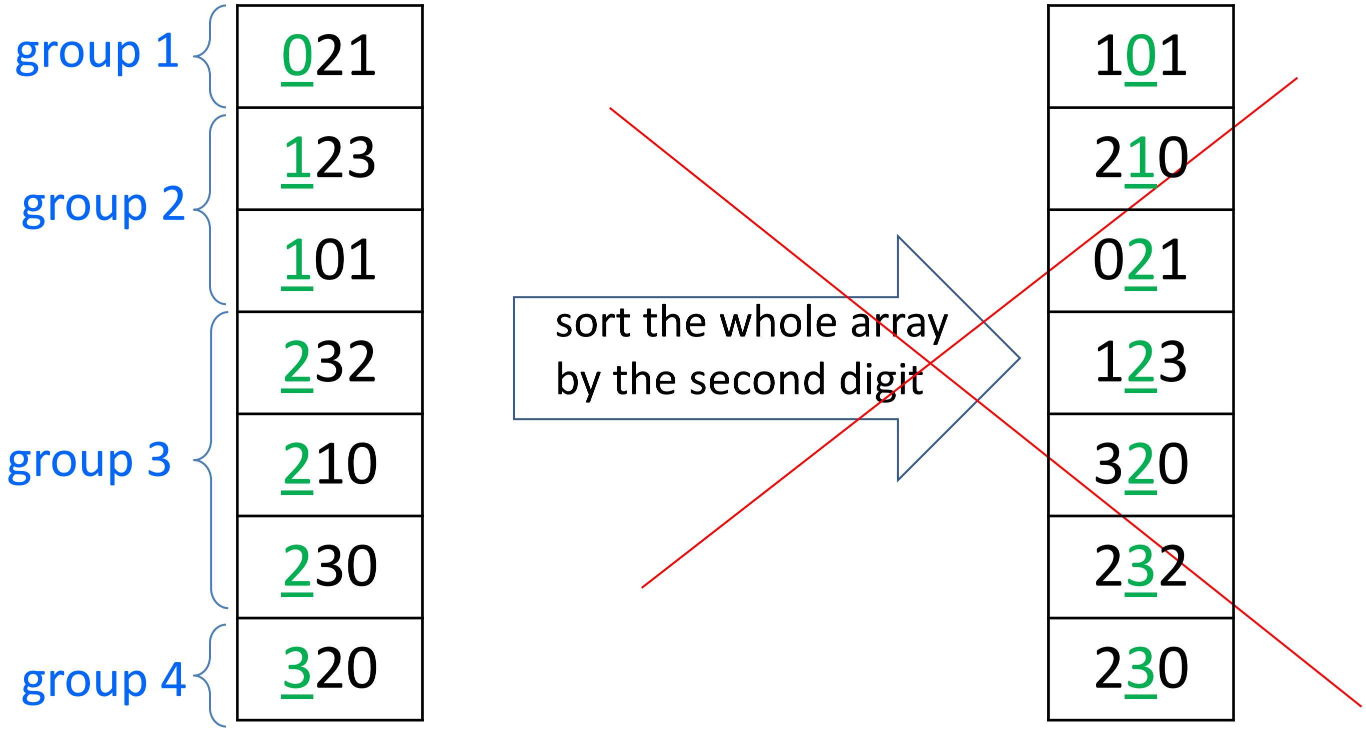
MSD-Radix-Sort

- 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

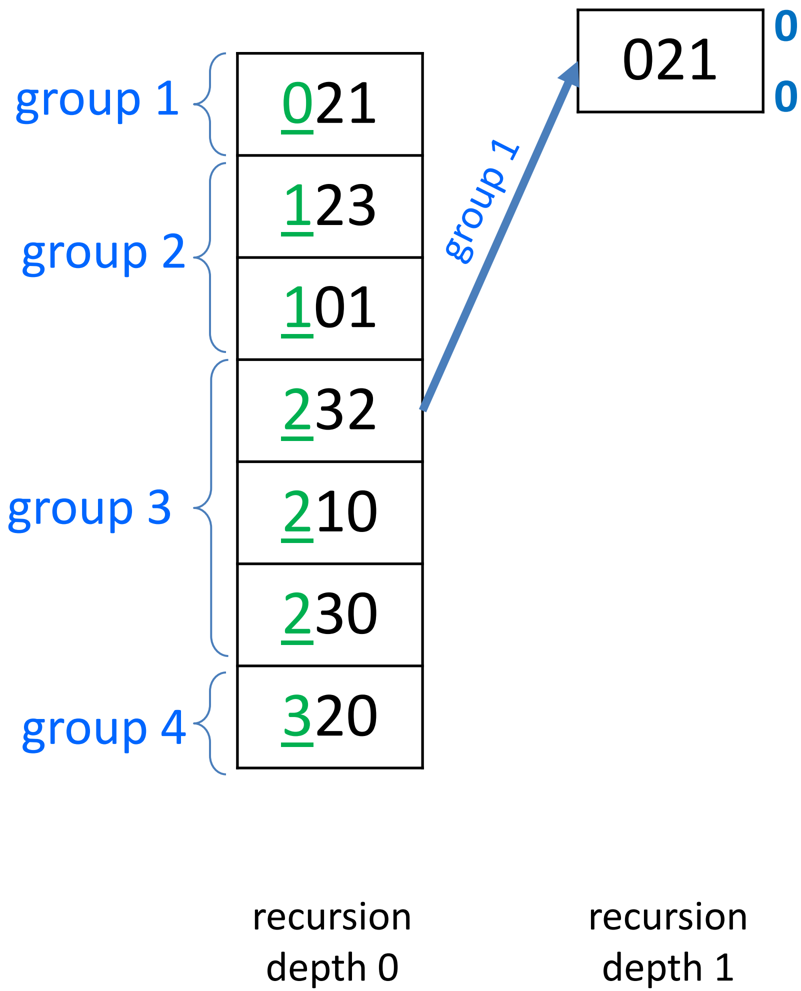
- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit



- 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

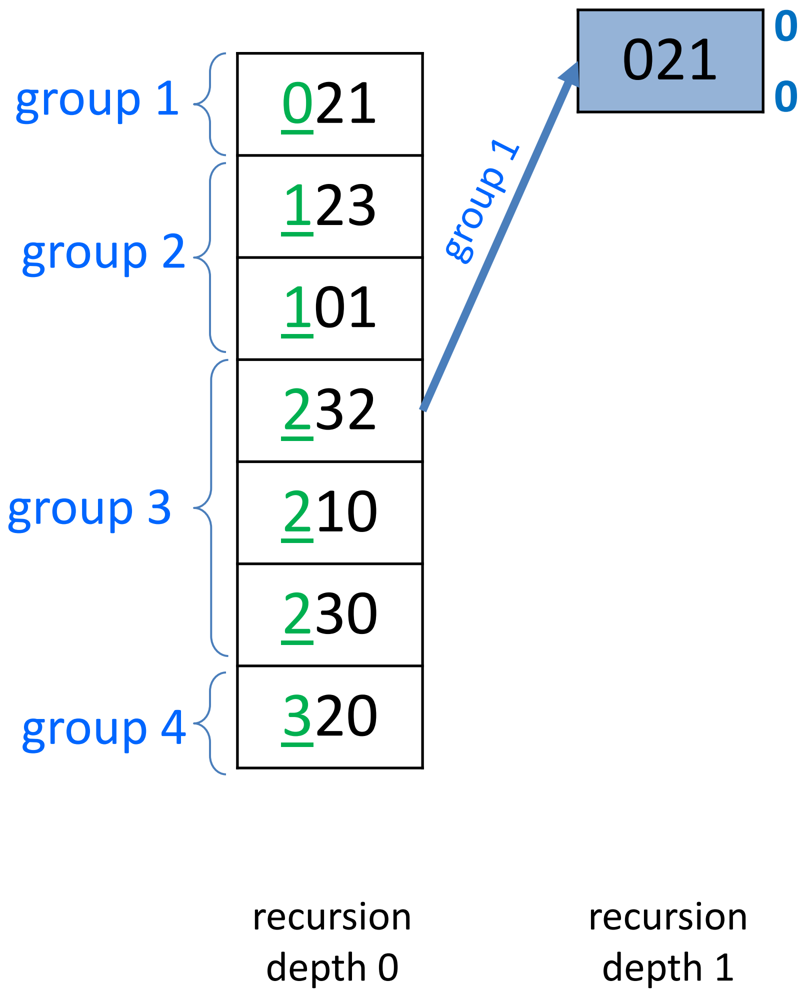
MSD-Radix-Sort

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



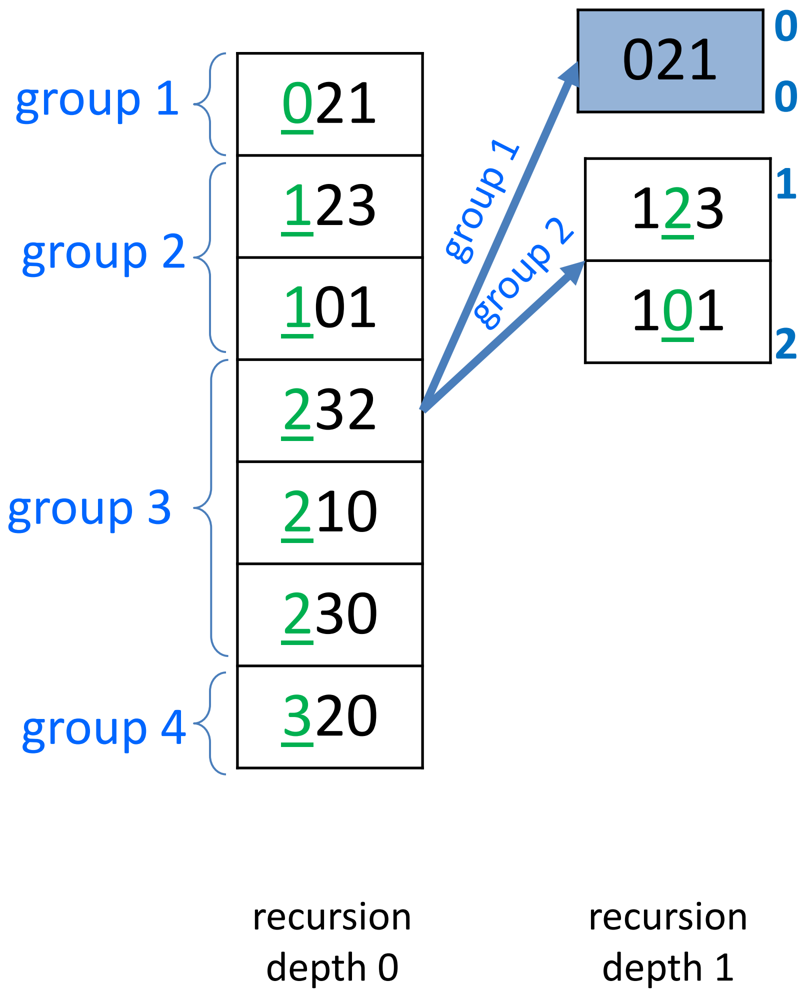
MSD-Radix-Sort

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



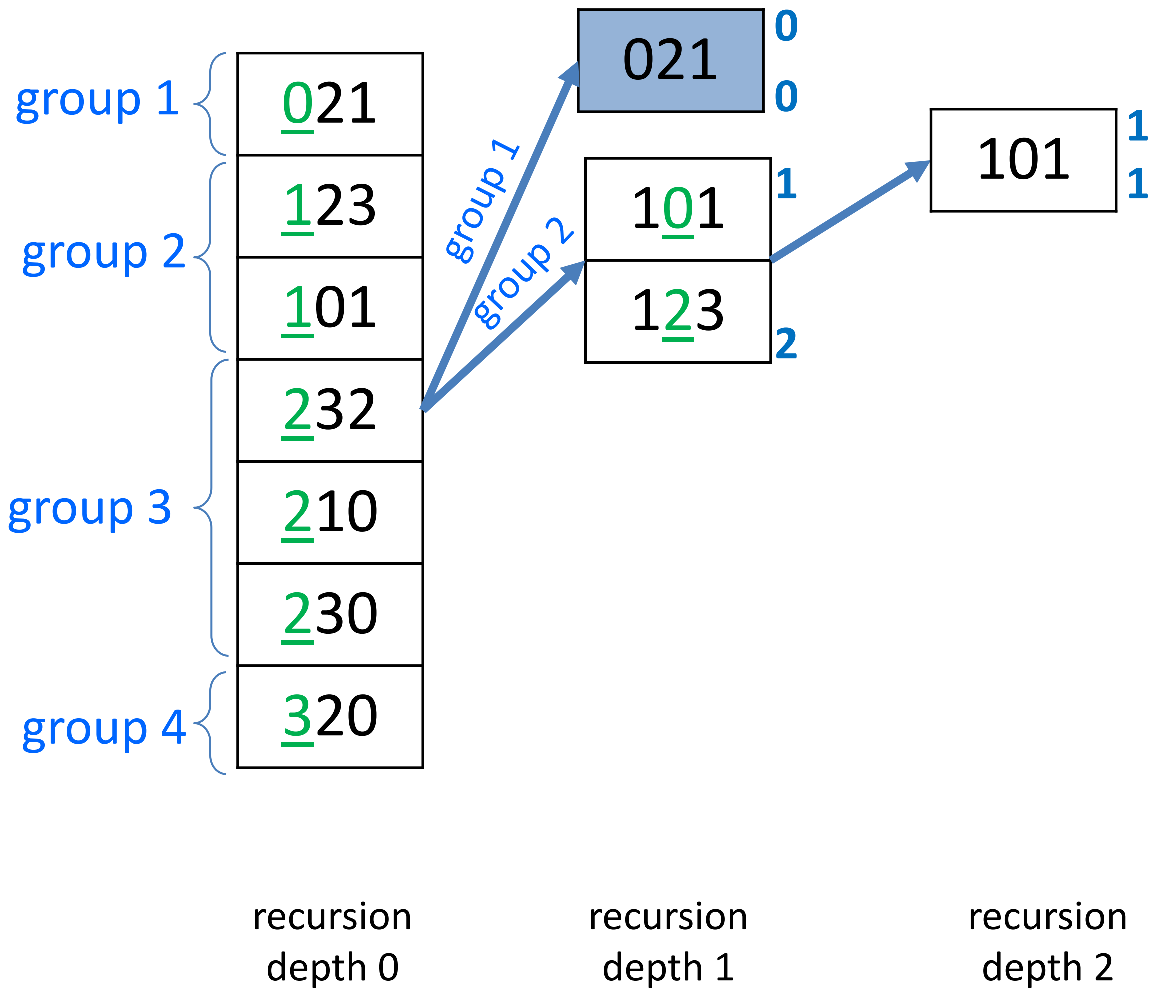
MSD-Radix-Sort

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



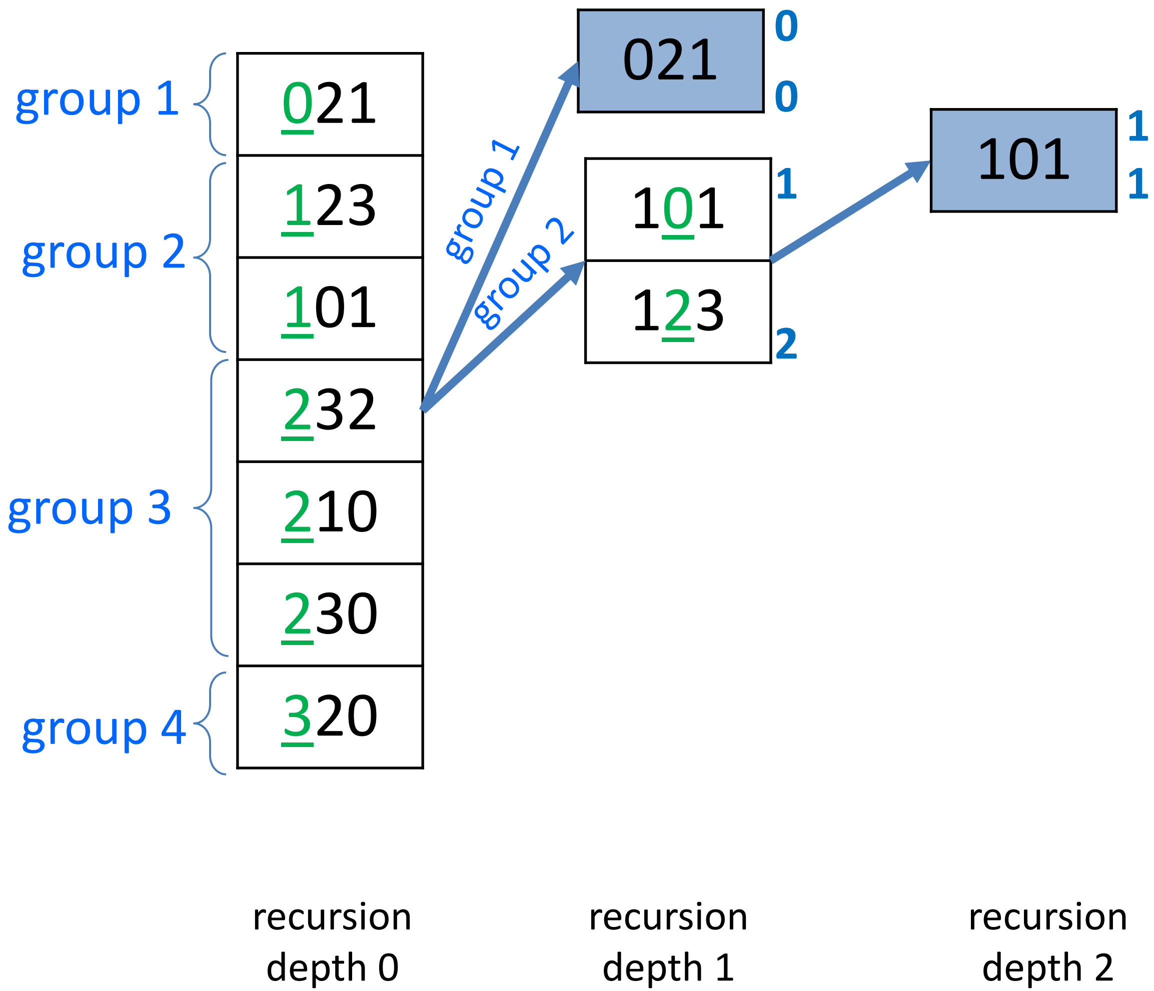
MSD-Radix-Sort

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



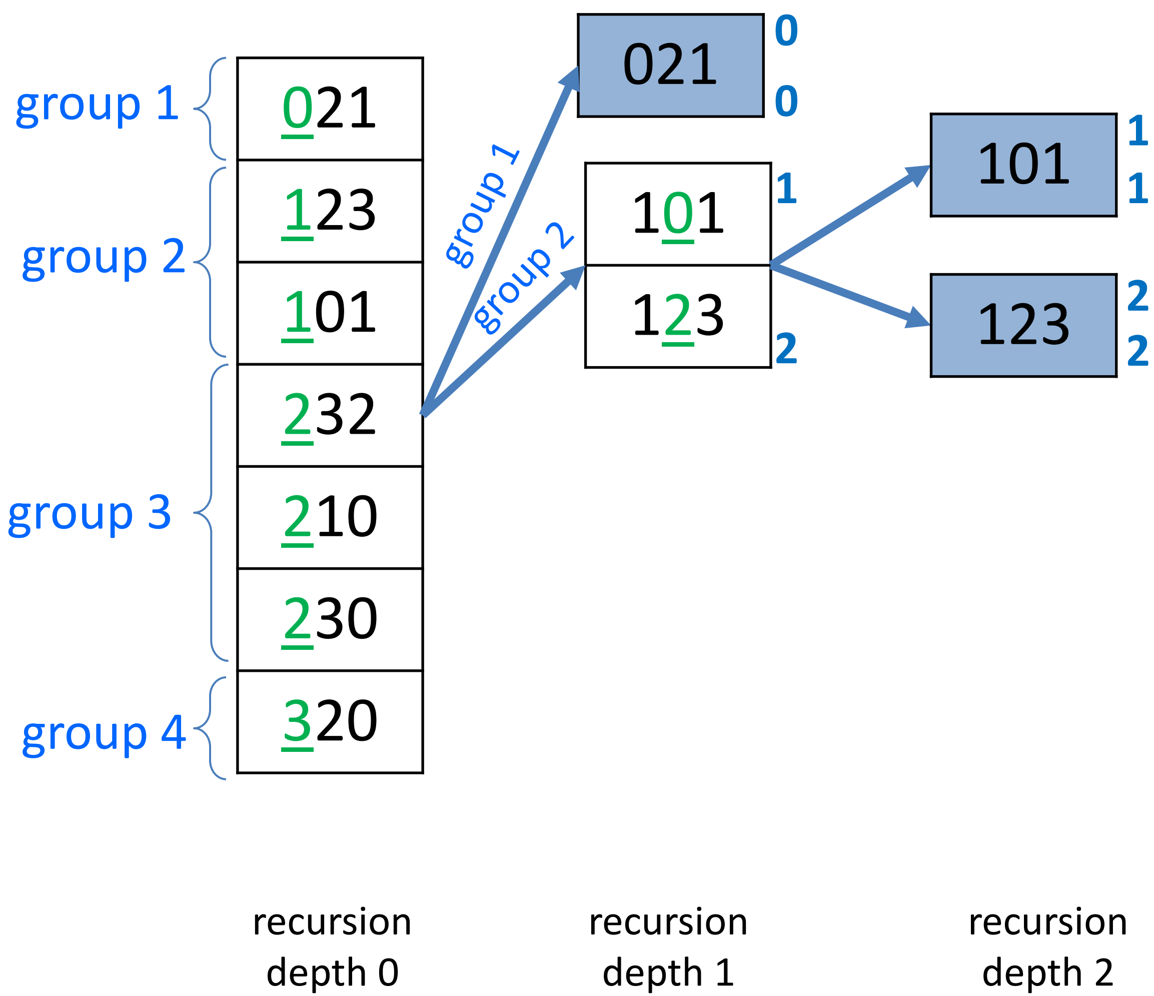
MSD-Radix-Sort

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



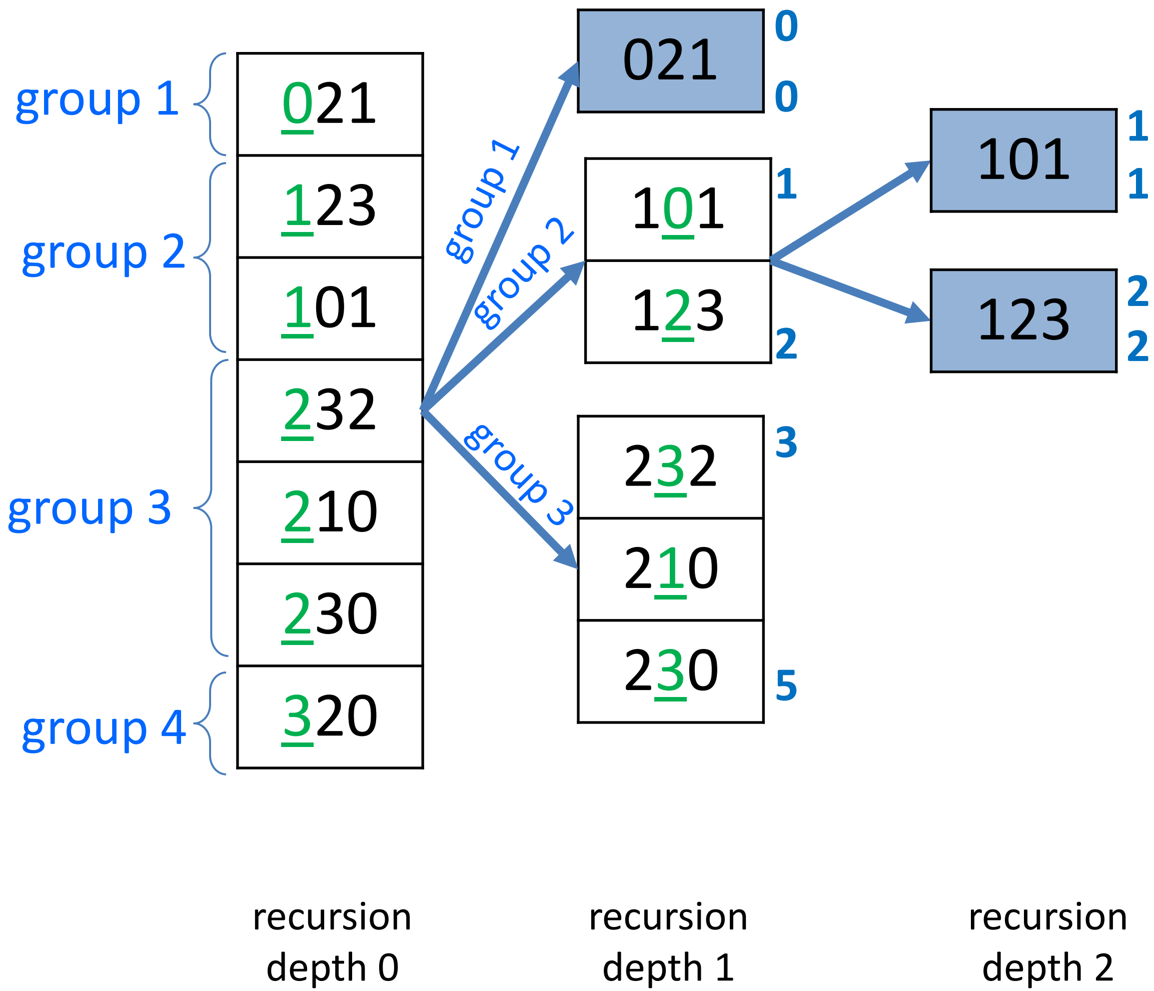
MSD-Radix-Sort

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



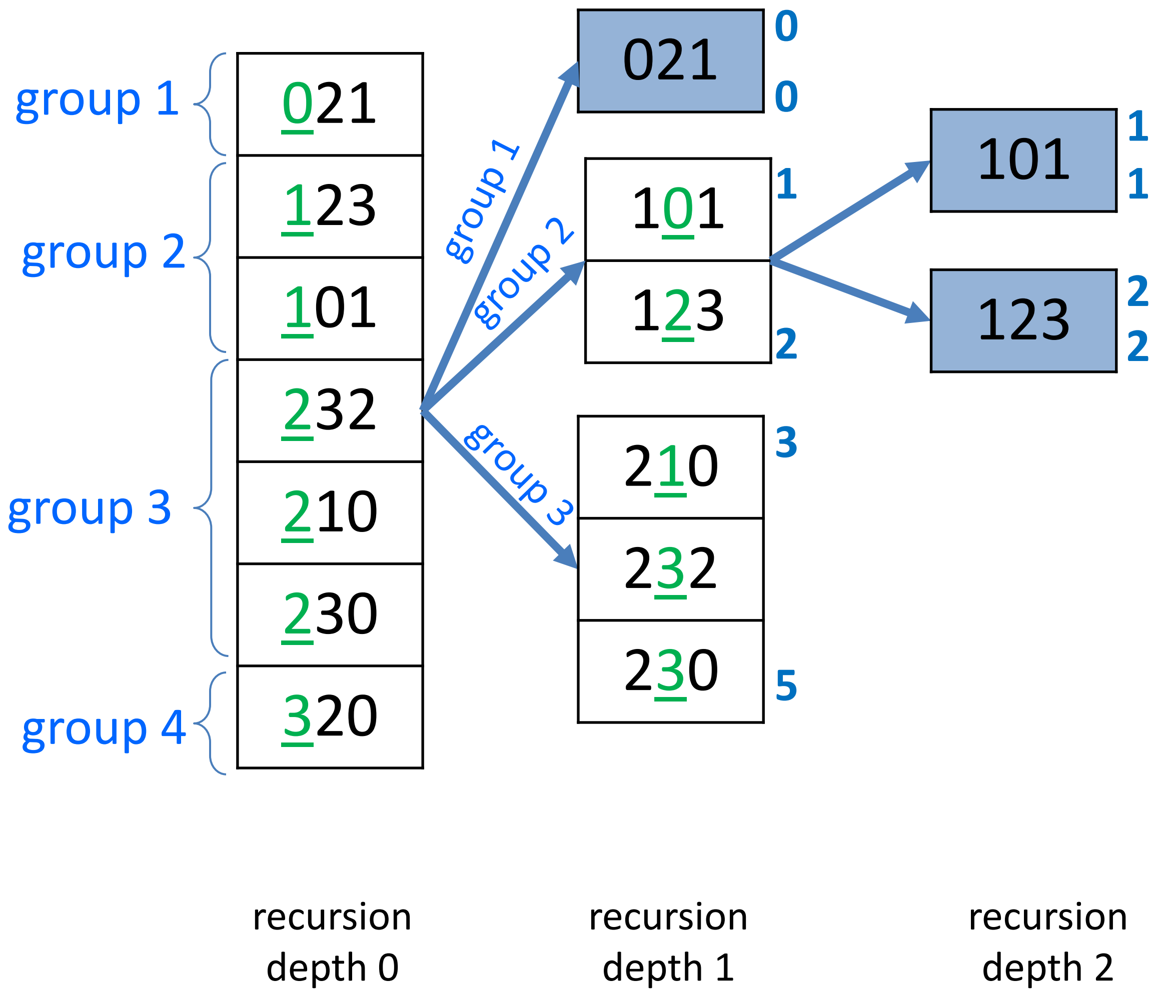
MSD-Radix-Sort

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



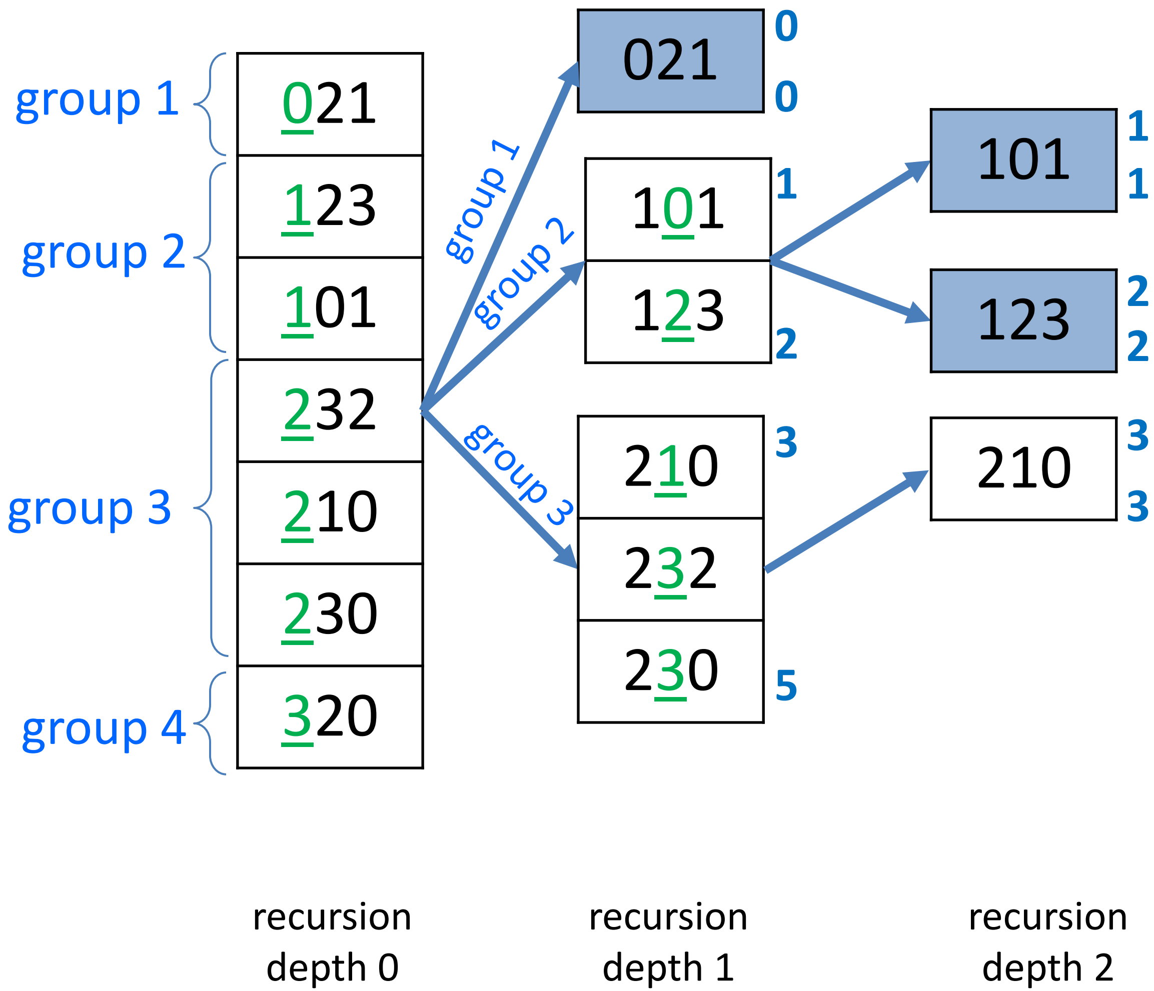
MSD-Radix-Sort

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



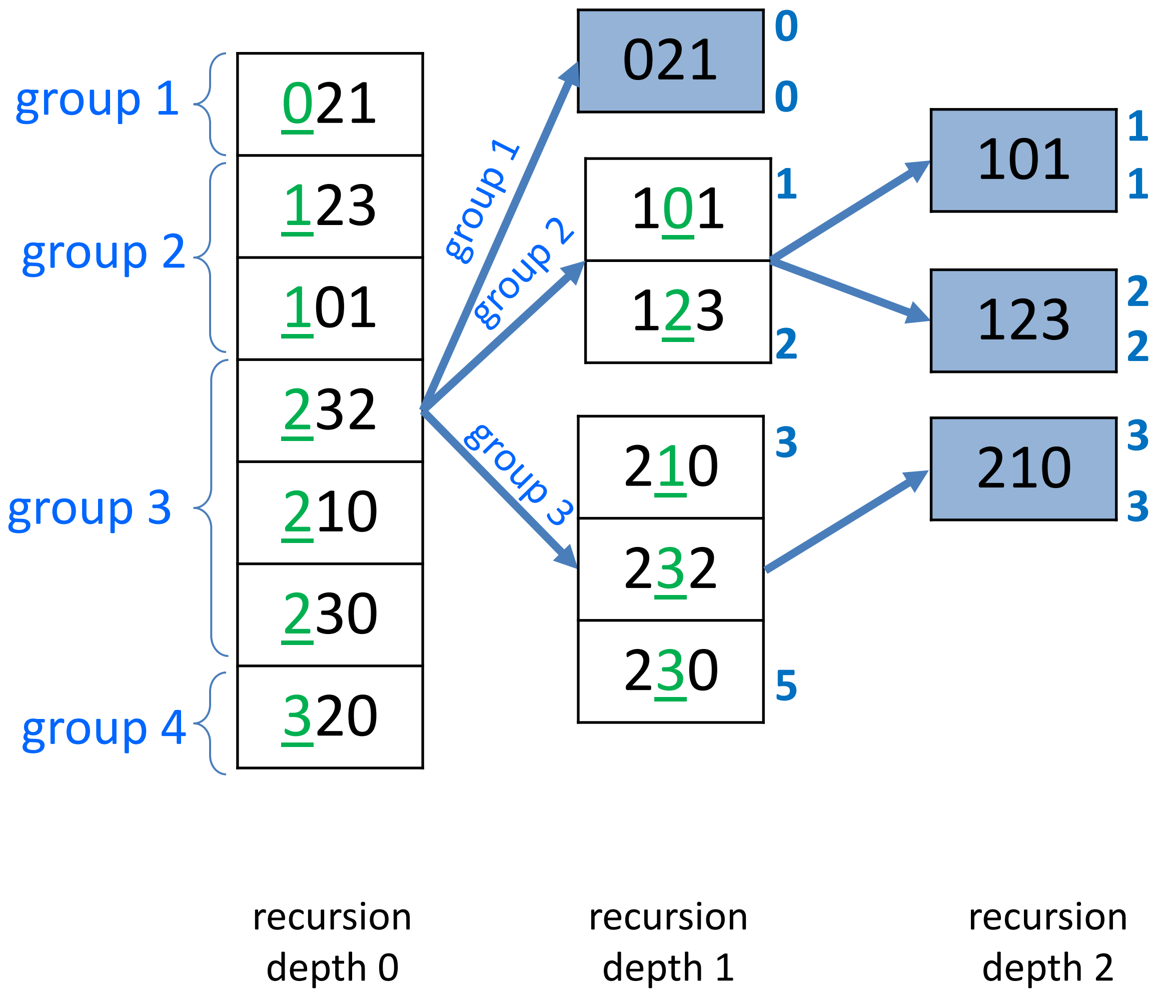
MSD-Radix-Sort

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



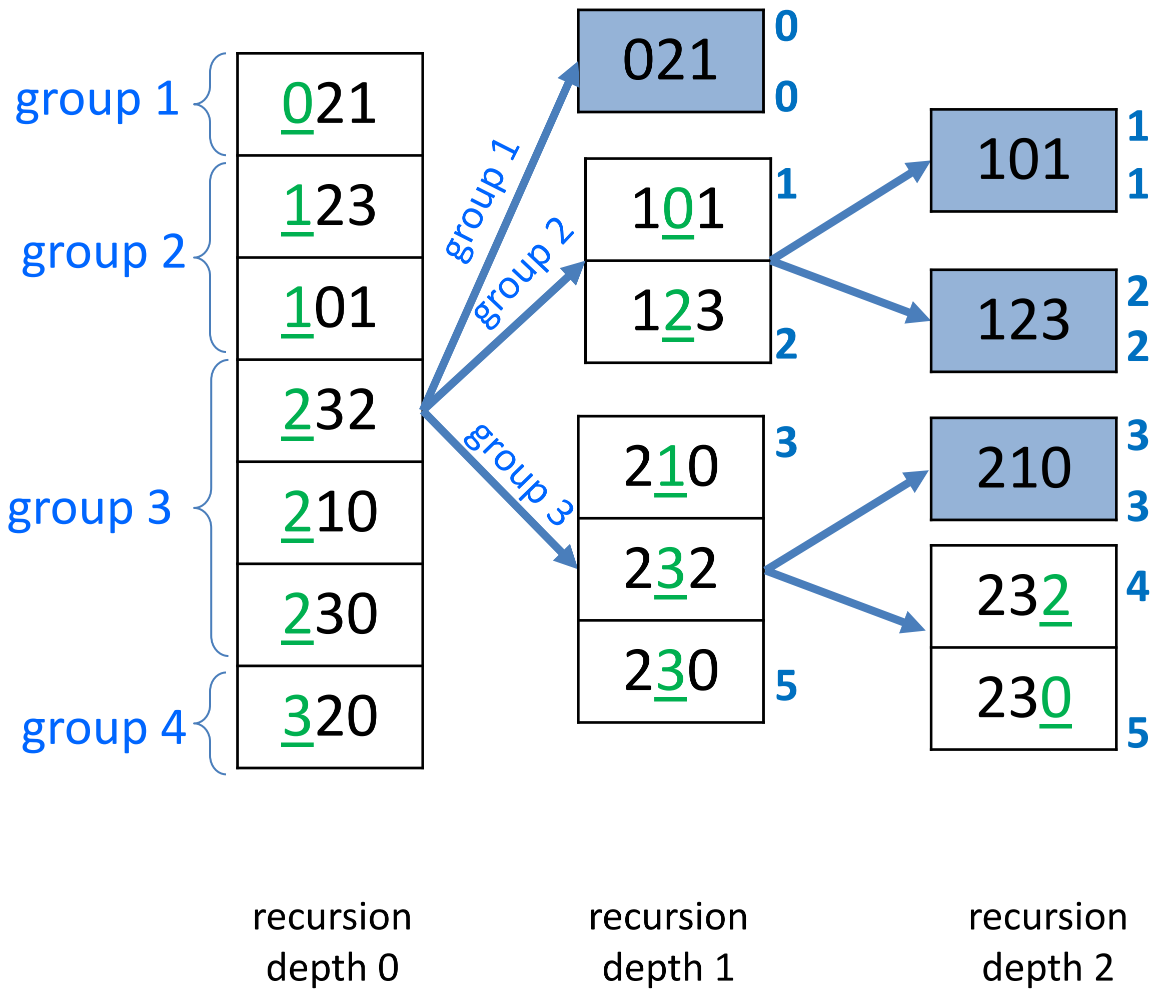
MSD-Radix-Sort

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



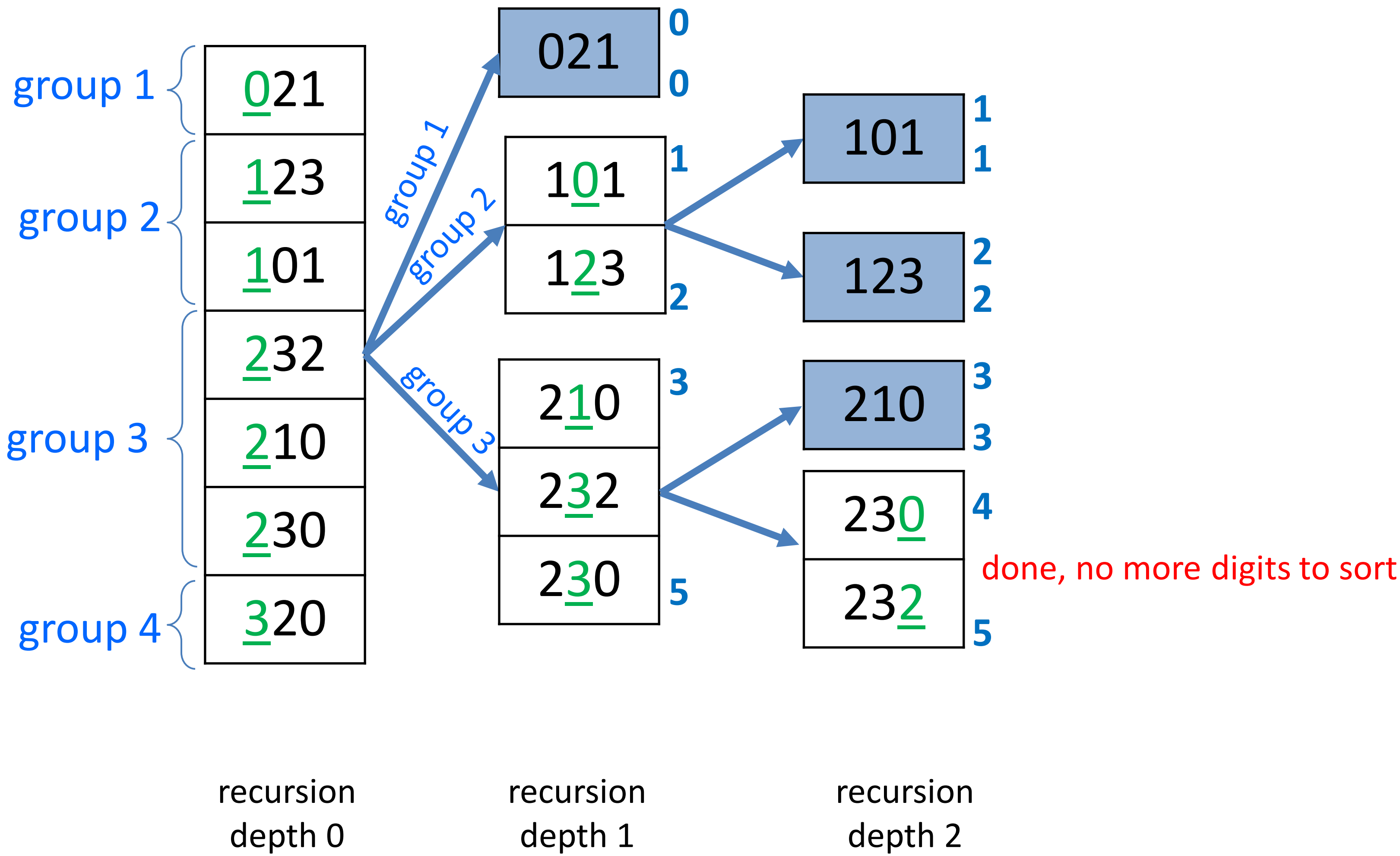
MSD-Radix-Sort

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



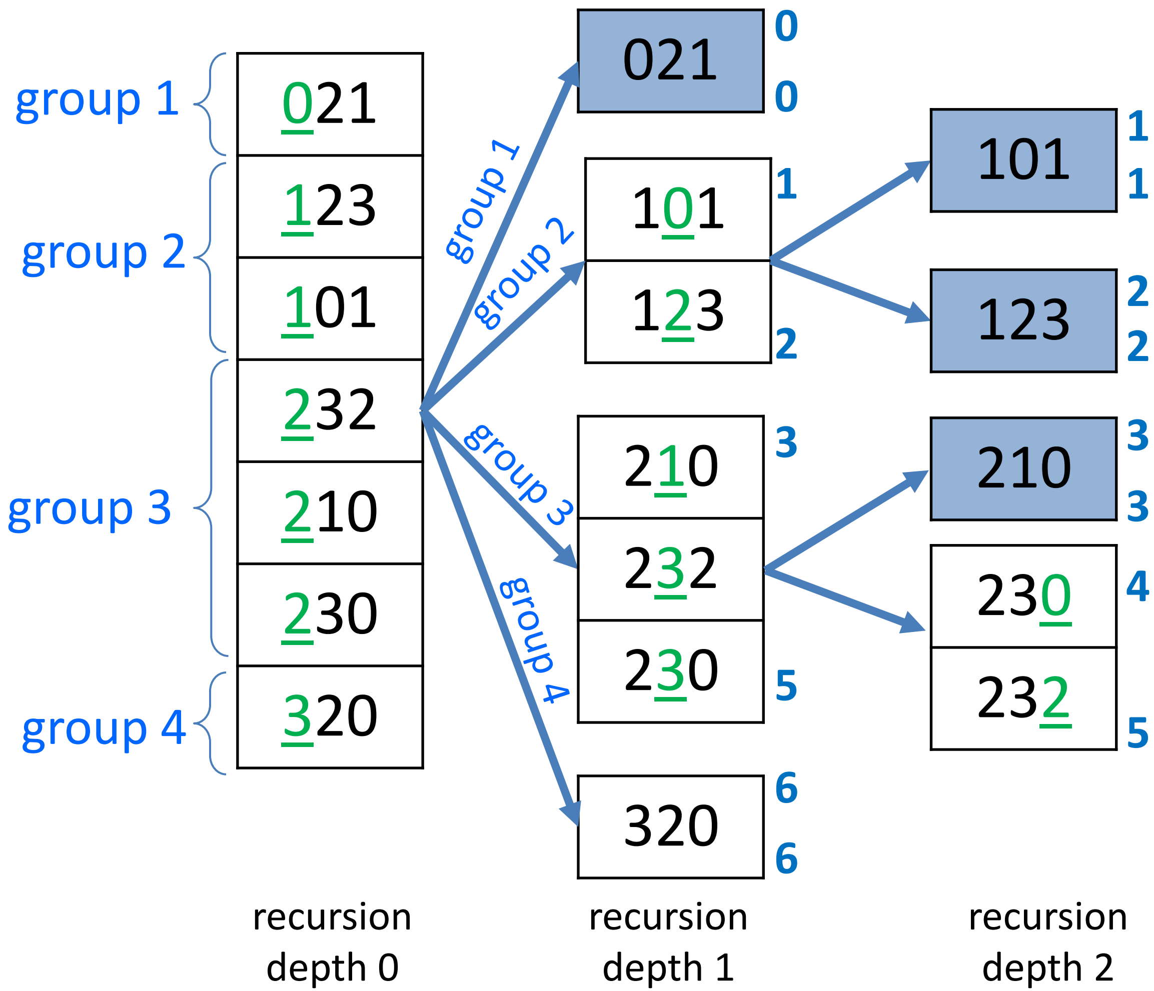
MSD-Radix-Sort

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



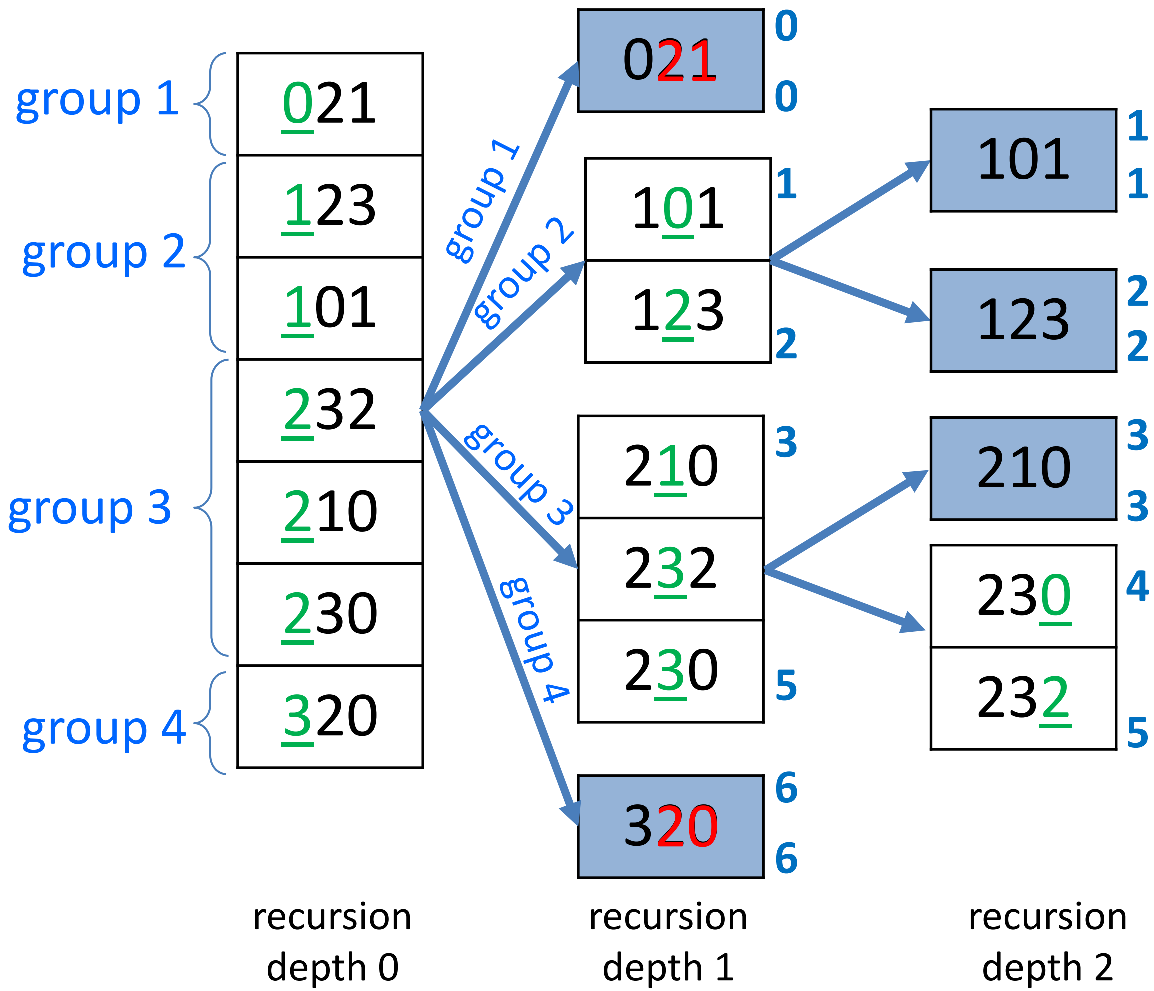
MSD-Radix-Sort

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



MSD-Radix-Sort

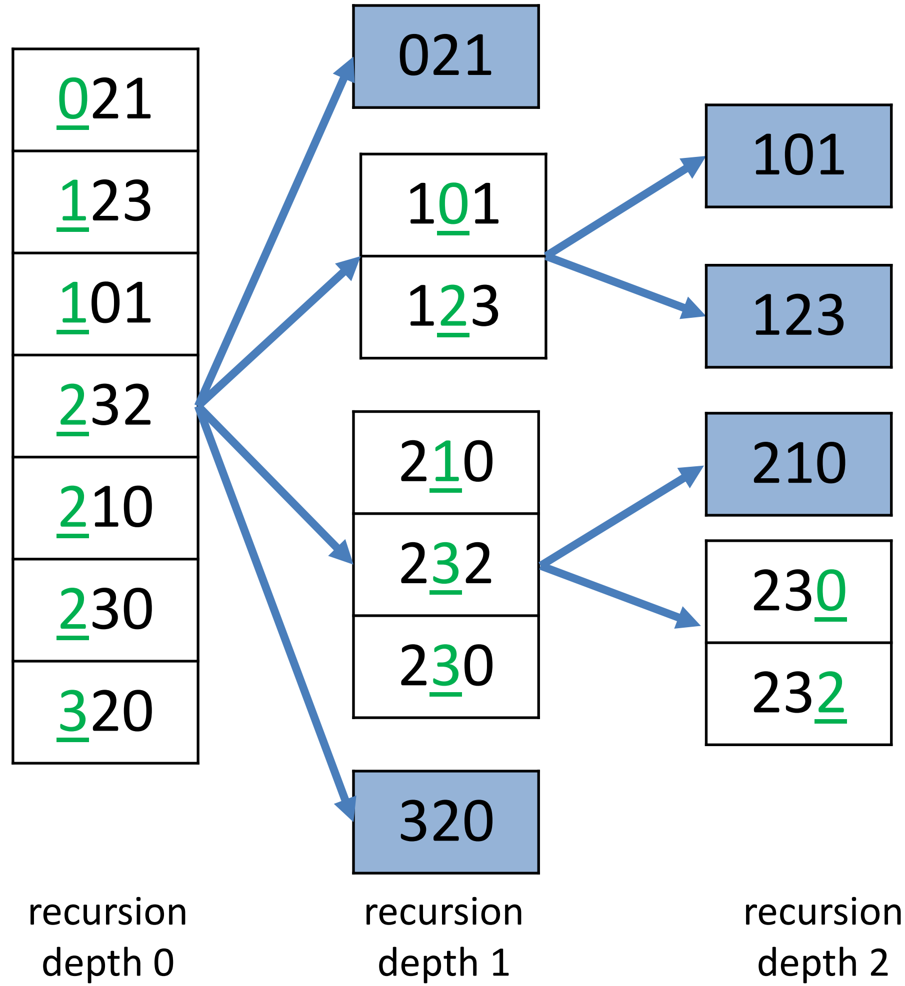
- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



many digits are never examined

MSD-Radix-Sort Space Analysis

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

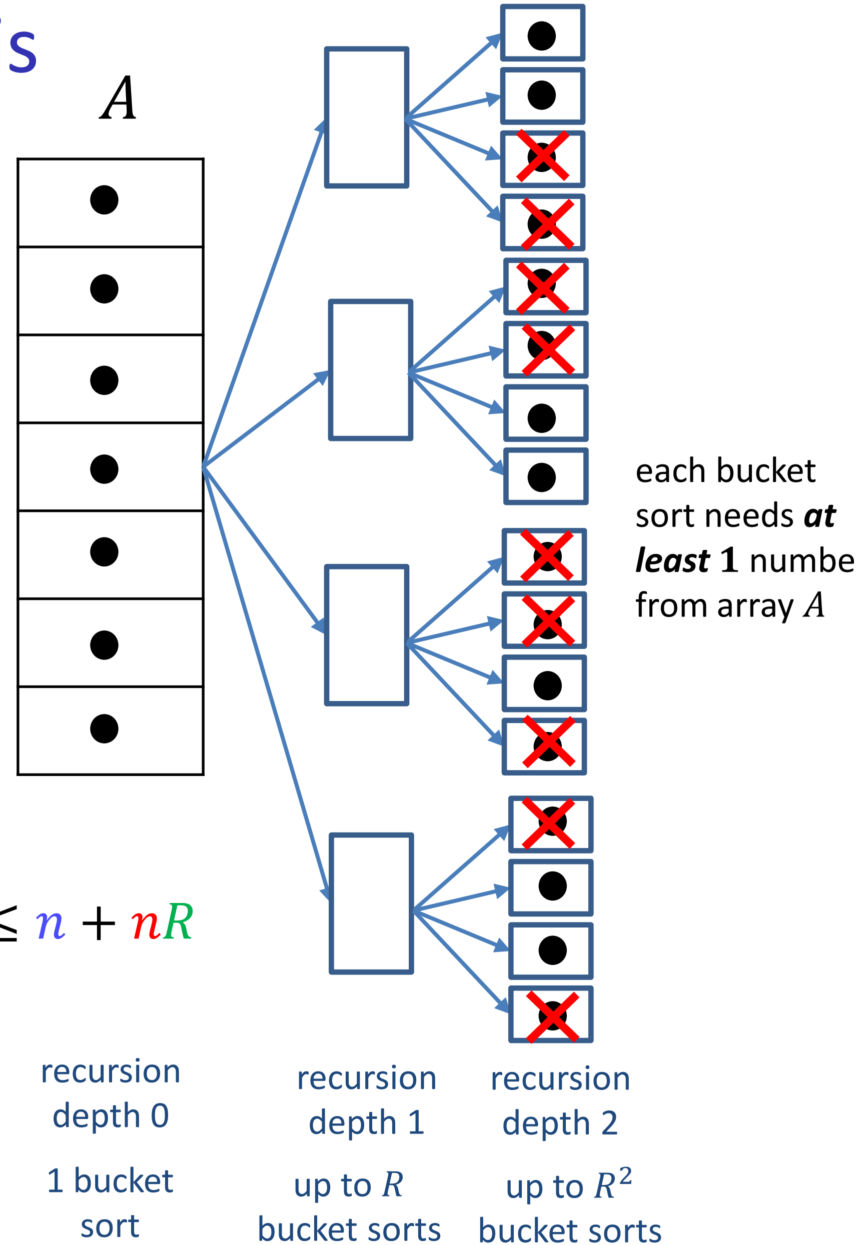


MSD-Radix-Sort Time Analysis

- Time spent for each recursion depth
  - Depth $d = 0$
    - one bucket sort on n items
    - $\Theta(n+R)$
  - At depth $d > 0$
    - let k be number of bucket sorts
    - $k \le n$
    - index bucketsorts as $1, \ldots, i \ldots, k$
    - bucketsort i involves n_i keys
    - bucket sort i takes $n_i + R$ time

$$\sum_{i=1}^{k}(n_i+R) = \sum_{i=1}^{k} n_i + \sum_{i=1}^{k} R \le n + kR \le n + nR$$

    - total time at depth d is $O(nR)$
- Number of depths is at most $m-1$
- Total time $O(mnR)$



at any depth, number of bucketsorts $\le n$

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, $0 \le l, r \le n-1$

if $l < r$

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

if there are digits left

$l' \leftarrow l$

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

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

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

$l' \leftarrow r' + 1$

- Run-time $O(mnR)$, auxiliary space is $\Theta(m+n+R)$
- Advantage: many digits may remain unexamined
- Drawback: many recursions

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

- Comparing different R
  - sort n integers in the range $[0, 2^{10})$
  - if $R = 2$, then $m = 10$, and sorting is $O(20n)$
  - if $R = 10$, then $m = 4$ $(2^{10} = 1024)$ and sorting is $O(40n)$

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	sorted by last digit	prepare to sort by middle digit	sorted by last two digits	prepare to sort by first digit	sorted by all three digits

- 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
- Randomized algorithms can eliminate “bad instances”
- Best-case, worst-case, average-case can all differ
- Often easier to analyze the run-time on randomly chosen input rather than the average-case runtime