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 2024

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**
- Cons:
    - usually difficult to compute
    - average-case and worst case run times are often 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'

$$\mathbb{I}_3 = \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 2 & 1 \\ \hline 1 & 0 & 2 \\ \hline 1 & 2 & 0 \\ \hline 2 & 0 & 1 \\ \hline 2 & 1 & 0 \\ \hline \end{array}$$

- 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 $\{0,1,\ldots,n-1\}$

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

- $n!$ 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

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

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

Average Case Analysis: Example 2

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

$sortednessTester(A, n)$

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*

- 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 2

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

$sortednessTester(A, n)$

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

- 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

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

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

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

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

- 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

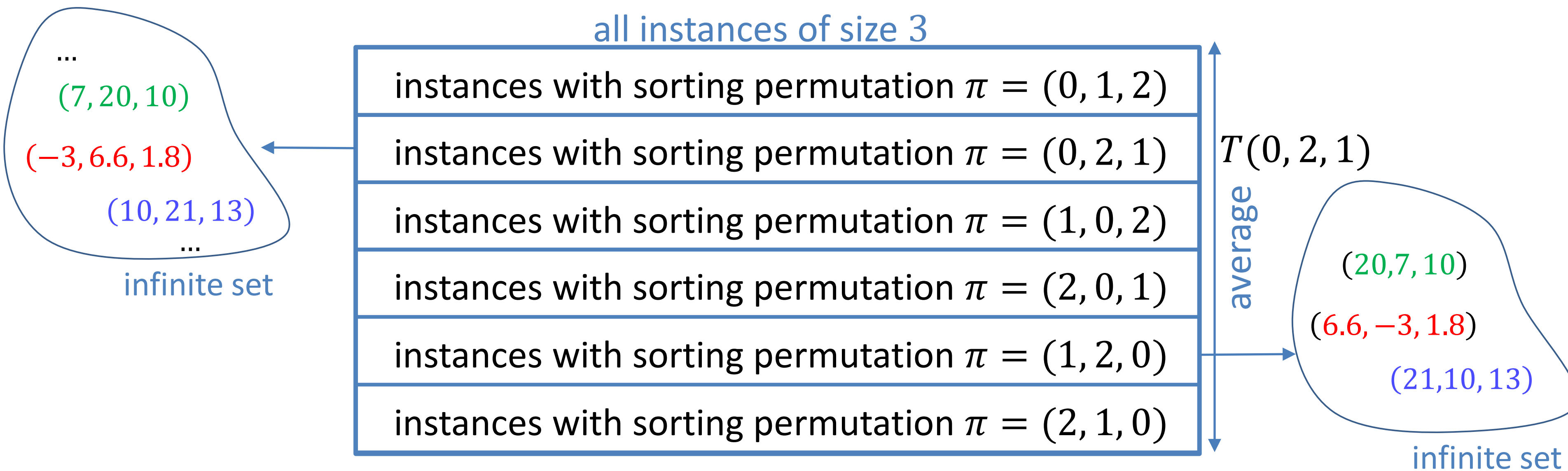
$$\pi = (4, 1, 2, 3, 6, 5, 0)$$

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: Example 1

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

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

- Runtime is $cn + c$
- Number of comparisons is $n - 1$
- Runtime is Θ(number of comparisons)
- To get rid of the constant in all calculations, let us define

$$T(\pi) = \textit{number of comparisons}$$

Average Case: 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 ← 1 to n − 1 do
        if A[i − 1] > A[i] then return false
    return true
```

- $T(\pi) =$ *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: $T(\pi) = n-1$

$$T^{avg}(3) = \frac{1}{3!}\big(\underbrace{T(0,1,2)}^{\text{3 comp}} + \underbrace{T(0,2,1)}^{\text{2 comp}} + \underbrace{T(1,0,2)}^{\text{1 comp}} + \underbrace{T(2,0,1)}^{\text{1 comp}} + \underbrace{T(1,2,0)}^{\text{2 comp}} + \underbrace{T(2,1,0)}^{\text{1 comp}}\big)$$

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

$$= \frac{1}{6}(3\cdot 1 + 2\cdot 2 + 1\cdot 3) = 10/6$$

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

Average Case Analysis: Example 1

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

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

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

#permutations with at least k comparisons

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

#permutations with exactly k comparisons

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

Average Case Analysis: Example 1

$sortednessTester(A, n)$

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*

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

- Permutations with at least 1 comparison
  - all $n!$ permutations

Average Case Analysis: Example 1

```
sortednessTester(A, n)
    A: array storing n distinct numbers
    for i ← 1 to n − 1 do
        if A[i − 1] > A[i] then return false
    return true
```

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

- Permutations with at least 2 comparisons
  - $A[0] < A[1]$

0	1	2	3	4	5	6
3	15	4	6	1	20	8

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

  - $0, 1$ occur in sorted order : $(4,3,2,0,1)$, $(4,3,0,2,1)$, $(4,0,3,2,1)$
  - $\binom{n}{2}(n-2)!$

Average Case Analysis: Example 1

$sortednessTester(A, n)$

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*

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

- Permutations with at least 3 comparisons
  - $A[0] < A[1] < A[2]$

0	1	2	3	4	5	6
3	15	44	6	1	20	8

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

  - $0, 1, 2$ occur in sorted order : $(4, 3, 0, 1, 2)$, $(4, 0, 3, 1, 2)$, $(0, 1, 3, 4, 2)$
  - $\binom{n}{3}(n-3)!$

Average Case Analysis: Example 1

> $\color{blue}{sortednessTester}(A, n)$
>
> 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*

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

- Permutations with at least k comparisons
  - $A[0] < A[1] < A[2] \ldots < A[k-1]$
  - $0, 1, \ldots, k$ occur in sorted order
  - $\binom{n}{k}(n-k)! = \frac{n!}{(n-k)!\,k!}(n-k)! = \frac{n!}{k!}$

Average Case Analysis: Example 1

- Let π_k be # of permutations with at least k comparisons, $\pi_k = \frac{n!}{k!}$
- Taylor expansion: $\sum_{k=0}^{\infty} \frac{1}{k!} = e \approx 2.8$

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

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

$$-1 \cdot \pi_2 - 2 \cdot \pi_3 - \cdots - (n-2) \cdot \pi_{n-1} - (n-1) \cdot \pi_n$$

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

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

- 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 $n \le 2$ **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

- Let $T(n)$ be the number of recursions
  - proportional to the running time
- Best case (array sorted in increasing order)
  - always get the good case, array size is divided by 2 at each recursion
  - $T(n) = \begin{cases} 0 & \text{if } n \le 2 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$
  - 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
  - $T(n) = T(n-2) + 1$ (for $n > 2$)
  - resolves to $\Theta(n)$

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

- $avgCaseDemo$ runtime is equal for instances with same relative element order
- Therefore can use sorting permutations for 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

good		bad
$(0, 1, 2)$	$\leftrightarrow$	$(0, 2, 1)$
$(1, 0, 2)$	$\leftrightarrow$	$(1, 2, 0)$
$(2, 0, 1)$	$\leftrightarrow$	$(2, 1, 0)$

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

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

- Cannot conclude that $$\cancel{T^{avg}(n) = \begin{cases} 1 + T^{avg}(n/2) & \text{if } \pi \text{ is good} \\ 1 + T^{avg}(n-2) & \text{if } \pi \text{ is bad} \end{cases}}$$

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

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

Average Case Analysis: Example 2

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

- Using formula for the **sum** of instances π from the previous slide

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

- Recall that there are $n!/2$ good permutations, $n!/2$ bad permutations

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

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

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

Average Case Analysis: Example 2

$$T^{avg}(n) = 1 + \frac{1}{2}T^{avg}(\ n/2) + \frac{1}{2}T^{avg}(\ n-2) \text{ if } n > 2$$
$$T^{avg}(n) = 0 \text{ if } n \le 2$$

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 $m < n$
- $T^{avg}(n) = 1 + \frac{1}{2}\underbrace{T^{avg}(n/2)}_{\text{induction hypothesis}} + \frac{1}{2}\underbrace{T^{avg}(n-2)}_{\text{induction hypothesis}}$

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

- This proves average-case running time is $O(\log(n))$
  - best case is $\Theta(\log(n))$
  - average case cannot be better than best case
  - therefore, average case is $\Theta(\log(n))$, much 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

$avgCaseDemo(A, n)$
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

- Average case is $O(\log(n))$ and worst-case is $O(n)$
- Would hope that in practice, time averaged over **different runs** of *avgCaseDemo* is $O(\log(n))$
- However, recall 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
    - averaging over **different runs** in practice for many algorithms is equivalent to averaging over **instances**
    - can expect *avgCaseDemo* to have $O(\log(n))$ runtime averaged over runs
- But humans often generate instances that are far from equally likely
- For example, if user mostly calls $avgCaseDemo$ on almost reverse sorted arrays, runtime averaged over **different runs** is $\Theta(n)$ in practice

Randomized Algorithms: Motivation

- Randomization improves runtime in practice when instances are not equally likely
  - makes sense to randomize algorithms which have better average-case than worst-case runtime

$avgCaseDemo(A, n)$
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

- Simple randomization: shuffle array A before calling $avgCaseDemo$, so that every instance is equally likely
  - now averaging over **runs** is the same as averaging over ***instances***
    - $O(\log(n))$
  - 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
- Runtime depends on both input I and random numbers R used
- **Goal:** shift dependency of run-time from what we cannot control (user input), to what we can control (random numbers)
  - 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
    - however, this is exceedingly rare, think of chances of sorting an array by a random sequence of swaps
- Side note: computers cannot generate truly random numbers
  - 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?
  - 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*

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

- Best-case and average-case expected running time defined similarly
- Usually consider only worst-case expected running time
  - usually design a randomized algorithm so that all instances of size n have the same expected runtime
- Sometimes we also want to know the running time if we get really unlucky with the random numbers R, i.e. worst case (or worst instance and worst random numbers case)

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

Randomized Algorithm: *Simple*

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

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

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

- Function $random(n)$ returns an integer sampled uniformly from $\{0, 1, \dots, n-1\}$
- $simple$ 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 running time, so $T^{exp}(n) \in \Theta(n)$

Randomized Algorithm: *Simple2*

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

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

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

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

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

$$T^{exp}(I) = \sum_{<r_1,\, r_2>} T(I, < r_1,\, r_2 >) \cdot \left(\frac{1}{n}\right)^2 = \left(\frac{1}{n}\right)^2 \sum_{<r_1,\, r_2>} c \cdot r_1 \cdot r_2$$

$$= \left(\frac{1}{n}\right)^2 \sum_{r_1} c \cdot r_1 \sum_{r_2} r_2 = \left(\frac{1}{n}\right)^2 \sum_{r_1} c \cdot r_1 \frac{n(n-1)}{2} = \left(\frac{1}{n}\right)^2 c \frac{n(n-1)}{2} \frac{n(n-1)}{2}$$

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

Randomized Algorithm: *expectedDemo*

```
avgCaseDemo(A, n)
  A: array storing n distinct numbers
  if n ≤ 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
```

- To randomize *avgCaseDemo*, could shuffle array A and then call *avgcaseDemo*
- A better solution which avoids shuffling

```
expectedDemo(A, n)
  A: array storing n distinct numbers
  if n ≤ 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 $\{0, 1, \ldots, n-1\}$
- For any array, $\Pr(\text{good case}) = \Pr(\text{bad case}) = \frac{1}{2}$

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

- Running time depends **both** on the input array A **and** the sequence R of random numbers generated during the run of the algorithm
  - $A = [1, 5, 0, 3, 7, 3],\ R = \langle 1,0,0 \rangle$
  - Step 1:

    $A = [1, 5, 0, 3, 7, 3] \quad R = \langle 1,0,0 \rangle \Rightarrow A = [1, 5, 0, 3, 3, 7] \Rightarrow \text{good case}$
  - Step 2:

    $A = [1, 5, 0] \quad R = \langle 1,0,0 \rangle \Rightarrow A = [1, 5, 0] \Rightarrow \text{bad case}$

Randomized Algorithm $expectedDemo$

$expectedDemo(A, n)$

A: array storing n distinct numbers

if $n \le 2$ **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 $\{0, 1, \ldots, n-1\}$
- For *any* array A, $\Pr(\text{good case}) = \Pr(\text{bad case}) = \frac{1}{2}$
- 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 numbers are $R = \langle x,\ R' \rangle$

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

examples

bad case since $8 > 1$ and do not swap

$$T([1,0,4,5,8,1], \langle 0, 1,1,0 \rangle) = \ T\big([1,0,4,5,8,1], \big\langle 0, \langle 1,1,0 \rangle \big\rangle\big) = 1 + T([1,0,4,5], \langle 1,1,0 \rangle)$$

good case since $8 > 1$ and we swap

$$T([1,0,4,5,8,1], \langle 1, 0,1,0 \rangle) = T\big([1,0,4,5,8,1], \big\langle 1, \langle 0,1,0 \rangle \big\rangle\big) = 1 + T([1,0,4], \langle 0,1,0 \rangle)$$

Expected running time of *expectedDemo*

$$T^{exp}(A) = \sum_R T(A, R) \cdot \Pr(R)$$

- Summing up over all sequences of random outcomes

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

example

$$\Pr(0)\Pr(0)\Pr(0) = \frac{1}{2}\frac{1}{2}\frac{1}{2}$$

$$\begin{aligned}\sum_R T([1,4,5,8,1], R) \cdot \mathbf{Pr}(\boldsymbol{R}) = \; & T([1,4,5,8,1], \langle \mathbf{0}, \mathbf{0}, \mathbf{0} \rangle) \cdot \Pr(\langle \mathbf{0}, \mathbf{0}, \mathbf{0} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{0}, \mathbf{0}, \mathbf{1} \rangle) \cdot \Pr(\langle \mathbf{0}, \mathbf{0}, \mathbf{1} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{0}, \mathbf{1}, \mathbf{0} \rangle) \cdot \Pr(\langle \mathbf{0}, \mathbf{1}, \mathbf{0} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{0}, \mathbf{1}, \mathbf{1} \rangle) \cdot \Pr(\langle \mathbf{0}, \mathbf{1}, \mathbf{1} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{1}, \mathbf{1}, \mathbf{0} \rangle) \cdot \Pr(\langle \mathbf{1}, \mathbf{1}, \mathbf{0} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle) \cdot \Pr(\langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{1}, \mathbf{0}, \mathbf{0} \rangle) \cdot \Pr(\langle \mathbf{1}, \mathbf{0}, \mathbf{0} \rangle) \\ & + T([1,4,5,8,1], \langle \mathbf{1}, \mathbf{1}, \mathbf{1} \rangle) \cdot \Pr(\langle \mathbf{1}, \mathbf{1}, \mathbf{1} \rangle)\end{aligned}$$

Expected running time of $expectedDemo$

- Summing up over all sequences of random outcomes

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

example

$$\begin{aligned}\sum_{R} T([1,4,5,8,1],R)\cdot \Pr(R) = \; & T\big([1,4,5,8,1],\langle 0,\langle 0,0\rangle\rangle\big)\cdot \Pr(0)\Pr(\langle 0,0\rangle)\\ & +T\big([1,4,5,8,1],\langle 0,\langle 0,1\rangle\rangle\big)\cdot \Pr(0)\Pr(\langle 0,1\rangle)\\ & +T\big([1,4,5,8,1],\langle 0,\langle 1,0\rangle\rangle\big)\cdot \Pr(0)\Pr(\langle 1,0\rangle)\\ & +T\big([1,4,5,8,1],\langle 0,\langle 1,1\rangle\rangle\big)\cdot \Pr(0)\,\Pr\langle 1,1\rangle)\\ & +T\big([1,4,5,8,1],\langle 1,\langle 1,0\rangle\rangle\big)\cdot \Pr(1)\,\Pr(\langle 1,0\rangle)\\ & +T\big([1,4,5,8,1],\langle 1,\langle 0,1\rangle\rangle\big)\cdot \Pr(1)\,\Pr(\langle 0,1\rangle)\\ & +T\big([1,4,5,8,1],\langle 1,\langle 0,0\rangle\rangle\big)\cdot \Pr(1)\,\Pr(\langle 0,0\rangle)\\ & +T\big([1,4,5,8,1],\langle 1,\langle 1,1\rangle\rangle\big)\cdot \Pr(1)\,\Pr(\langle 1,1\rangle)\end{aligned}$$

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

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

example

$$\sum_{R} T([1,4,5,8,1],R)\cdot \Pr(R) = \boxed{\begin{aligned} & T\big([1,4,5,8,1],\big\langle 0,\langle 0,0\rangle\big\rangle\big)\cdot \Pr(0)\Pr(\langle 0,0\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 0,\langle 0,1\rangle\big\rangle\big)\cdot \Pr(0)\Pr(\langle 0,1\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 0,\langle 1,0\rangle\big\rangle\big)\cdot \Pr(0)\Pr(\langle 1,0\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 0,\langle 1,1\rangle\big\rangle\big)\cdot \Pr(0)\Pr\langle 1,1\rangle) \end{aligned}}$$

$$\boxed{\begin{aligned} & +T\big([1,4,5,8,1],\big\langle 1,\langle 1,0\rangle\big\rangle\big)\cdot \Pr(1)\Pr(\langle 1,0\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 1,\langle 0,1\rangle\big\rangle\big)\cdot \Pr(1)\Pr(\langle 0,1\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 1,\langle 0,0\rangle\big\rangle\big)\cdot \Pr(1)\Pr(\langle 0,0\rangle) \\ & +T\big([1,4,5,8,1],\big\langle 1,\langle 1,1\rangle\big\rangle\big)\cdot \Pr(1)\Pr(\langle 1,1\rangle) \end{aligned}}$$

Expected running time of $expectedDemo$

- Summing up c

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

$expectedDemo(A, n)$

A: array storing n distinct numbers

if $n \le 2$ **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

example

$$\sum_R T([1,4,5,8,1], R) \cdot \Pr(R) = T\big([1,4,5,8,1], \langle 0, \langle 0,0\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 0,0\rangle)$$

$$+T\big([1,4,5,8,1], \langle 0, \langle 0,1\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 0,1\rangle)$$

$$+T\big([1,4,5,8,1], \langle 0, \langle 1,0\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 1,0\rangle)$$

$$+T\big([1,4,5,8,1], \langle 0, \langle 1,1\rangle\rangle\big) \cdot \Pr(0)\Pr\langle 1,1\rangle)$$

bad cases

$$+T\big([1,4,5,8,1], \langle 1, \langle 1,0\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 1,0\rangle)$$

$$+T\big([1,4,5,8,1], \langle 1, \langle 0,1\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 0,1\rangle)$$

$$+T\big([1,4,5,8,1], \langle 1, \langle 0,0\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 0,0\rangle)$$

$$+T\big([1,4,5,8,1], \langle 1, \langle 1,1\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 1,1\rangle)$$

good cases

Expected running time of *expectedDemo*

- Summing up c

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

> *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

example

$$
\sum_{R} T([1,4,5,8,9], R) \cdot \Pr(R) = \boxed{\begin{aligned}
&T\big([1,4,5,8,9], \langle 0, \langle 0,0\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 0,0\rangle) \\
+&T\big([1,4,5,8,9], \langle 0, \langle 0,1\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 0,1\rangle) \\
+&T\big([1,4,5,8,9], \langle 0, \langle 1,0\rangle\rangle\big) \cdot \Pr(0)\Pr(\langle 1,0\rangle) \\
+&T\big([1,4,5,8,9], \langle 0, \langle 1,1\rangle\rangle\big) \cdot \Pr(0)\Pr\langle 1,1\rangle)
\end{aligned}} \text{ good cases}
$$

$$
\boxed{\begin{aligned}
+&T\big([1,4,5,8,9], \langle 1, \langle 1,0\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 1,0\rangle) \\
+&T\big([1,4,5,8,9], \langle 1, \langle 0,1\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 0,1\rangle) \\
+&T\big([1,4,5,8,9], \langle 1, \langle 0,0\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 0,0\rangle) \\
+&T\big([1,4,5,8,9], \langle 1, \langle 1,1\rangle\rangle\big) \cdot \Pr(1)\Pr(\langle 1,1\rangle)
\end{aligned}} \text{ bad cases}
$$

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

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

bad cases   good cases

or

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

good cases   bad cases

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

$$= \sum_{\langle x=0,R'\rangle} \underbrace{T(A,\langle x,R'\rangle)\cdot \frac{1}{2}}_{\text{bad cases}} \Pr(R') + \sum_{\langle x=1,R'\rangle} \underbrace{T(A,\langle x,R'\rangle)\cdot \frac{1}{2}}_{\text{good cases}} \Pr(R')$$

or

$$= \sum_{\langle x=0,R'\rangle} \underbrace{T(A,\langle x,R'\rangle)\cdot \frac{1}{2}}_{\text{good cases}} \Pr(R') + \sum_{\langle x=1,R'\rangle} \underbrace{T(A,\langle x,R'\rangle)\cdot \frac{1}{2}}_{\text{bad cases}} \Pr(R')$$

Expected running time of $expectedDemo$

- Summing up over all sequences of random outcomes

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

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

bad cases good cases

or

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

good cases bad cases

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

$$= \frac{1}{2} \sum_{\langle x=0,R' \rangle} \underbrace{(1 + T(A[0 \ldots n-3],\ R')}_{\text{bad cases}} \Pr(R') + \frac{1}{2} \sum_{\langle x=1,R' \rangle} \underbrace{(1 + T(A[0 \ldots n/2-1],\ R')}_{\text{good cases}} \Pr(R')$$

or

$$= \frac{1}{2} \sum_{\langle x=0,R' \rangle} \underbrace{(1 + T(A[0 \ldots n/2-1],\ R')}_{\text{good cases}} \Pr(R') + \frac{1}{2} \sum_{\langle x=1,R' \rangle} \underbrace{(1 + T(A[0 \ldots n-3],\ R')}_{\text{bad cases}} \Pr(R')$$

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

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

$$= \frac{1}{2}\sum_{R'} \underbrace{(1+T(A[0\ldots n-3],\ R')\Pr(R')}_{\text{bad cases}} + \frac{1}{2}\sum_{R'} \underbrace{(1+T(A[0\ldots n/2-1],\ R')\Pr(R')}_{\text{good cases}}$$

or

two cases just differ in the order of elements

$$= \frac{1}{2}\sum_{R'} \underbrace{(1+T(A[0\ldots n/2-1],\ R')\Pr(R')}_{\text{good cases}} + \frac{1}{2}\sum_{R'} \underbrace{(1+T(A[0\ldots n-3],\ R')\Pr(R')}_{\text{bad cases}}$$

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

Expected running time of *expectedDemo*

- Summing up over all sequences of random outcomes

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

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

bad cases — good cases

or

two cases just differ in the order of elements

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

good cases — bad cases

- Replace both cases with

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

Expected running time of $expectedDemo$

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

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

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

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

Expected running time of $expectedDemo$

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

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

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

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

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

Expected running time of *expectedDemo*

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

$$= \frac{1}{2}\sum_{R'} \left(1 + T(A[0 \dots n/2-1], R')\right) \cdot \Pr(R') \quad + \text{second part}$$

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

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

$\mathbb{I}_2$ =all instances of size 2

specific instance of size 2

$$\sum_{R'} T([1,4], R') \cdot \Pr(R') \leq max \left\{ \begin{array}{c} \sum_{R'} T([4,5], R') \cdot \Pr(R') \\ \sum_{R'} T([1,4], R') \cdot \Pr(R') \\ \sum_{R'} T([1,3], R') \cdot \Pr(R') \\ \dots \end{array} \right\}$$

Expected running time of $expectedDemo$

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

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

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

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

$$\leq \frac{1}{2} + \frac{1}{2}\max_{A'\in \mathbb{I}_{n/2}} \sum_{R'} T(A', R')\cdot \Pr(R') \quad + \text{second part}$$

Expected running time of *expectedDemo*

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

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

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

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

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

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

Expected running time of $expectedDemo$

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

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

- Therefore it also holds for A which maximizes this sum

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

- Same recurrence as for $averCaseDemo$
  - expected running time is $O(\log(n))$
- Is it a coincidence that expected time of randomized version is the same as average case of non-randomized version?
  - not in general (depends on randomization)
  - but yes if randomization is a shuffle
    - choose instance randomly with equal probability

Average-case vs. Expected runtime

- Average case runtime and expected runtime are different concepts!

average case	expected
$T^{avg}(n) = \frac{\sum_{I \in \mathbb{I}_n} T(I)}{\lvert \mathbb{I}_n \rvert}$	$T^{exp}(I) = \sum_{\text{outcomes } R} T(I,R) \cdot \Pr(R)$
average over instances	average over random outcomes
usually applied to a deterministic (i.e. not randomized) algorithm	**applied only to a randomized algorithm**

- Sometimes can relate average-case runtime of an algorithm to the expected runtime of its randomized version, but not always

Average-case vs. Expected runtime

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

- Ignoring time needed for the first two lines

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

- To compute average case for an algorithm, can compute the expected runtime of its shuffled version
  - usually easier than computing average case time

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 $0 \le k < n$, 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

	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

$\textit{select}(2) = 20$

- 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
  - *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

j = 6 i = 7

almost done, just swap with pivot v

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

j = 6 **i = 7**

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

Efficient In-Place partition (Hoare)

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

$\leq v$	?	$\geq v$	v

i $\qquad\qquad$ j

- One possible implementation

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

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

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

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

do $j \leftarrow j-1$ **while** $j > 0$ **and** $A[j] > v$

- Simplify the loop bounds

do $i \leftarrow i+1$ **while** $A[i] < v$ // i will not run out of bounds as $A[n-1] = v$

do $j \leftarrow j-1$ **while** $j \geq i$ **and** $A[j] > v$ // j will not run out of bounds as $i \geq 0$

Efficient In-Place partition (Hoare)

partition (A, p)

A: array of size n

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

swap$(A[n-1],\ A[p])$ // put pivot at the end

$i \leftarrow -1, \qquad j \leftarrow n-1, \qquad v \leftarrow A[n-1]$

loop

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

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

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

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

end loop

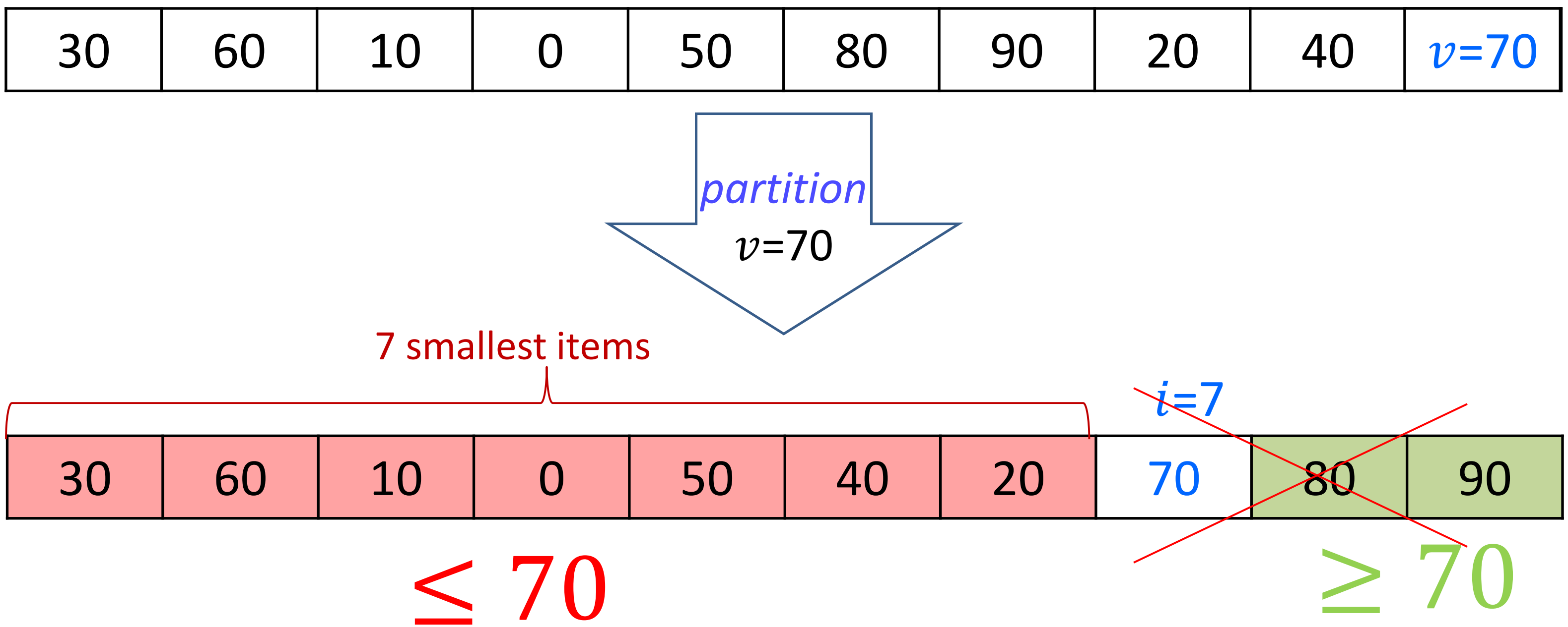
swap$(A[n-1],\ A[i])$ // put pivot in correct position

return i

- Running time is $\Theta(n)$

Quick Select Algorithm

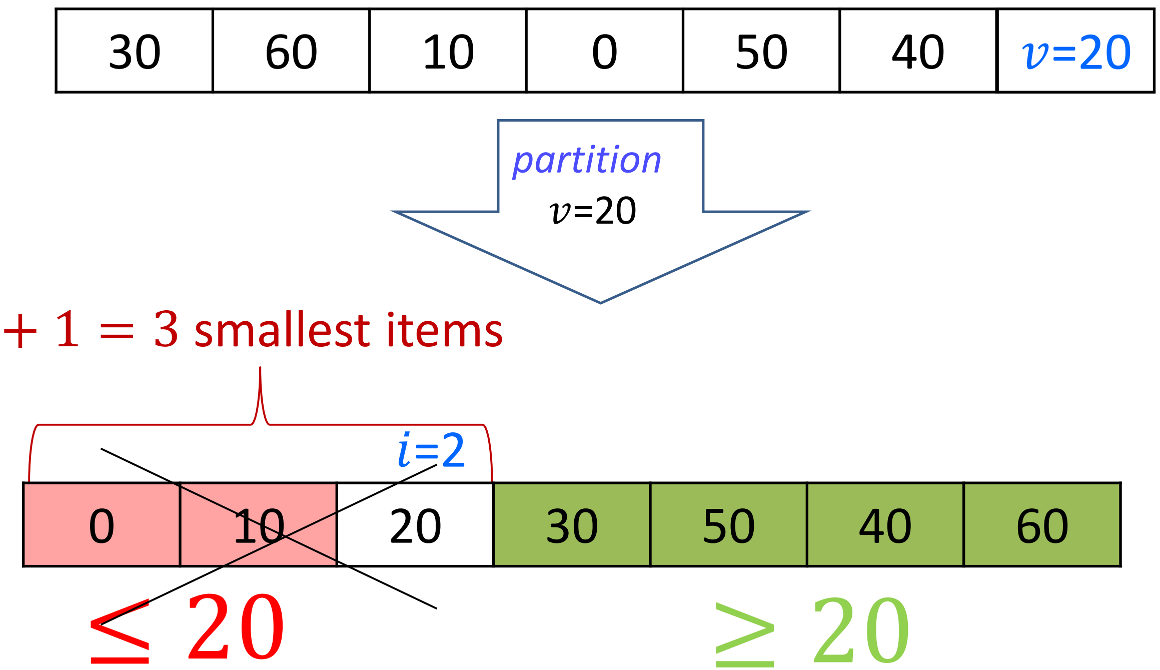
- Find item that would be in $A[k]$ if A was sorted
- Similar to quick-sort, but recurse only on one side ("quick-sort with pruning")
- Example: select$(k = 4)$



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

Quick Select Algorithm

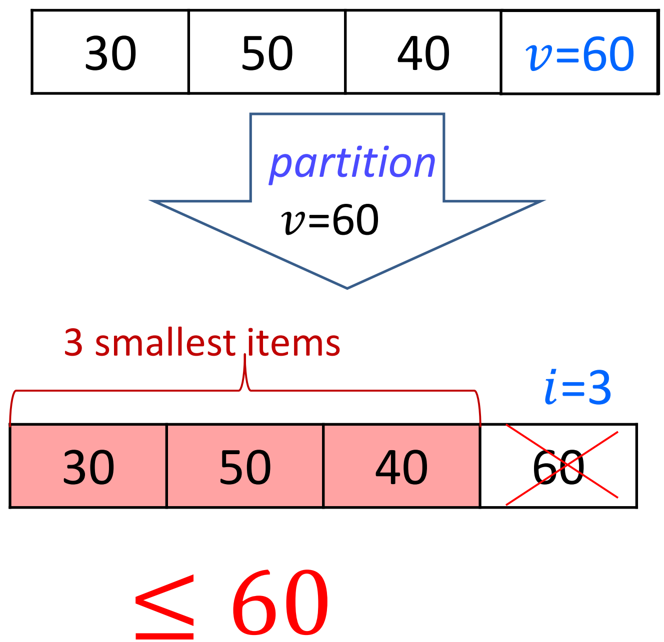
- Example continued: select($k = 4$)



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

Quick Select Algorithm

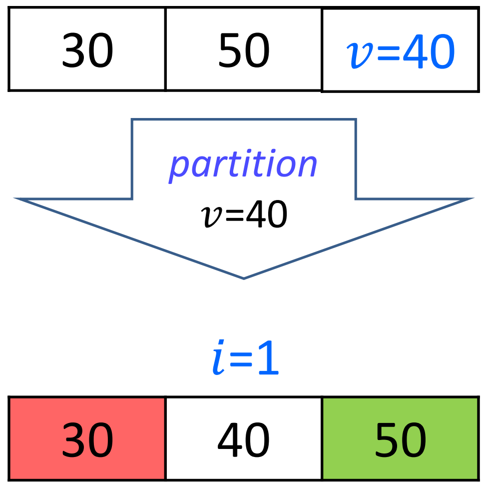
- Example continued: select($k = 1$)



- $i > k$, 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. $0 \le k < n$

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

- **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 $k = 0$
  - 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

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

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

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

after 1 expansion

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

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

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

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

Average-Case Analysis of *QuickSelect*

- Runtime depends only on the order of the elements
- Therefore, can use sorting permutations

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

- Can show (complicated) that average-case runtime is $\Theta(n)$
    - better than the worst case runtime, $\Theta(n^2)$
- Therefore, can create a better algorithm in practice by randomizing *QuickSelect*
    - no more bad instances
    - if randomization is done with shuffling, the expected time *randomizedQuickSelect* is the same as average case runtime of non-randomized *QuickSelect*
        - proved earlier
    - expected runtime is easier to derive
        - randomization leads to an easier analysis of average-case

Randomized QuickSelect: Shuffling

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

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

- $random(n)$ returns integer uniformly sampled from $\{0, 1, 2, \ldots, n-1\}$
- 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*
  - $\Theta(n)$

Randomized QuickSelect Algorithm

- **Second idea**: change pivot selection

RandomizedQuickSelect(A, k)

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

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

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

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

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

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

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

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

- Advantage: no need to spend time shuffling
- It is possible to prove that *RandomizedQuickSelect* has the same expected runtime as *quickSelectShuffled* (no details)
- So expected runtime of *RandomizedQuickSelect* is $\Theta(n)$
- But it is actually not as hard to derive expected time for *RandomizedQuickSelect*

Randomized QuickSelect: Analysis

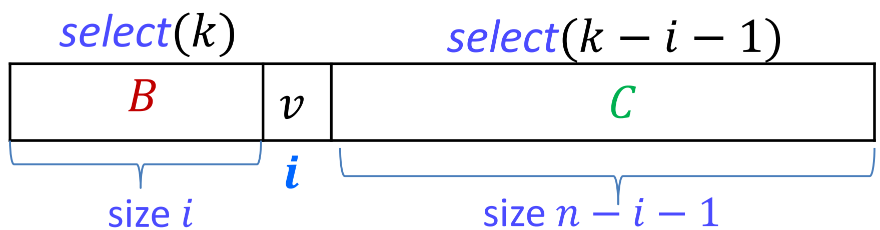
$RandomizedQuickSelect(A, k)$

$p \leftarrow random(A.size)$

$i \leftarrow partition(A, p)$

$\ldots$

- 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

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

- For *expectedDemo*

$$T^{exp}(n) = \max_{A\in\mathbb{I}_n} \sum_R T(A,R)\Pr(R)$$

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

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

- First, let us work on $\sum_R T(A,k,R)\Pr(R)$

Randomized QuickSelect: Analysis

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

$$\Pr(i) = \frac{1}{n}$$

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

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

$$= \frac{1}{n} \sum_{R=\langle i,R'\rangle} \boxed{T(A,k,\langle i,R'\rangle)\Pr(R')}$$

$$= \underbrace{\sum_{R=\langle 0,R'\rangle} \square + \sum_{R=\langle 1,R'\rangle} \square + \cdots + \sum_{R=\langle k-1,R'\rangle} \square}_{i<k:\ \text{recurse on } C} + \underbrace{\sum_{R=\langle k,R'\rangle} \square}_{\text{base case}} + \underbrace{\sum_{R=\langle k+1,R'\rangle} \square + \cdots + \sum_{R=\langle n-1,R'\rangle} \square}_{i>k:\ \text{recurse on } B}$$

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

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

Randomized QuickSelect: Analysis

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

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

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

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

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

$$= \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} n\Pr(R') + \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} T(C,k-i-1,R')\Pr(R') + \text{the rest}$$

Randomized QuickSelect: Analysis

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

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

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

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

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

$$= \frac{n}{n}\sum_{i=0}^{k-1}\overbrace{\boxed{\sum_{R'} \Pr(R')}}^{=1} + \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} T(C,k-i-1,R')\Pr(R') + \text{the rest}$$

$$= \quad k \quad + \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} T(C,k-i-1,R')\Pr(R') \quad + \text{the rest}$$

Randomized QuickSelect: Analysis

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

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

some instance C of size $n-i-1$
some integer $k-i-1 \in \{0, \ldots k-1\}$

$$= k + \frac{1}{n}\sum_{i=0}^{k-1} \boxed{\sum_{R'} T(C, k-i-1, R')\Pr(R')} + \text{the rest}$$

max over **all** instances D of size $n-i-1$
and **all** integers $\in \{0, \ldots k-1\}$

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

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

Randomized QuickSelect: Analysis

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

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

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

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

apply same steps as to first sum

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

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

Randomized QuickSelect: Analysis

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

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

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

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

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

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

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

- Expected runtime for *RandomizedQuickSelect* satisfies

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

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

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

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

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

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

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

Summary of Selection

- Thus expected runtime of *RandomizedQuickSelect* is $O(n)$
  - it is also $\Theta(n)$, since the best case is $O(n)$
    - have to partition the array
- Therefore *quickSelectShuffled* ehas expected runtime $O(n)$
  - no details
- Therefore *quickSelect* has average case runtime $O(n)$
- *RandomizedQuickSelect* is generally the fastest implementation of selection algorithm
- There is a selection algorithm with worst-case running time $\mathrm{O}(n)$
  - 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
- He also used them to *sort* based on partitioning

QuickSort(A)

Input: array A of size n

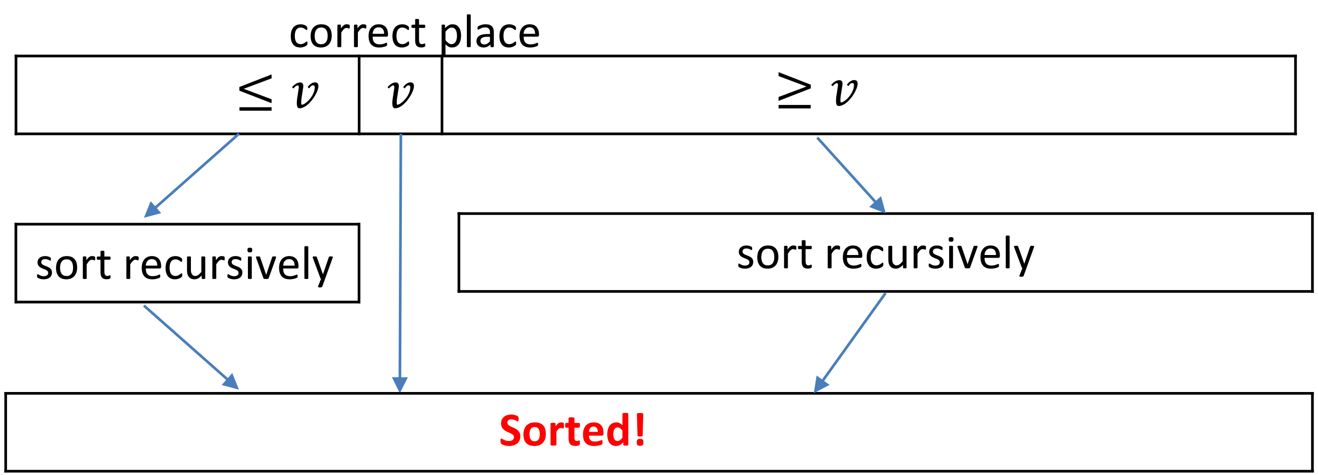
if $n \le 1$ **then return**

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

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

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

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



QuickSort

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 number of comparisons on size n array
  - running time is Θ(number of comparisons)
- If we know pivot-index i, then $T(n) = n + T(i) + T(n-i-1)$
- 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

$RandomizedQuickSort(A)$

…

$p \leftarrow random(A.size)$

…

- 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*

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

Randomized QuickSort: Expected Runtime

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

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

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

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

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

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

Randomized QuickSort

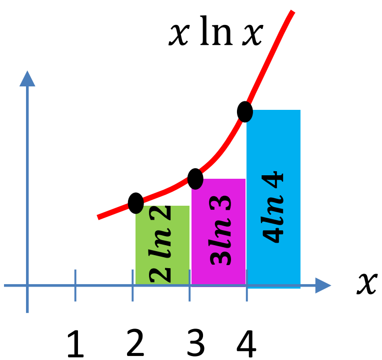
$$T^{exp}(n) \le n + \frac{2}{n}\sum_{i=2}^{n-1} T^{exp}(i)$$

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

by induction hypothesis

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

- 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

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

- Claim $T^{exp}(n) \leq 2n\ln n$ for all $n \geq 0$
- Proof (by induction on n):
  - $T^{exp}(0) = T^{exp}(1) = 0$
  - Suppose true for $2 \leq m < n$
  - Let $n \geq 2$:

$$\sum_{i=2}^{n-1} i\ln i \leq \frac{1}{2}n^2\ln n - \frac{1}{4}n^2$$

by induction hypothesis

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

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

- Expected running time of *RandomizedQuickSort* is $O(n\log n)$
- 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*$(A,\ l,\ r)$

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

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

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

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

else

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

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

InsertionSort(A)

- This is often the most efficient sorting algorithm in practice
  - 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

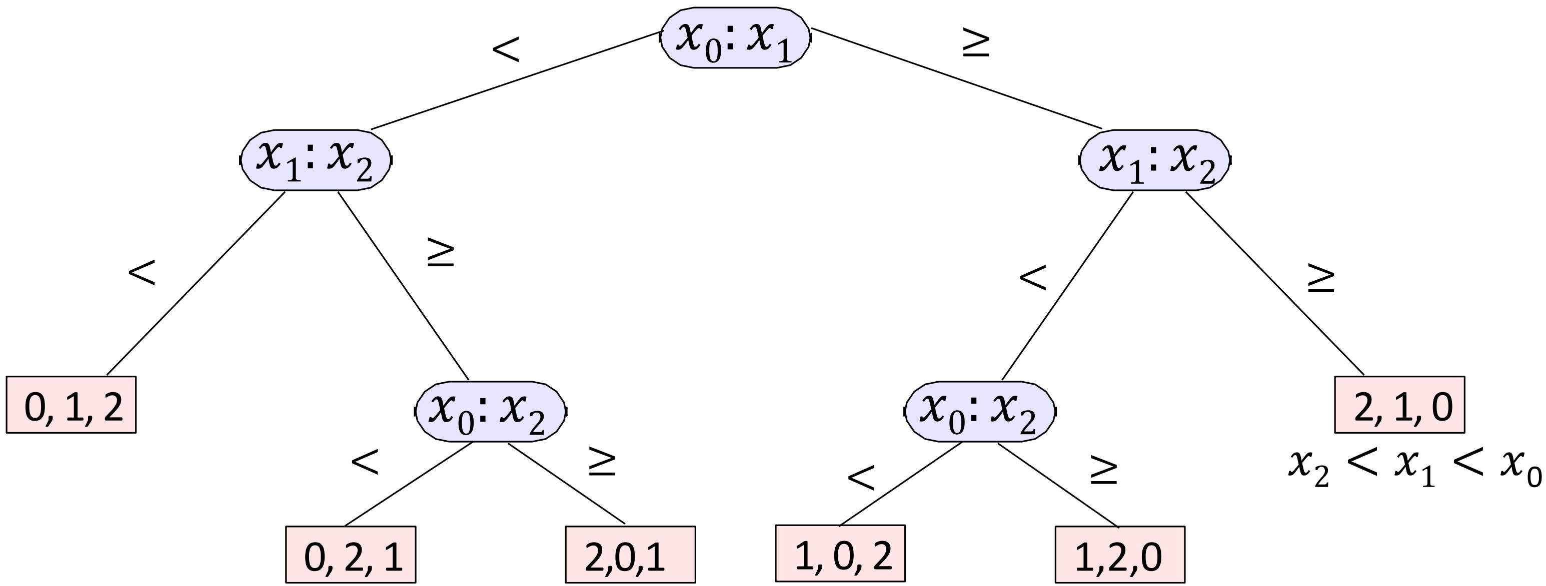
- All sorting algorithms seen so far are in the comparison model
- In the *comparison model* data can only be accessed in two ways
    - comparing two elements
        - $A[i] \leq A[j]$
    - moving elements around (e.g. copying, swapping)
- This makes very few assumptions on the things we are sorting
    - 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 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

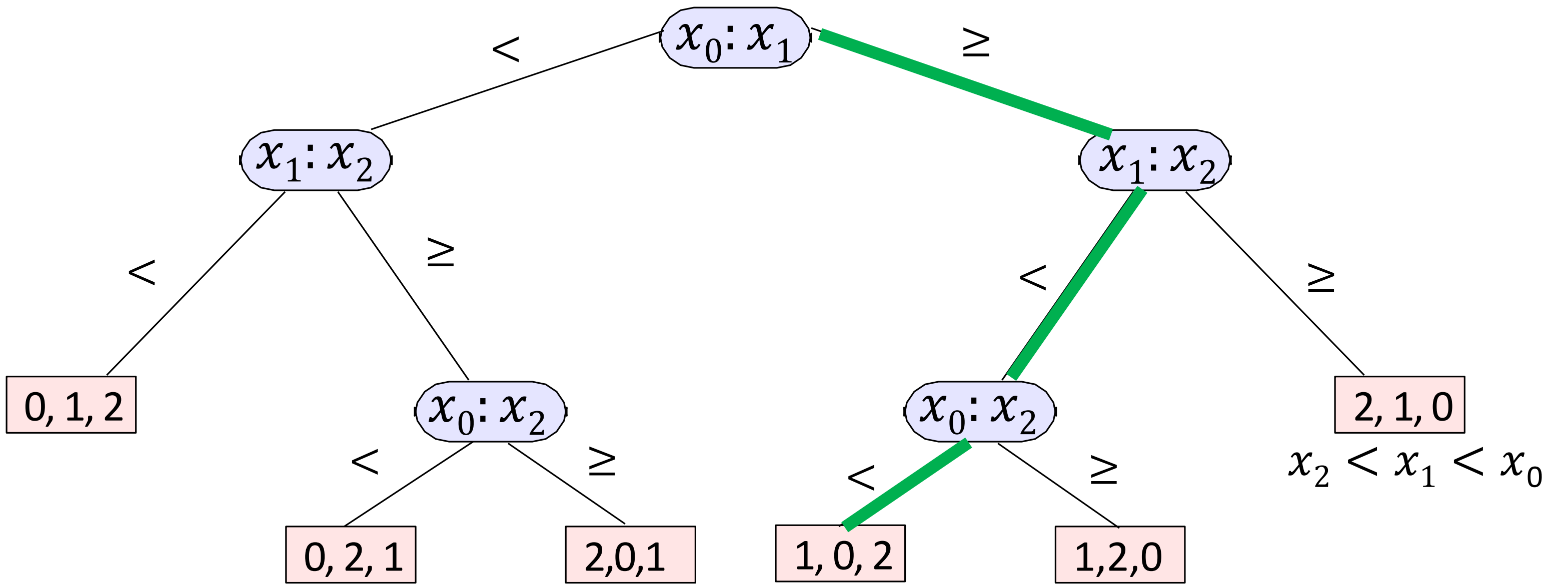
- Decision tree for a concrete comparison based algorithm for sorting 3 elements



- 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: Sorting Example

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

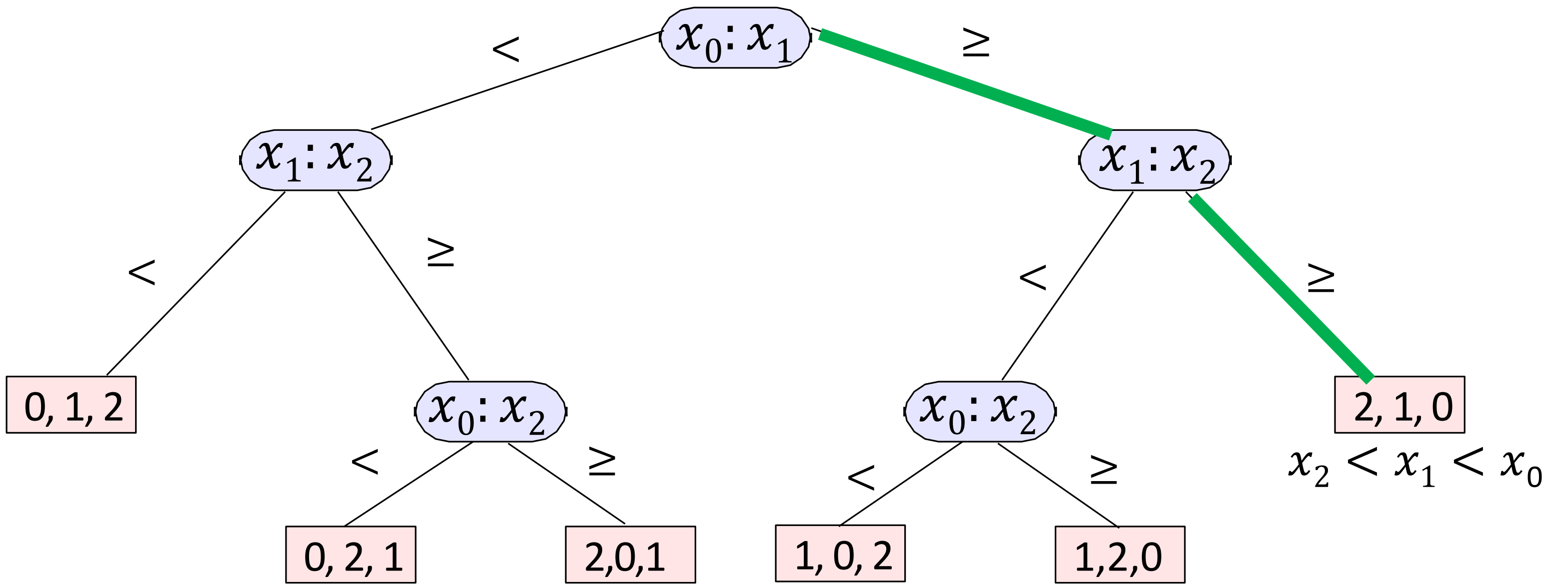


$x_1 = 2 \leq x_0 = 4 \leq x_2 = 7$

3 comparisons

Decision Tree: Sorting Example

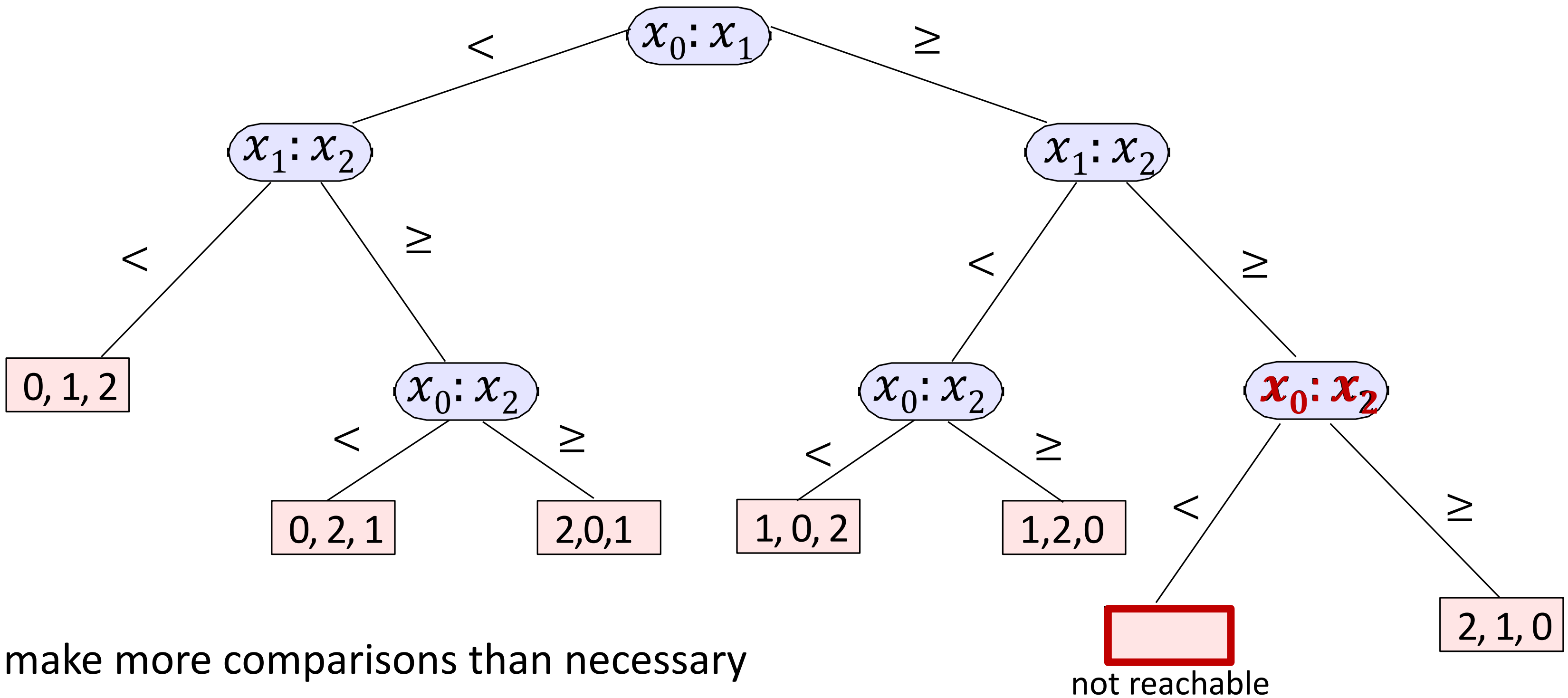
$x_0 = 8, x_1 = 7, x_2 = 7$



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

2 comparisons

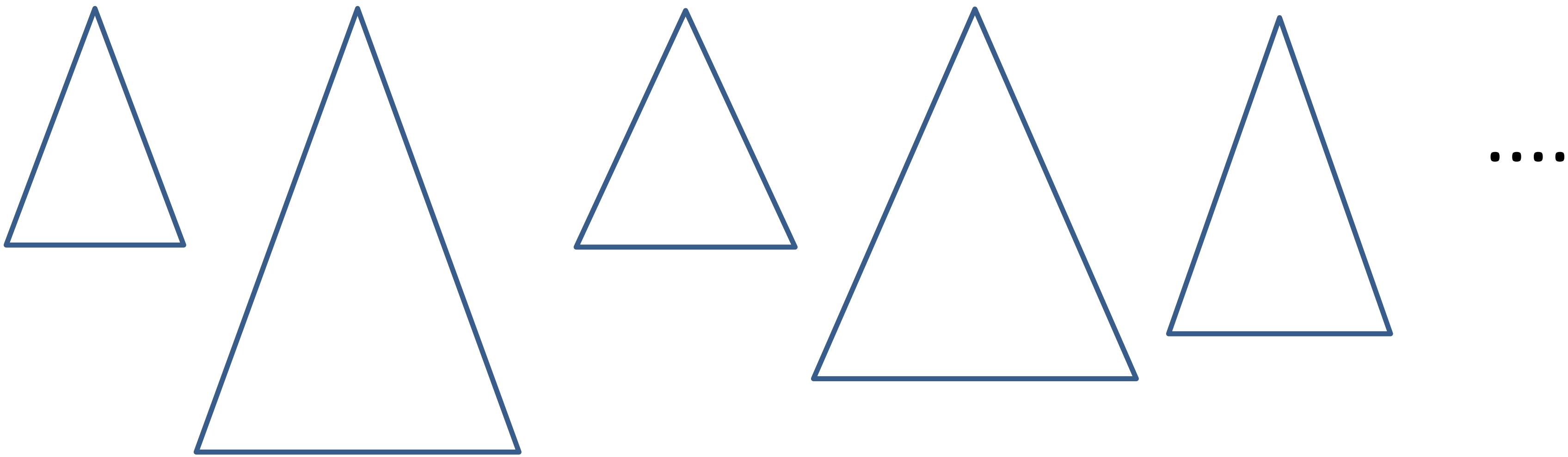
Decision Tree



- 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 $c \log n$
  - which is $\Omega(n \log n)$

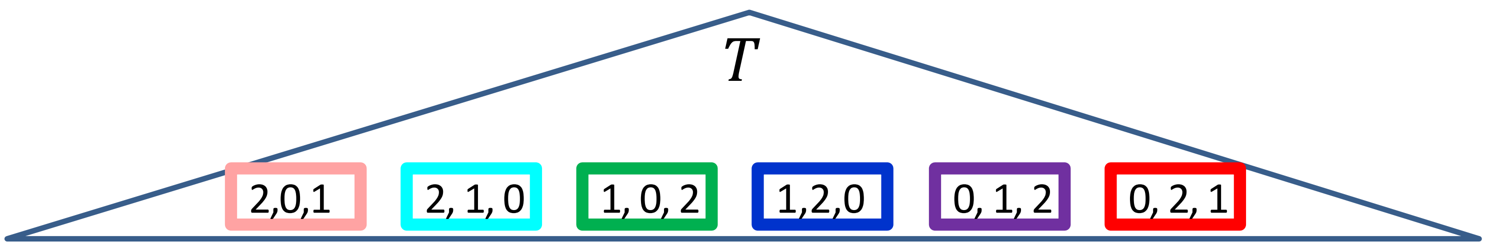
Lower bound for sorting in the comparison model

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

Proof:

- Let *SortAlg* be ***any*** comparison based sorting algorithm
- Since algorithm 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 all arrays consisting of distinct integers in $\{1, \ldots, n\}$
- $|S_n| = n!$
- Let π_x denote the sorting permutation of $x \in S_n$
- When running 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 $\pi_x \neq \pi_y$
- Thus we determined $n!$ instances which must go to distinct leaves
- Therefore, the 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

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

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

$\square$

- 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 $\{1, \dots, n\}$
  - 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

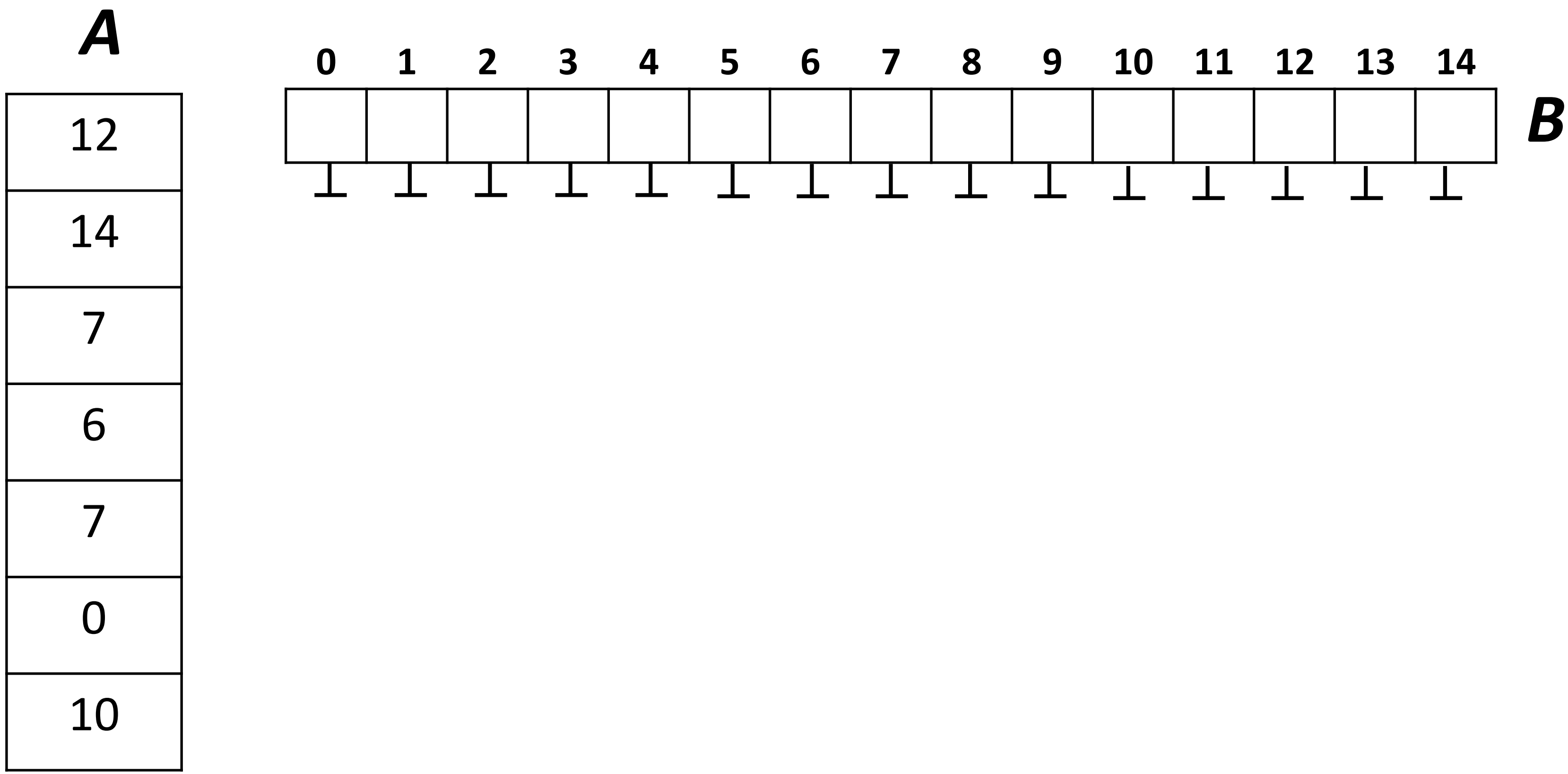
- Sort without comparing items to each other
- **Non-comparison based sorting is less general than comparison based sorting**
- In particular, we need to make assumptions about items we sort
  - unlike in comparison based sorting, which sorts any data, as long as it can be compared
- Will assume we are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

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

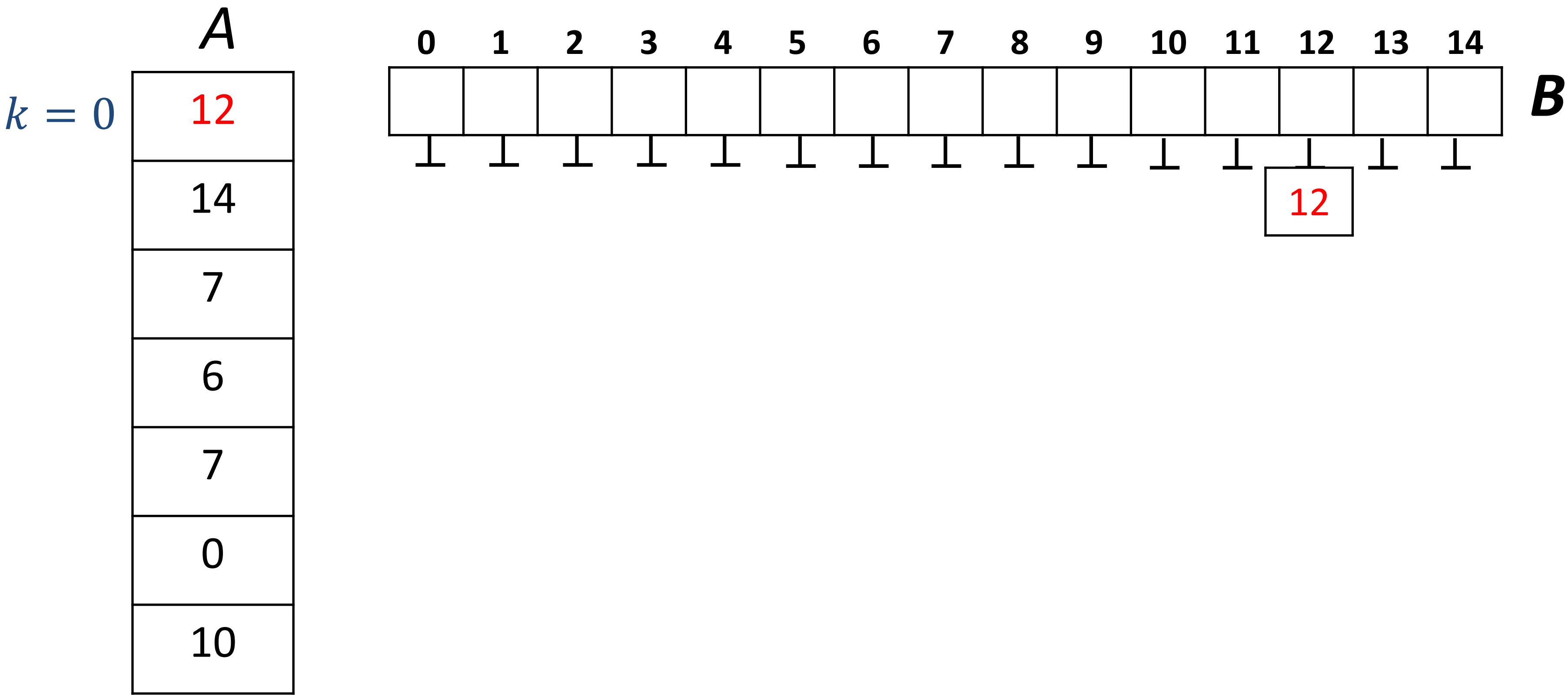
Bucket Sort

- Suppose all keys in A are integers in range $[0, \ldots, L-1]$
- How would you sort if L is not too large?
- 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 $L = 15$



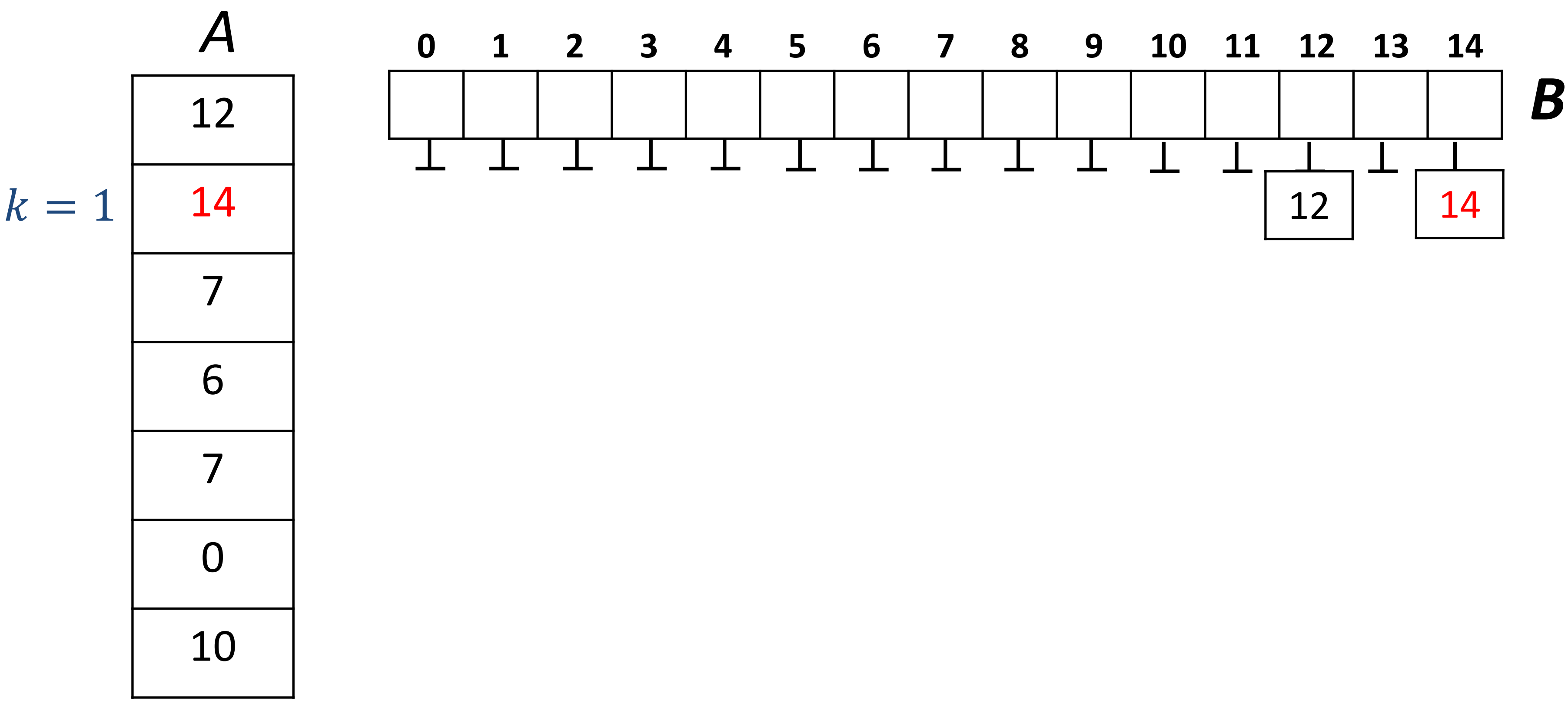
Bucket Sort

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



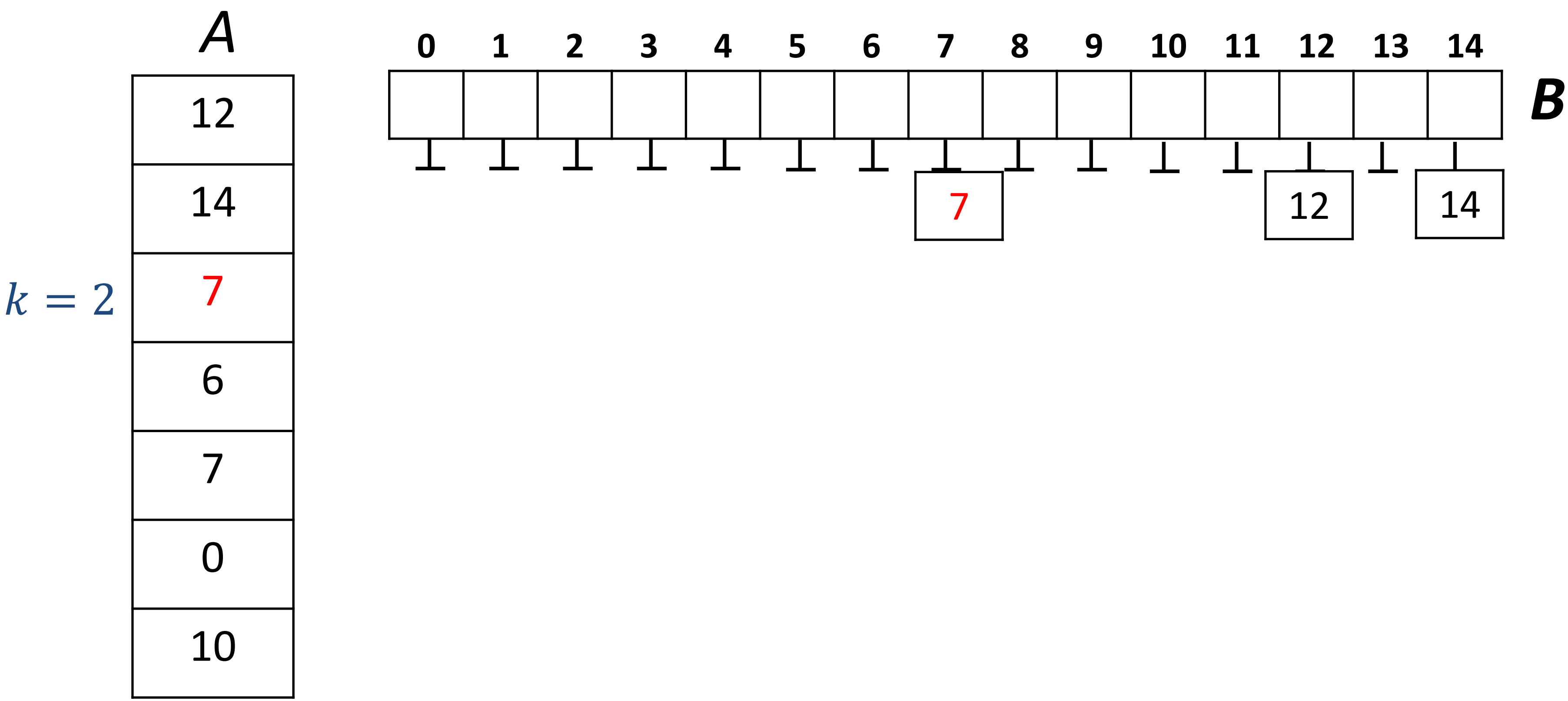
Bucket Sort

- 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 $L = 15$



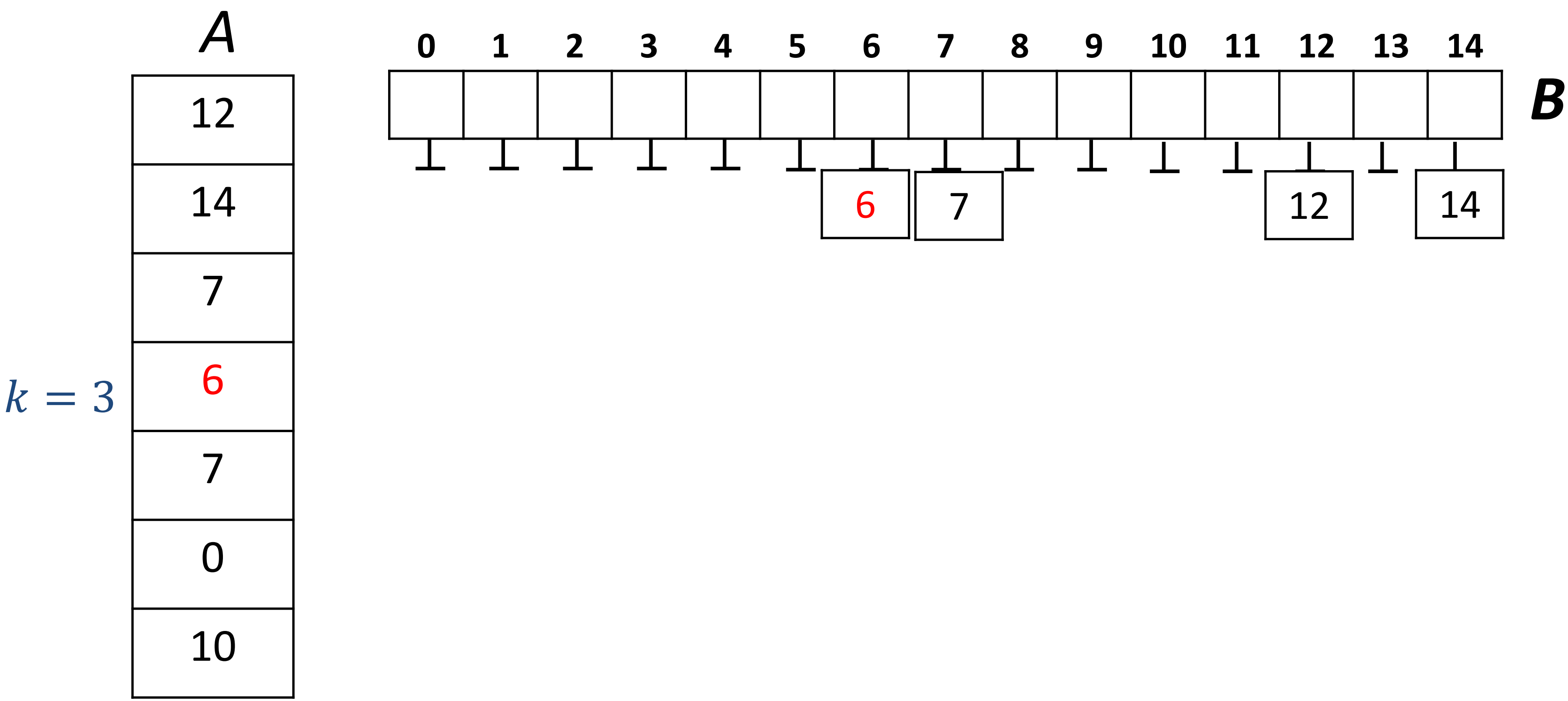
Bucket Sort

- 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 $L = 15$



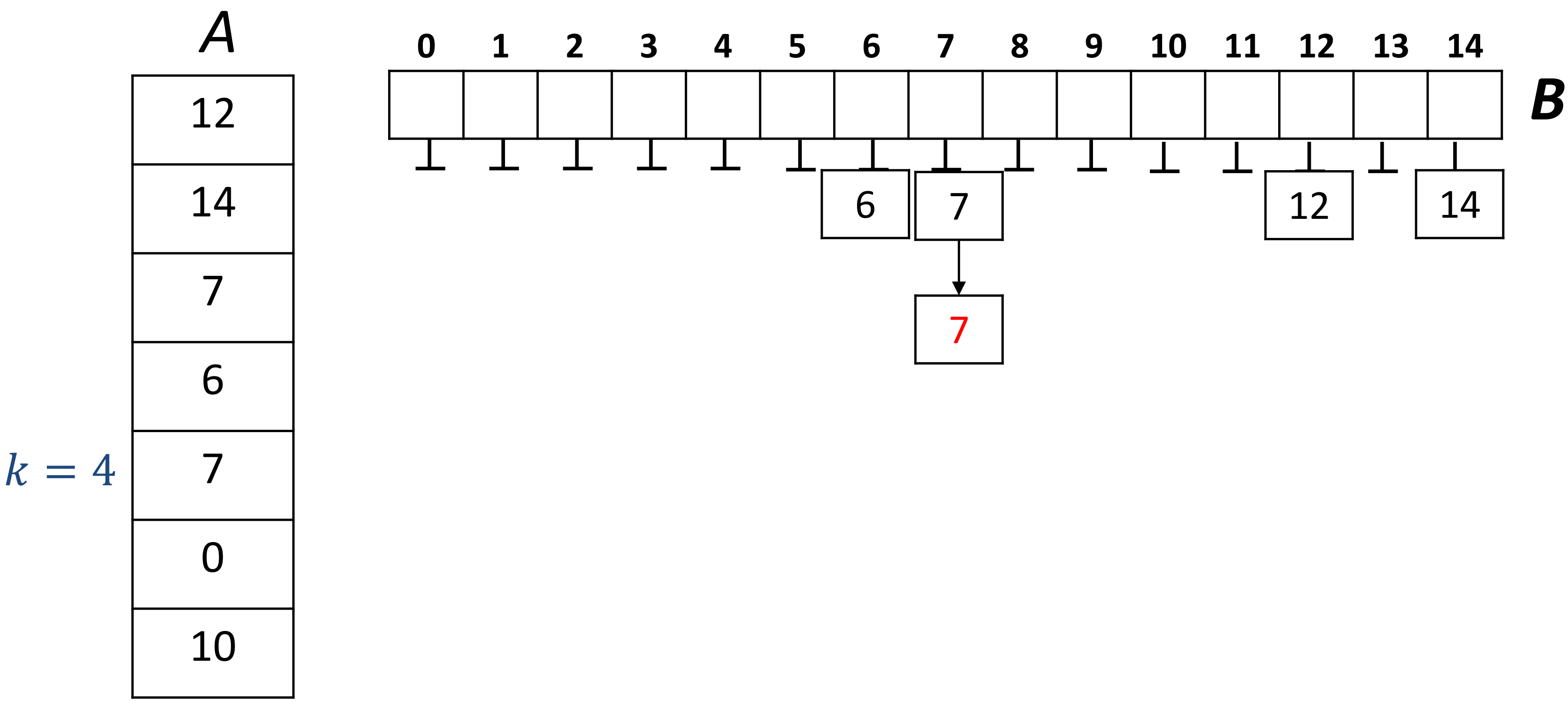
Bucket Sort

- 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 $L = 15$



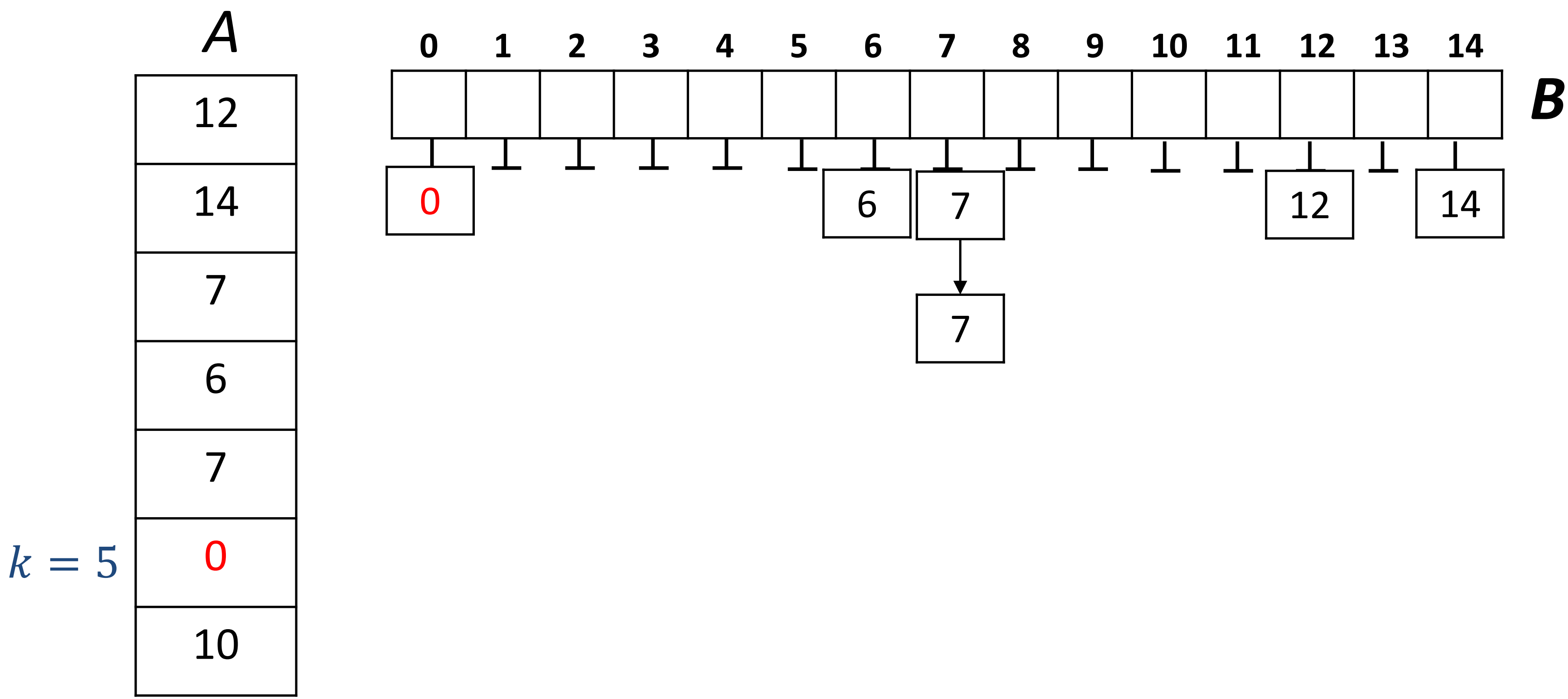
Bucket Sort

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



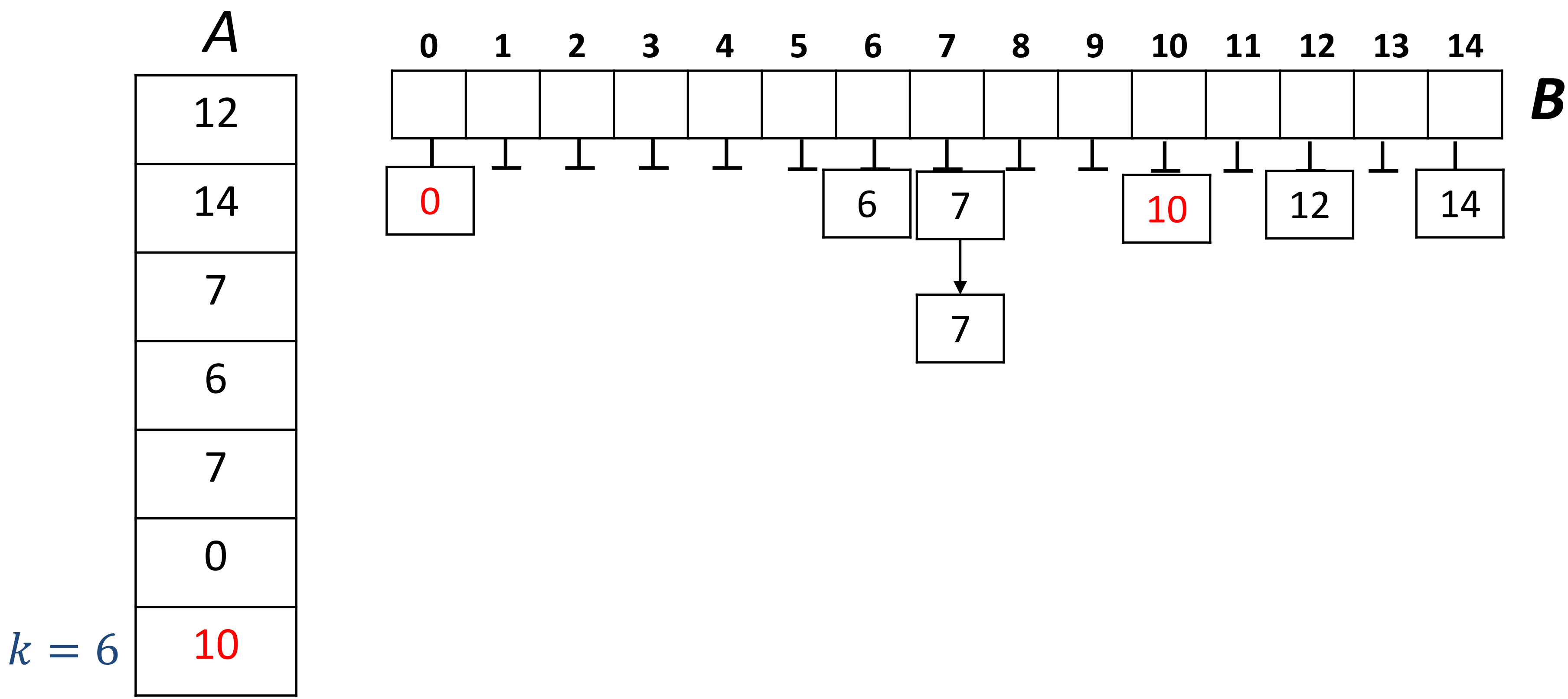
Bucket Sort

- 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 $L = 15$



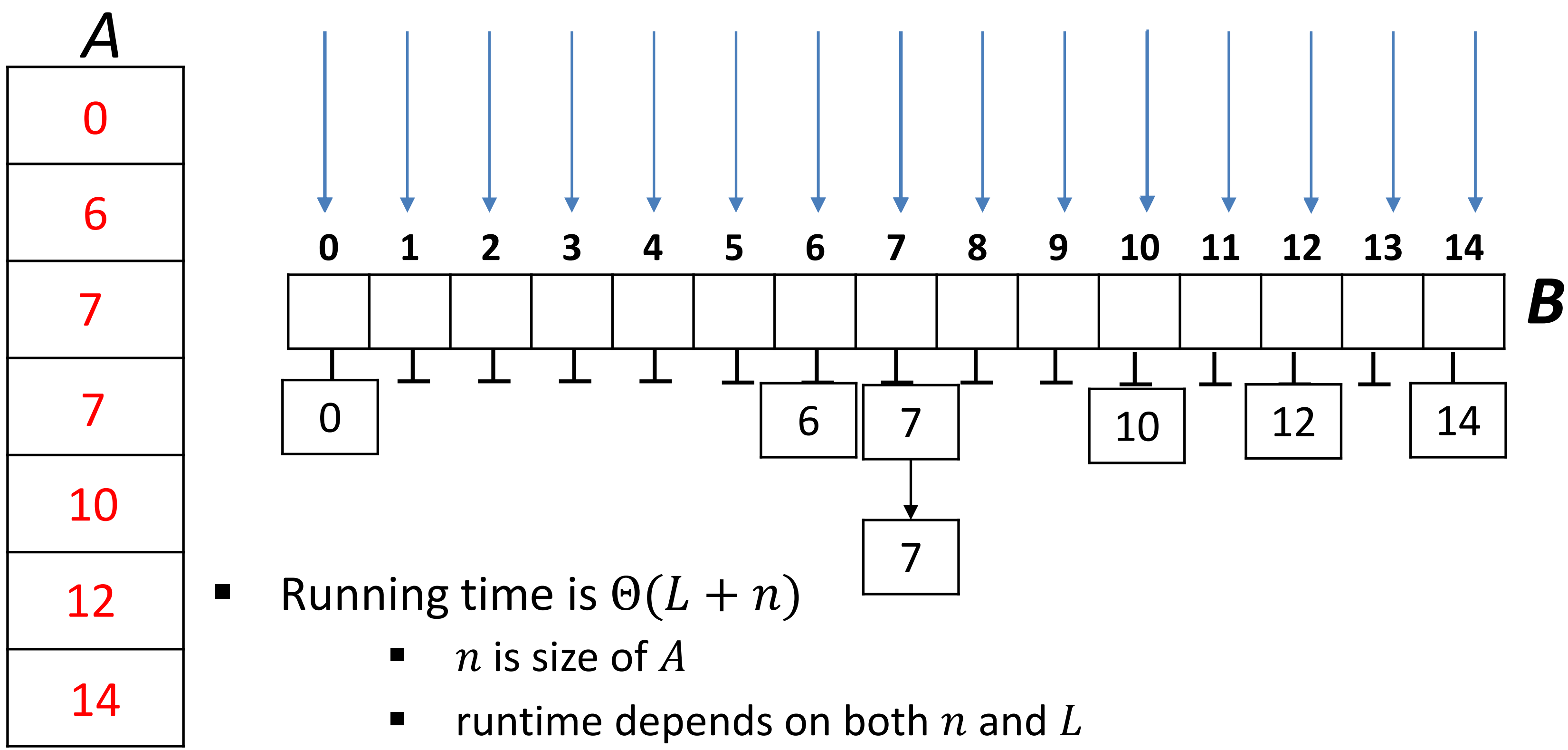
Bucket Sort

- 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



- Running time is $\Theta(L+n)$
  - n is size of A
  - runtime depends on both n and L

Digit Based Non-Comparison-Based Sorting

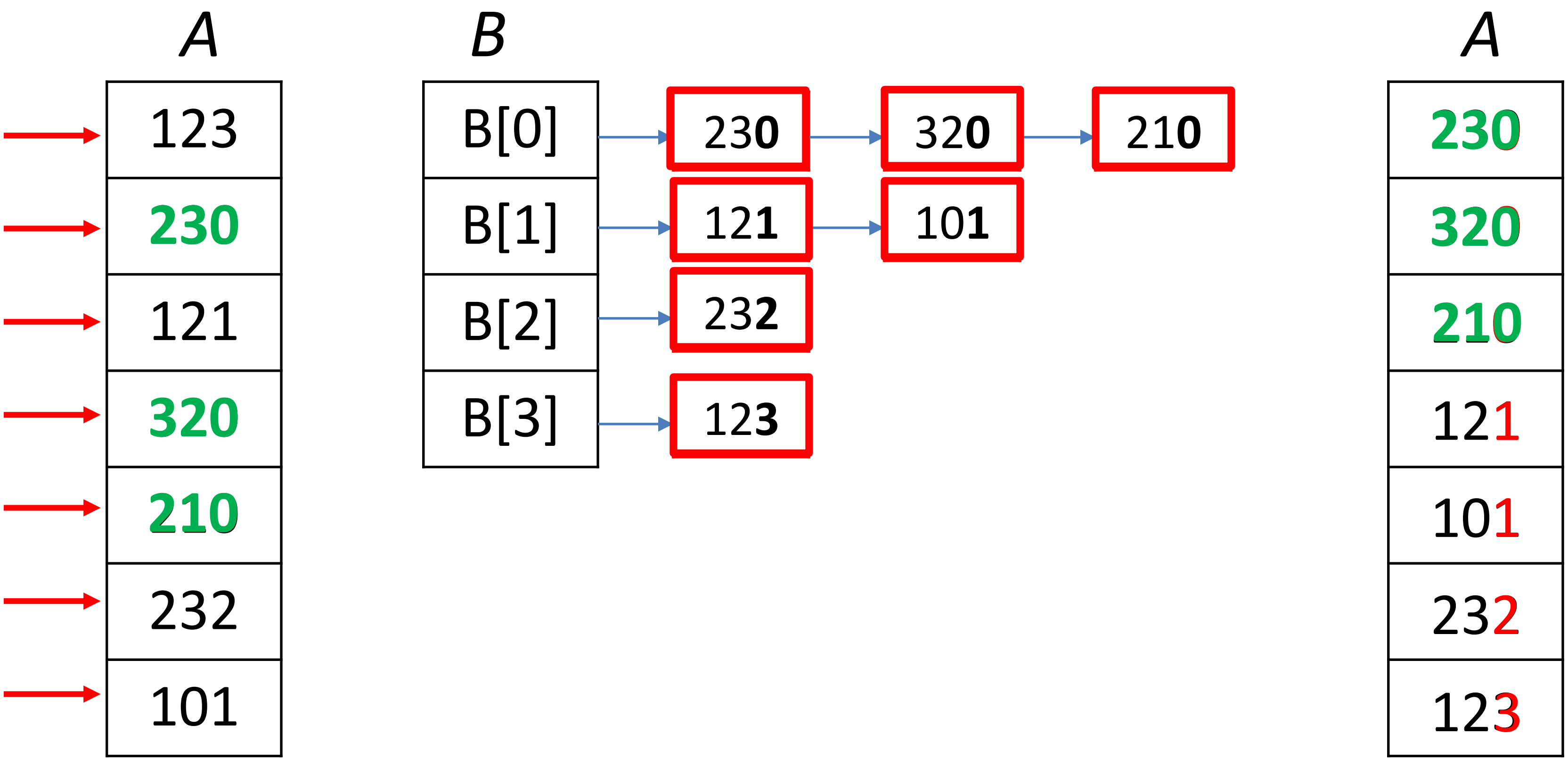
- Running time of bucket sort is $\Theta(L+n)$
  - n is size of A
  - L is range $[0, L)$ of integers in A
- What if L is much larger than n?
  - i.e. A has size 100, range of integers in A is $[0, \dots, 99999]$
- Assume at most m digits in any key
  - pad with leading 0s

123	230	021	320	210	232	101

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

Bucket Sort on Last Digit

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



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

Base R number representation

- Number of distinct digits gives the number of buckets R
- Useful to control number of buckets
  - larger R means less digits (less iterations), but more work per iteration (larger bucket array)
  - may want exactly 2, or 4, or even 128 buckets
- Can do so with base R representation
  - digits go from 0 to $R-1$
  - R buckets
  - numbers are in the range $\{0, 1, \ldots, R^m - 1\}$
- 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 $A[i{+}{+}]$

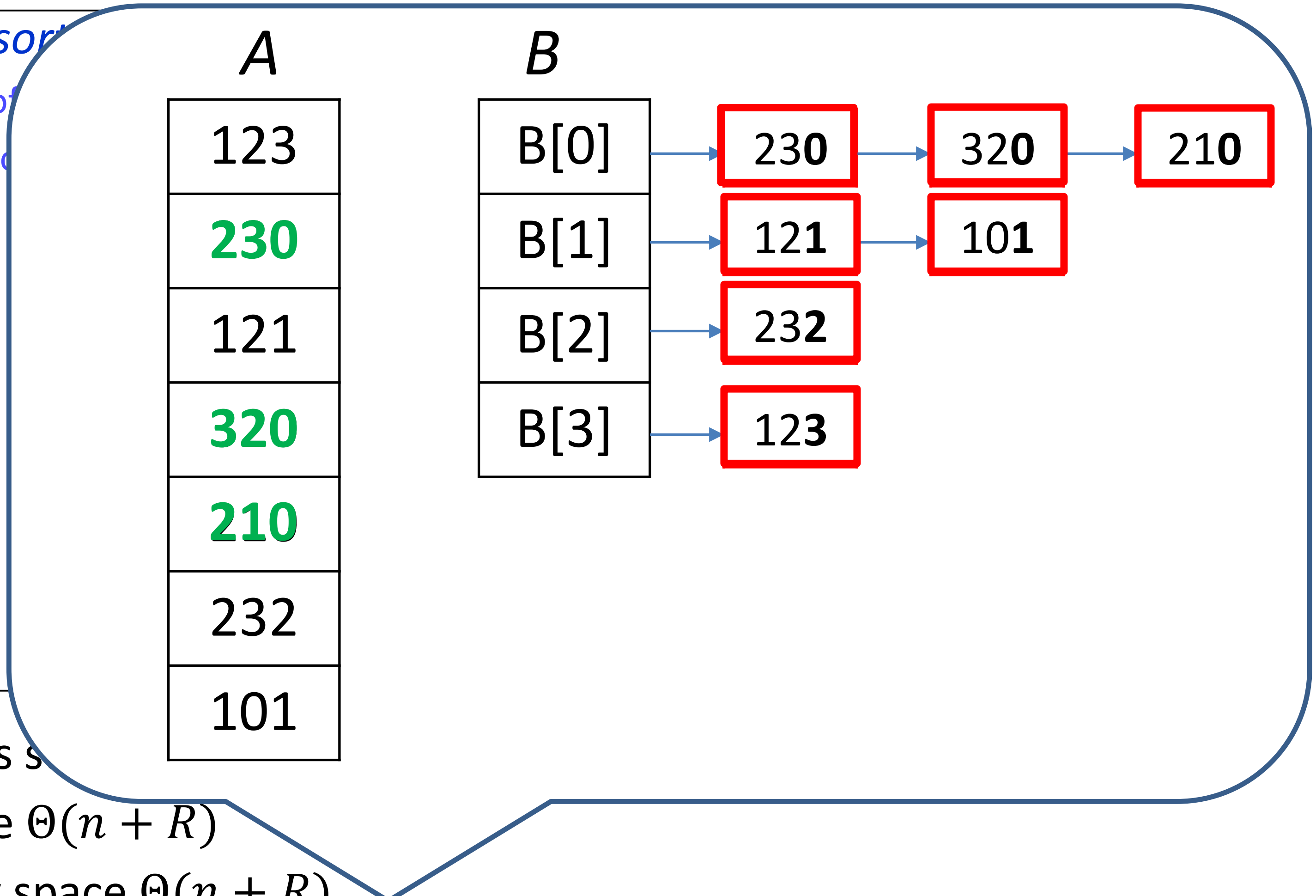
- Sorting is stable: equal items stay in original order
- Run-time $\Theta(n+R)$
- Auxiliary space $\Theta(n+R)$
  - $\Theta(R)$ for array B, and linked lists are $\Theta(n)$

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

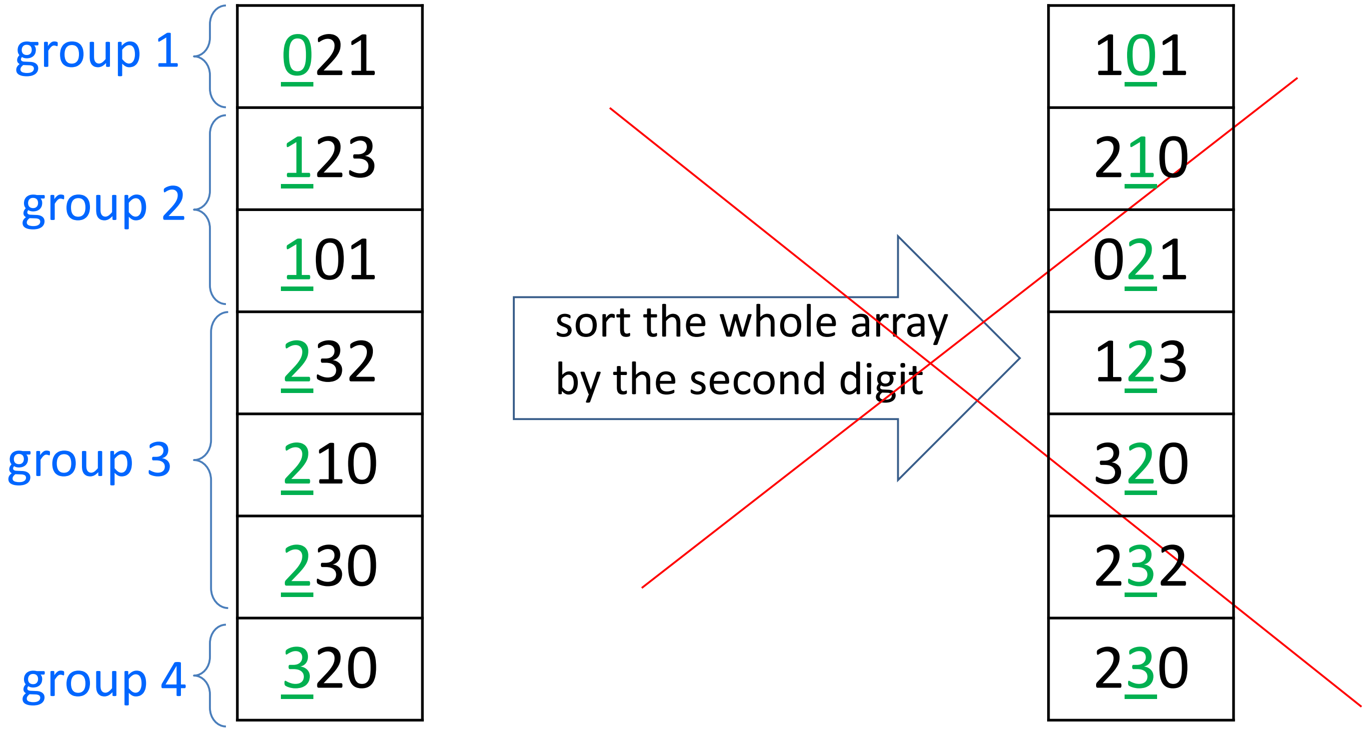
MSD-Radix-Sort

- 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

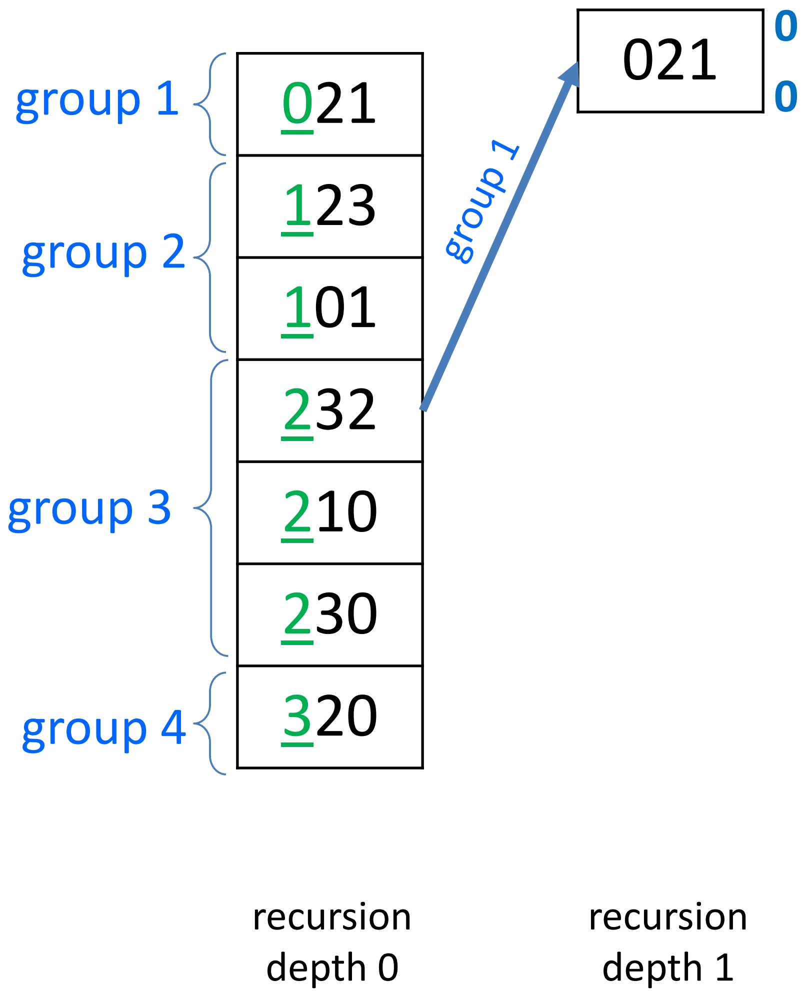
- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit



- 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

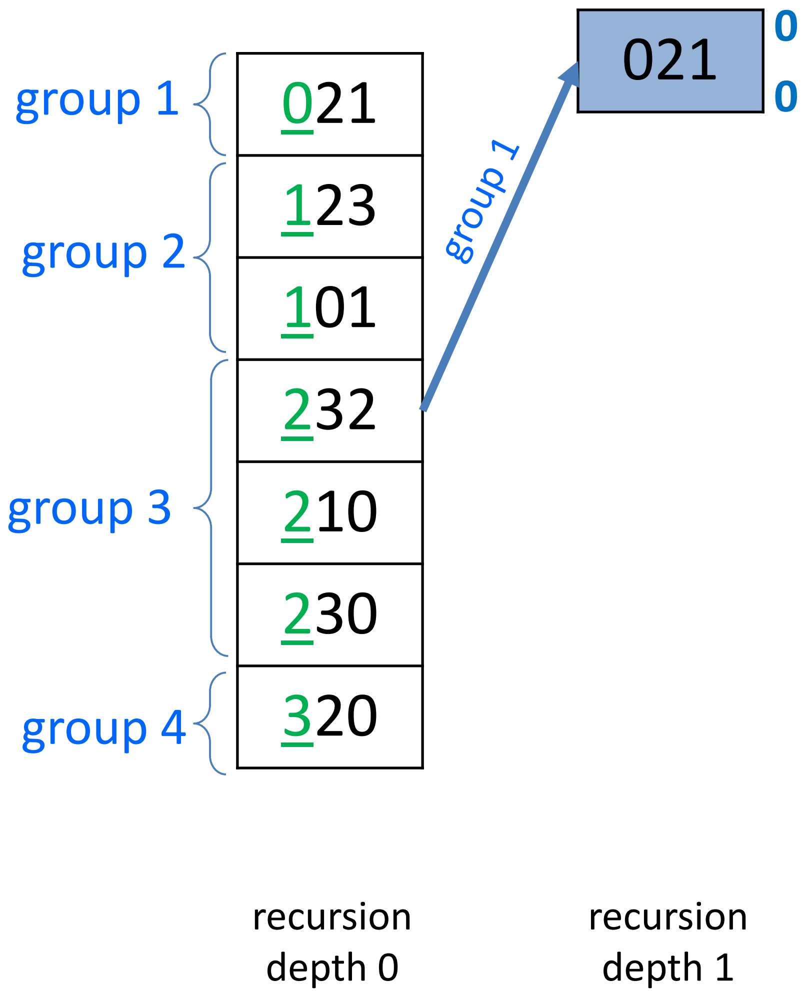
MSD-Radix-Sort

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



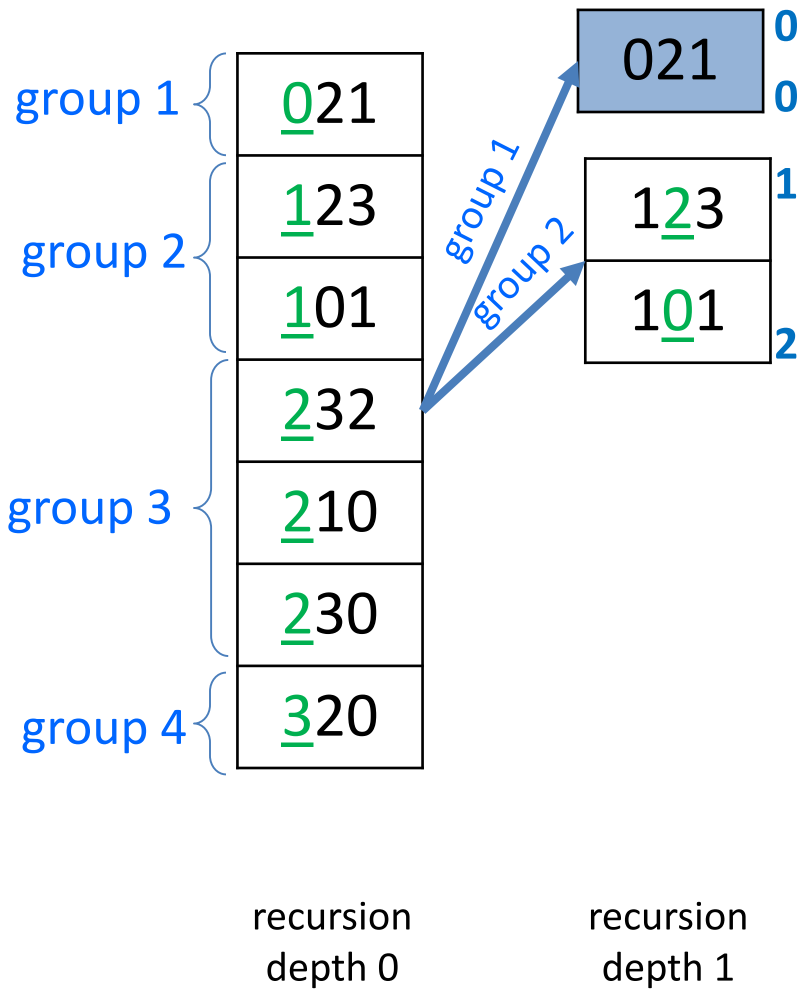
MSD-Radix-Sort

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



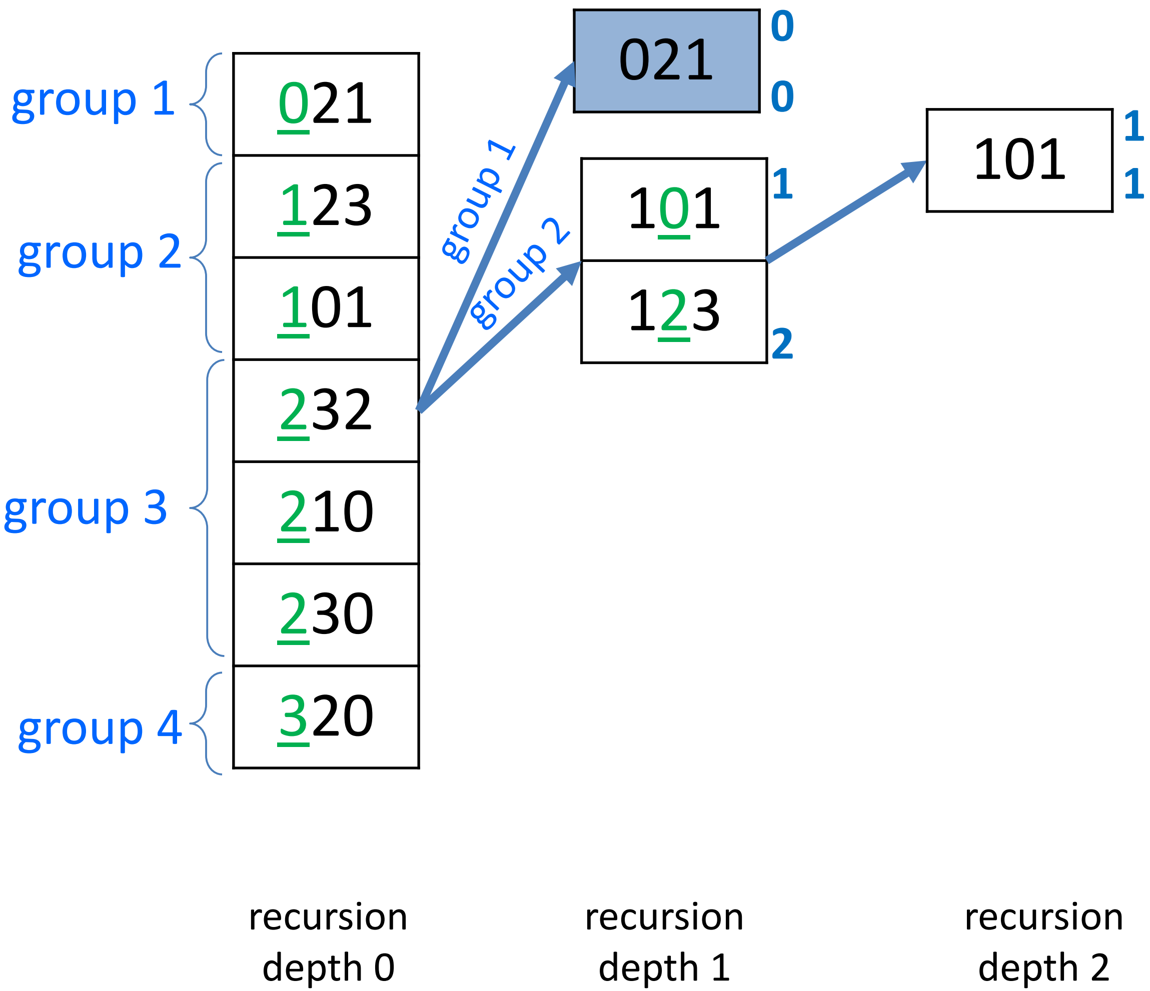
MSD-Radix-Sort

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



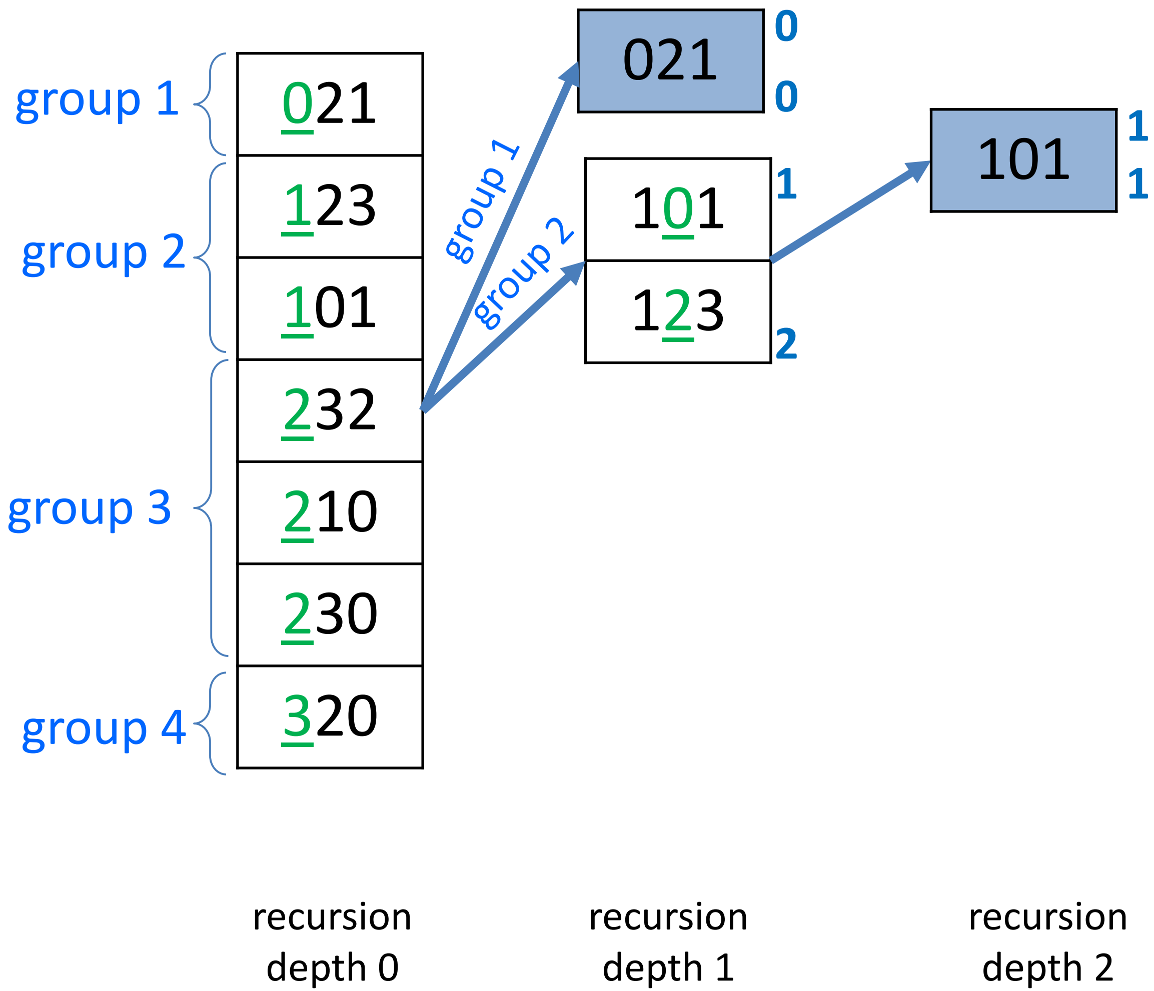
MSD-Radix-Sort

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



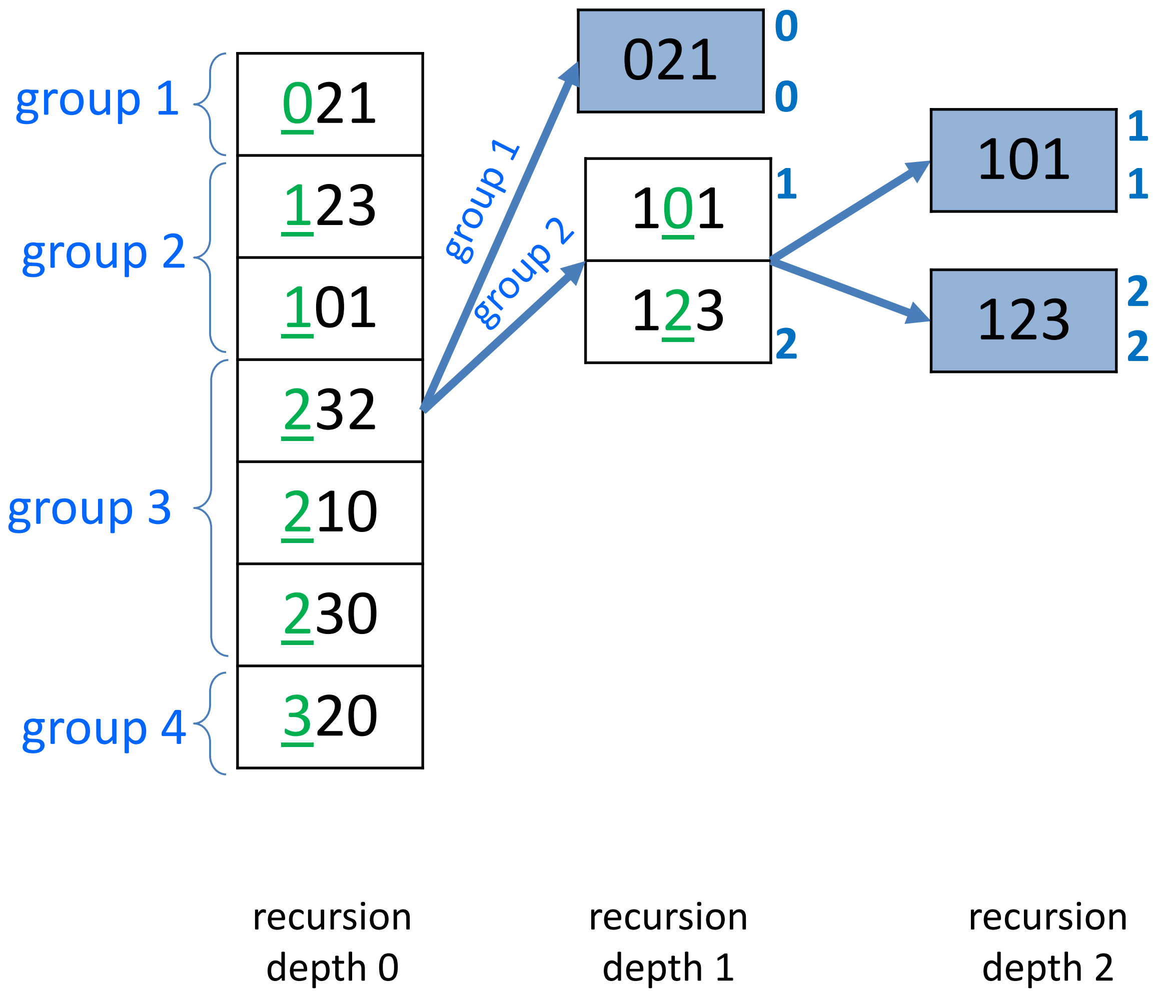
MSD-Radix-Sort

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



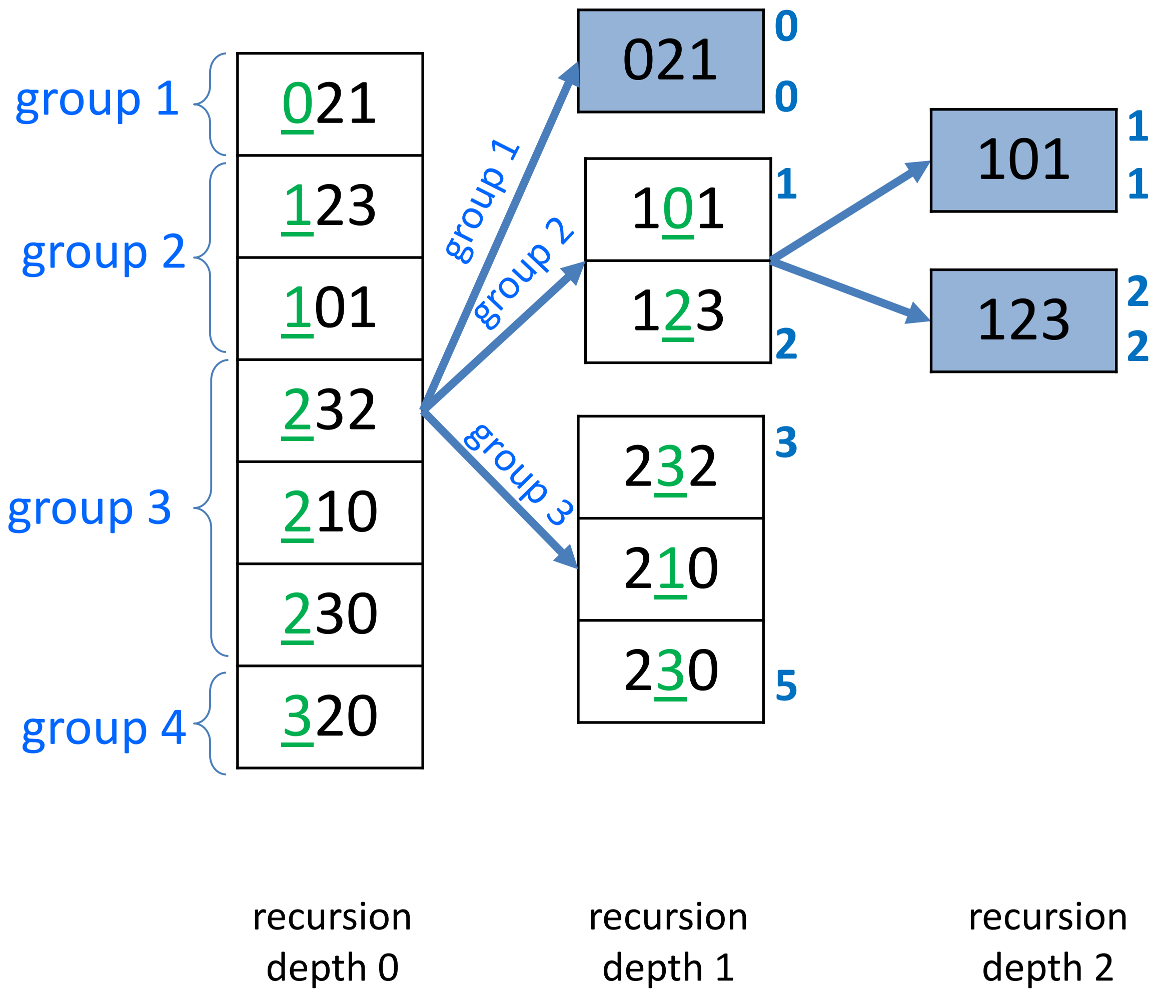
MSD-Radix-Sort

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



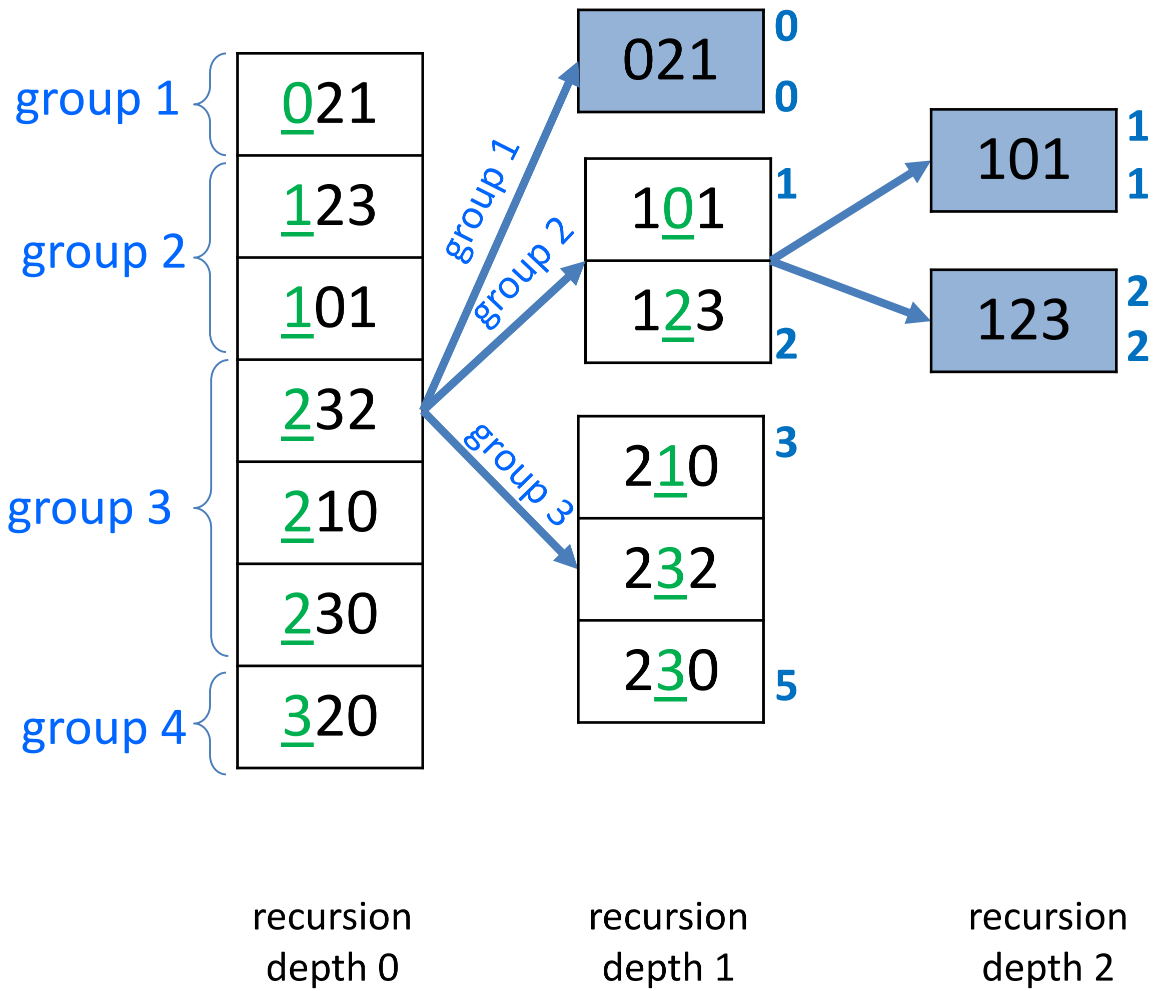
MSD-Radix-Sort

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



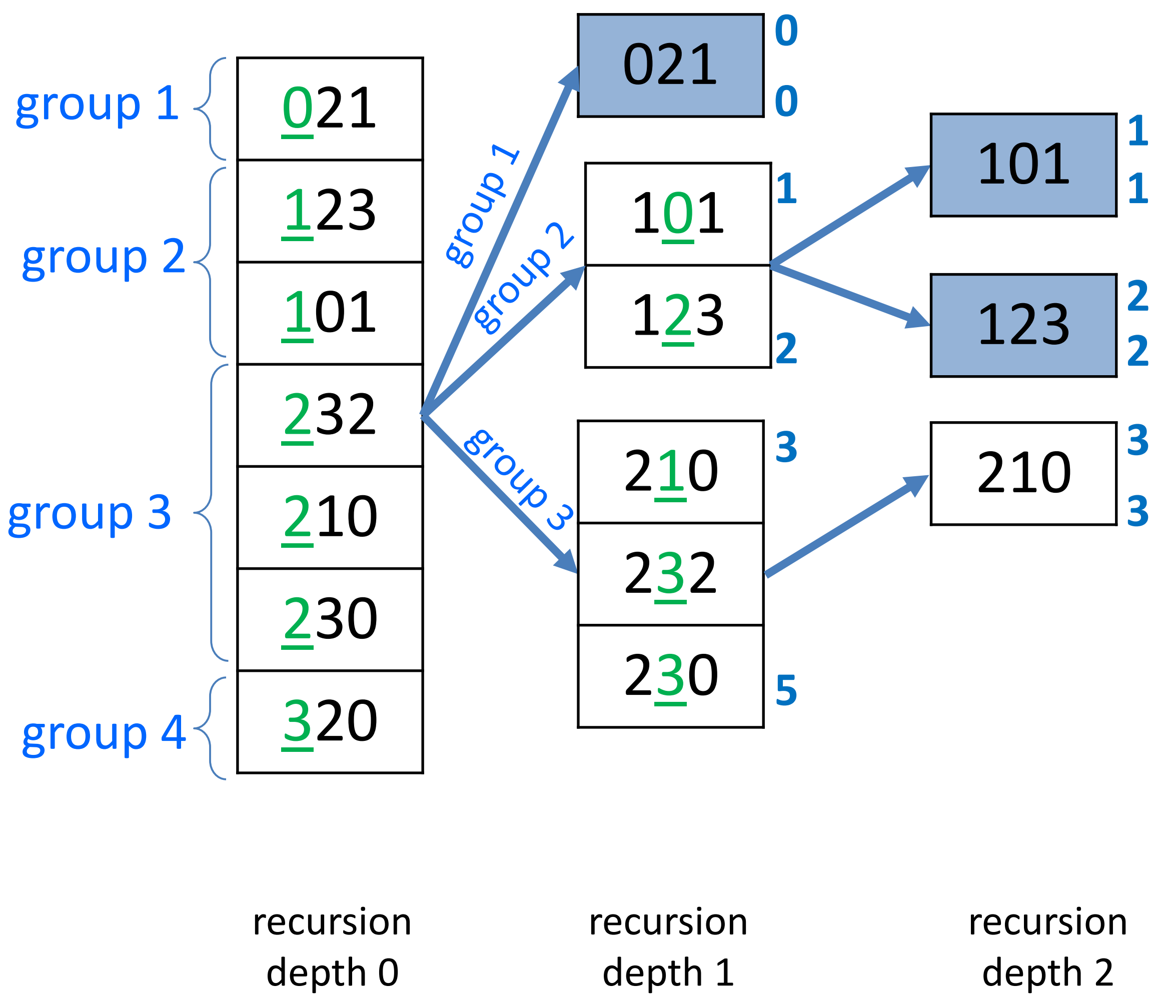
MSD-Radix-Sort

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



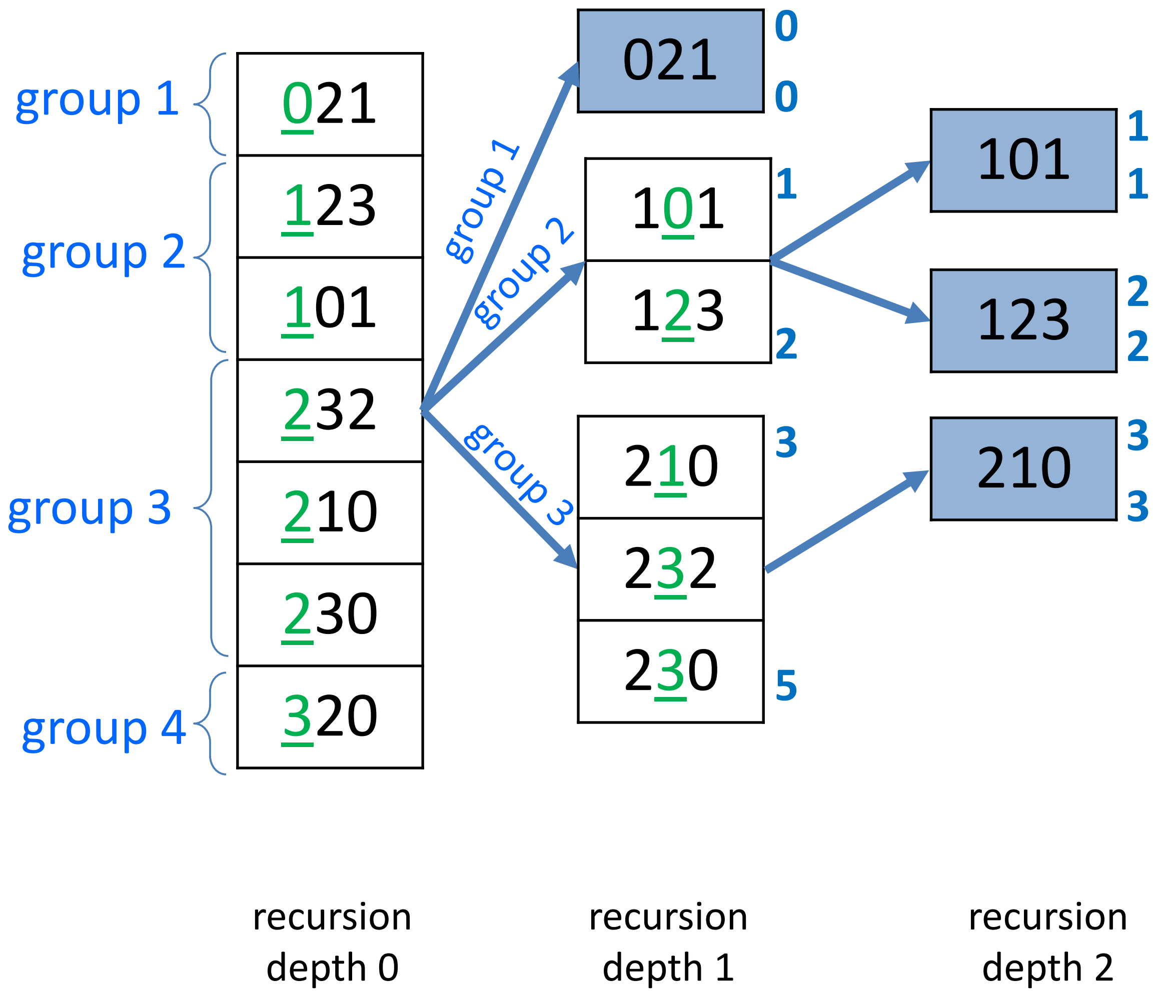
MSD-Radix-Sort

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



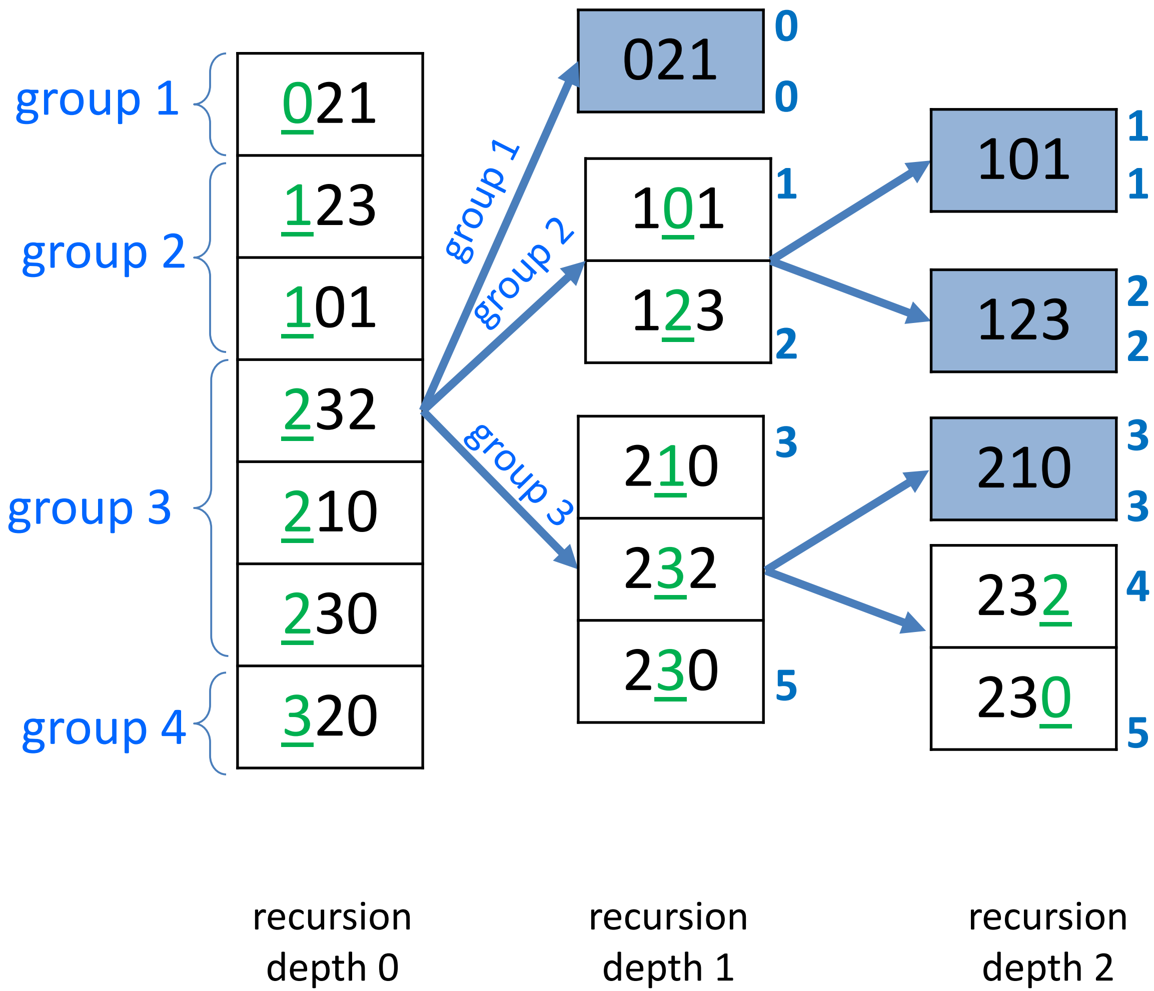
MSD-Radix-Sort

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



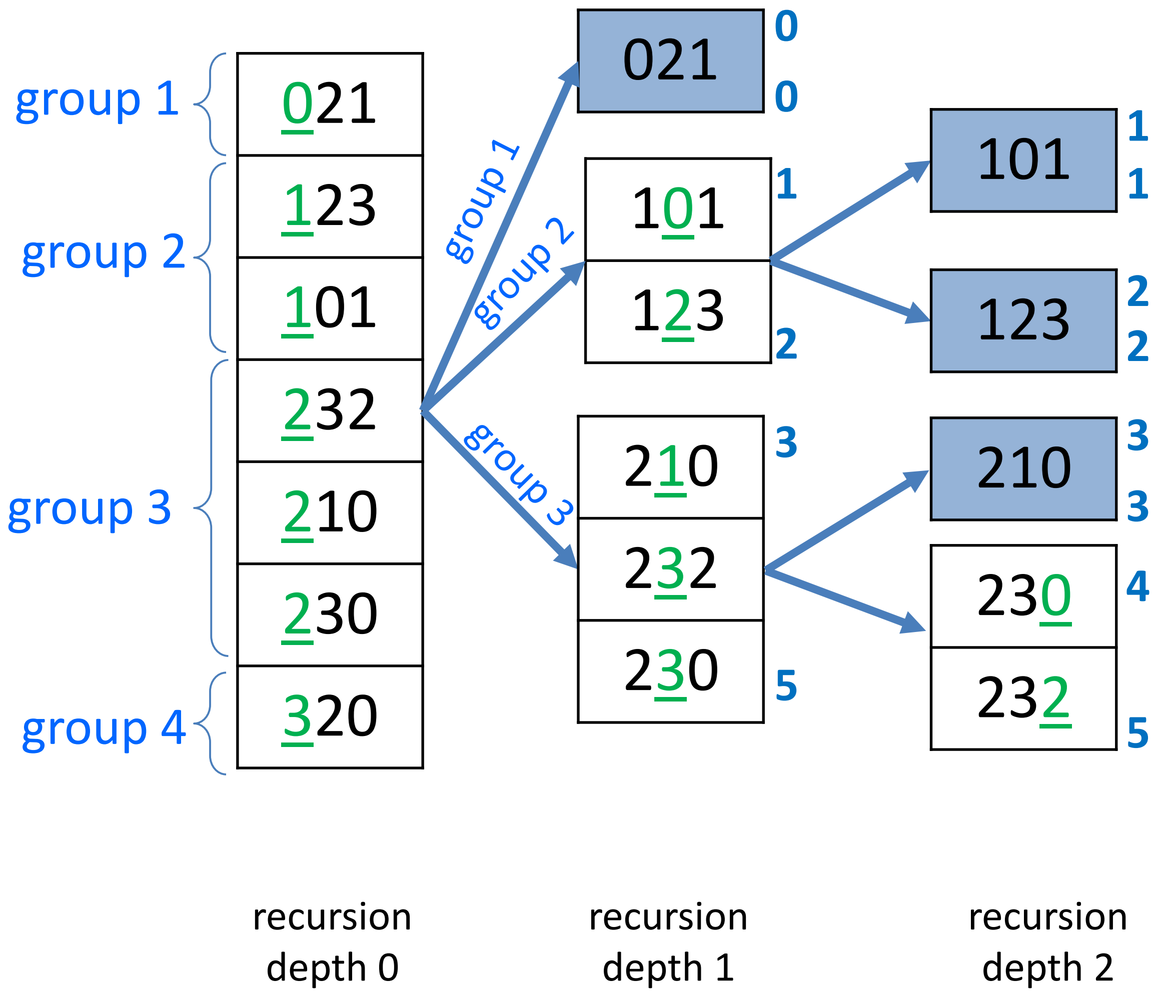
MSD-Radix-Sort

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



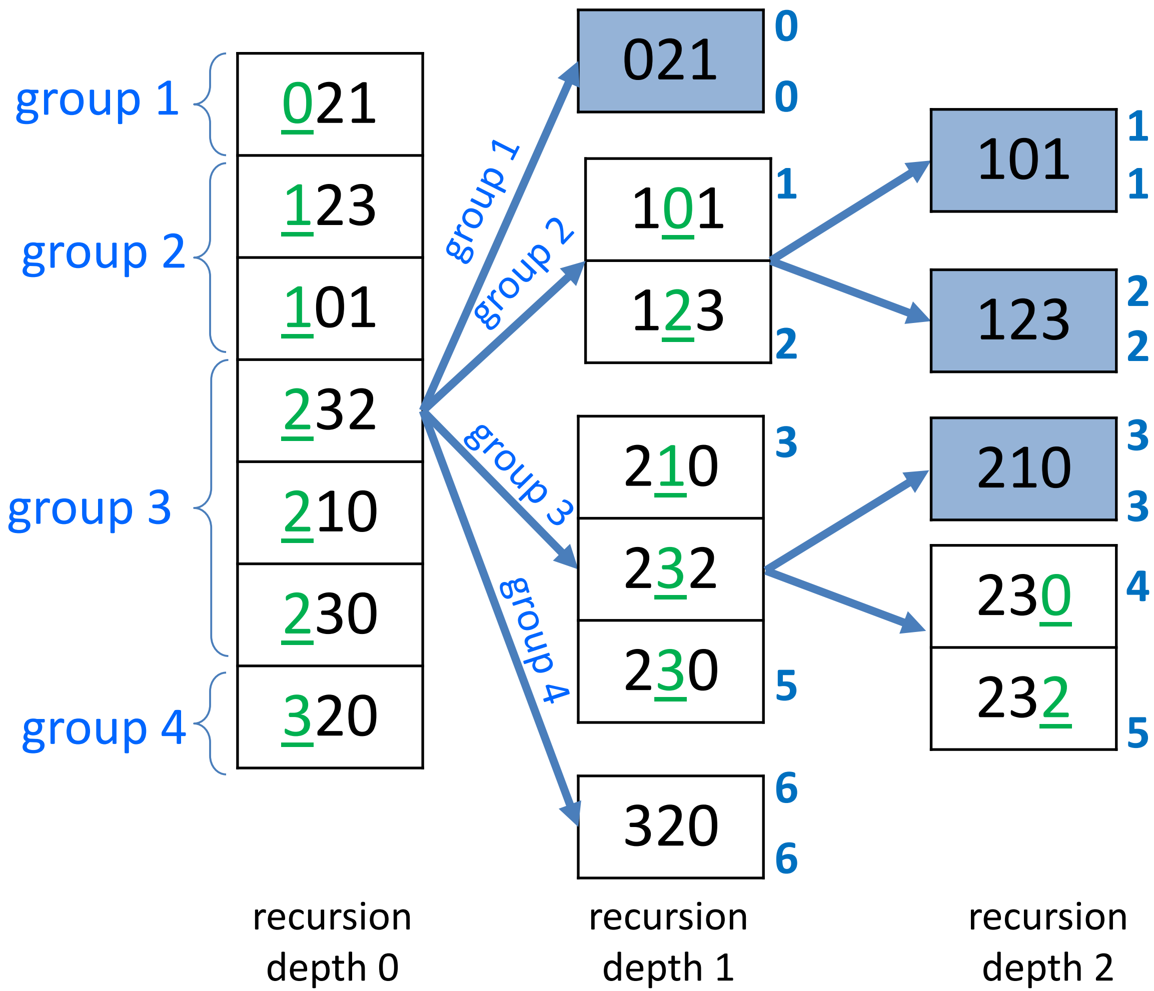
MSD-Radix-Sort

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



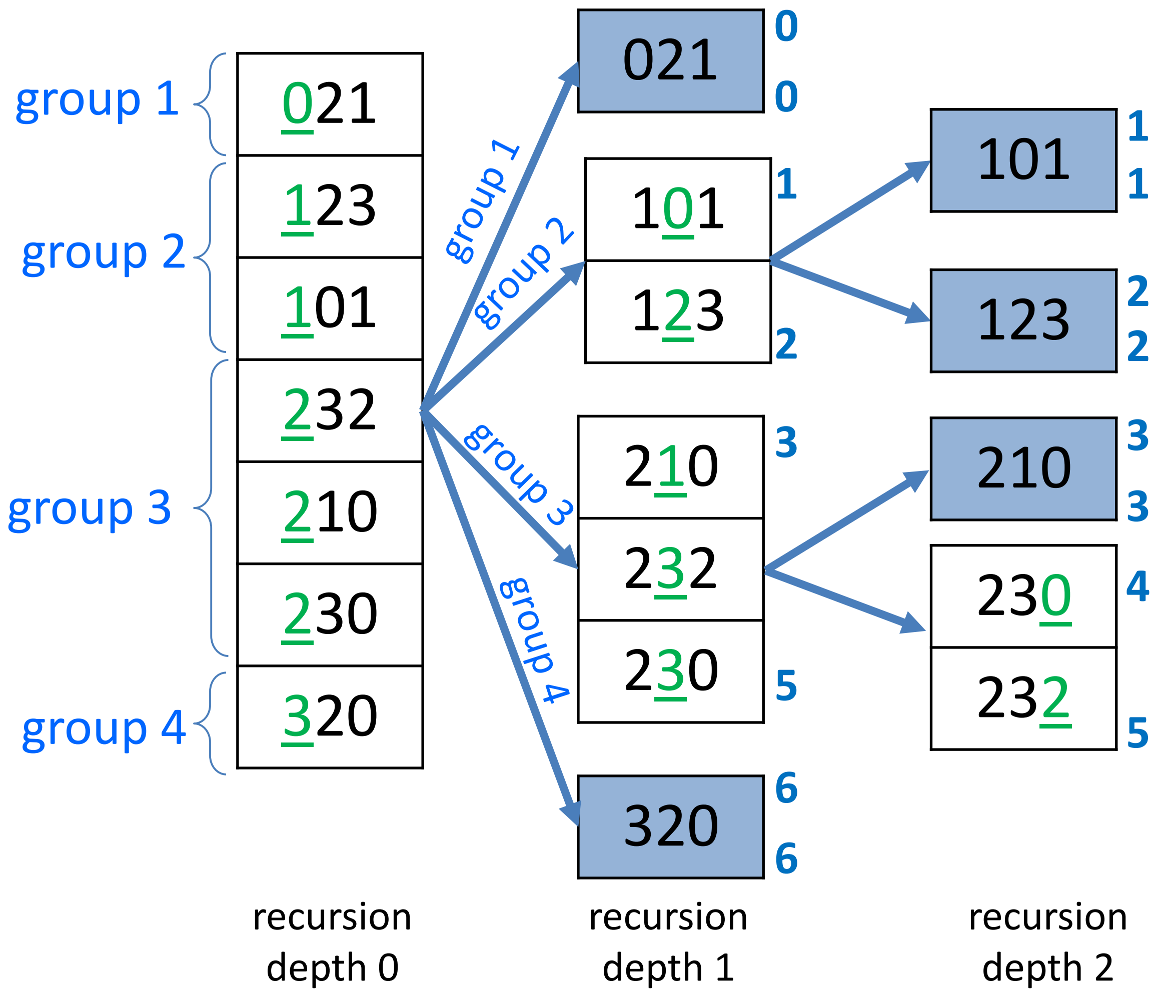
MSD-Radix-Sort

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



MSD-Radix-Sort

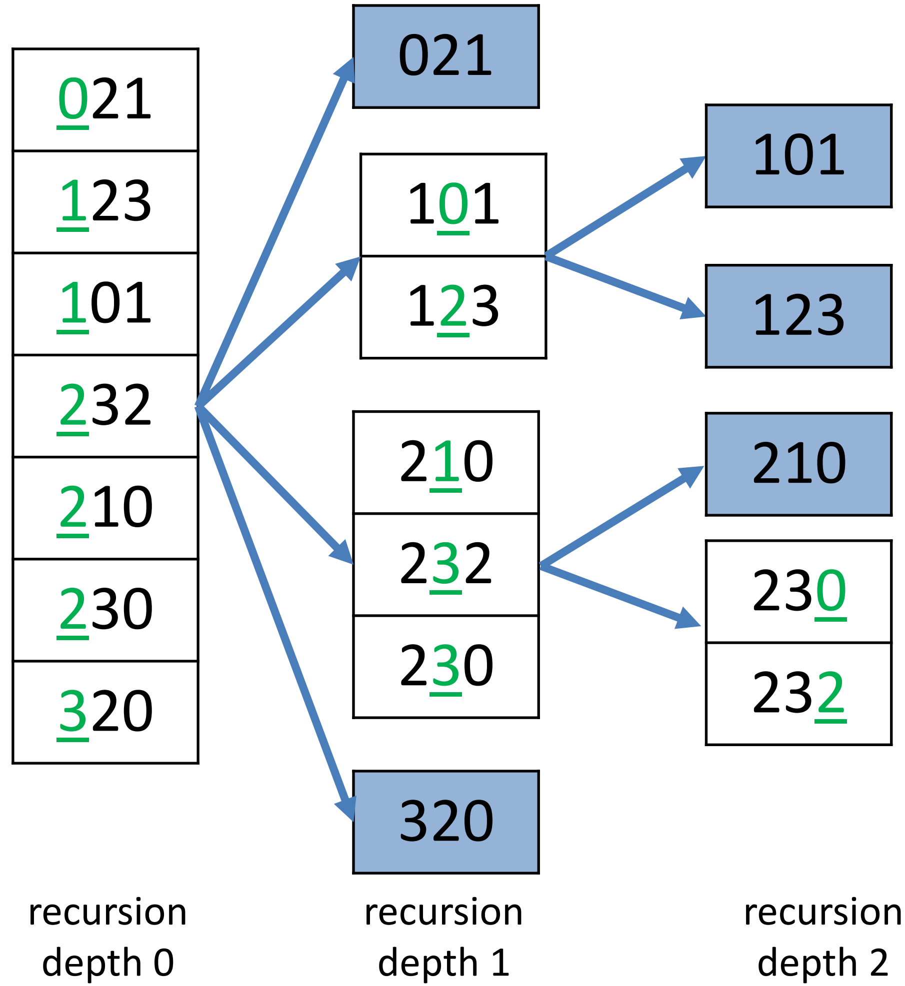
- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group



Note that many digits are never explored

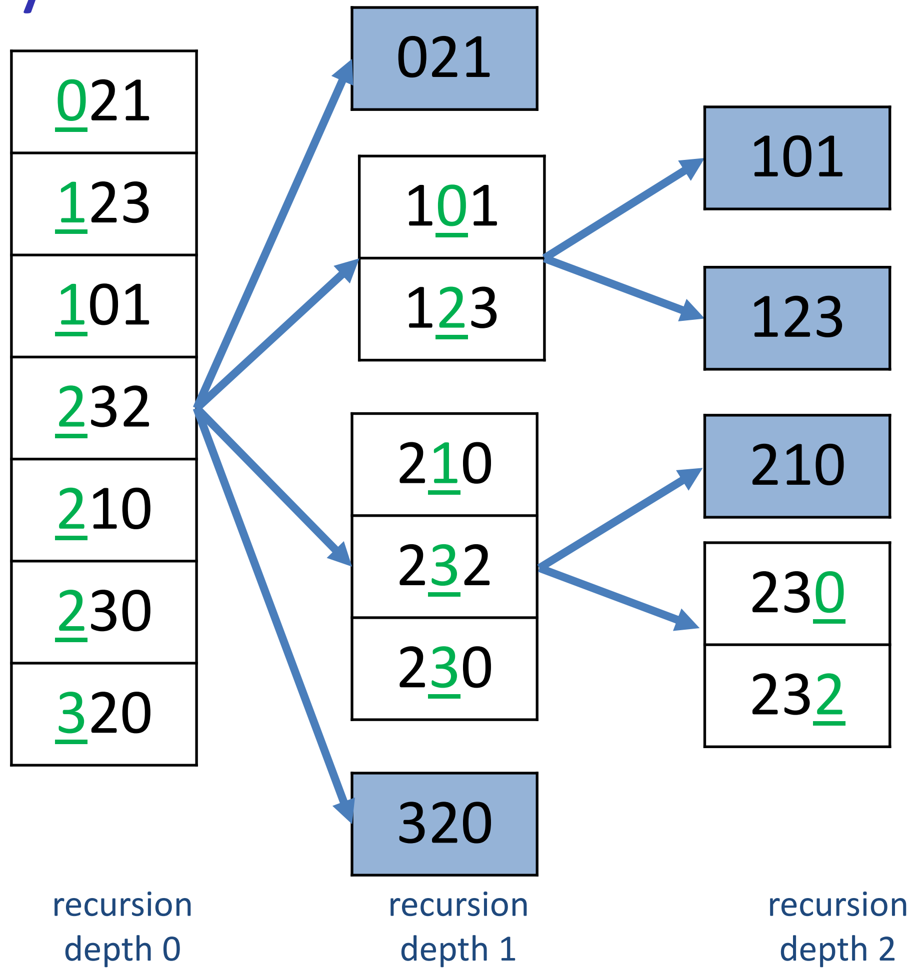
MSD-Radix-Sort Space Analysis

- 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
    - $\sum_{i=0}^{k} n_i \leq n$
    - each bucket sort is $n_i + R$
    - $\sum_{i=0}^{k}(n_i+R) \leq n + \sum_{i=0}^{k} R \leq n + nR$
    - total time at any depth is $O(nR)$
- Number of depths is at most $m-1$
- Total time $O(mnR)$
- Space: $\Theta(n+R)$ for bucket sort, $\Theta(m)$ for recursion stack, total $\Theta(m+n+R)$



MSD-Radix-Sort Pseudocode

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

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

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

if $l < r$

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

if there are digits left

$l' \leftarrow l$

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

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

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 $\mathrm{O}(1)$ 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 $\mathrm{o}(n \log n)$ if the input is special
- Randomized algorithms can eliminate “bad cases”
- Best-case, worst-case, average-case can all differ, but for well designed randomizations of algorithms, the average case runtime of an algorithm is the same as expected runtime of its randomized version