# CS 240 – Data Structures and Data Management

## Module 3: Sorting, Average-case and Randomization

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

Based on lecture notes by many previous cs240 instructors

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

Winter 2022

*version 2022-01-18 14:09*

# Outline

1. 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

# Average-case analysis

We will introduce (and solve) a new problem, and then analyze the *average-case run-time* of our algorithm.

# Average-case analysis

We will introduce (and solve) a new problem, and then analyze the *average-case run-time* of our algorithm.

Recall definition of average-case run-time:

$$T^{avg}(n) = \frac{\sum_{I:\text{size}(I)=n} T(I)}{\#\text{instances of size } n} = \frac{\sum_{I \in \mathcal{I}_n} T(I)}{|\mathcal{I}_n|}$$

(Note: We need that $\mathcal{I}_n$ is finite $\rightarrow$ later)

# Average-case analysis

We will introduce (and solve) a new problem, and then analyze the *average-case run-time* of our algorithm.

Recall definition of average-case run-time:

$$T^{avg}(n) = \frac{\sum_{I:\text{size}(I)=n} T(I)}{\#\text{instances of size } n} = \frac{\sum_{I \in \mathcal{I}_n} T(I)}{|\mathcal{I}_n|}$$

(Note: We need that $\mathcal{I}_n$ is finite $\rightarrow$ later)

To learn how to do this, we will do a simpler example first.

# A contrived example

```
avgCaseDemo(A, n)
A: array of size n with distinct items
1.    if n ≤ 2 return
2.    if A[n−2] < A[n−1]
3.        avgCaseDemo(A[0..n/2−1], n/2)      // Good case
4.    else avgCaseDemo(A[0..n−3], n−2)       // Bad case
```

Let $T(n)$ be the number of *recursions*.
(This is asymptotically the same as the run-time.)

**Worst-case analysis**: Recursive call could always have size $n-2$.
$T(n) = 1 + T(n-2) = 1 + 1 + \cdots + T(2) = n/2 - 1 \in \Theta(n)$

# A contrived example

```
avgCaseDemo(A, n)
A: array of size n with distinct items
1.    if n ≤ 2 return
2.    if A[n−2] < A[n−1]
3.        avgCaseDemo(A[0..n/2−1], n/2)        // Good case
4.    else avgCaseDemo(A[0..n−3], n−2)         // Bad case
```

Let $T(n)$ be the number of *recursions*.
(This is asymptotically the same as the run-time.)

**Worst-case analysis**: Recursive call could always have size $n-2$.
$T(n) = 1 + T(n-2) = 1 + 1 + \cdots + T(2) = n/2 - 1 \in \Theta(n)$

**Best-case analysis**: Recursive call could always have size $n/2$.
$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = \cdots = \log n - 1 \in \Theta(\log n)$

**Average-case analysis?**

# Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size $n$?
  (There are infinitely many sets of $n$ numbers.)

# Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size $n$?
  (There are infinitely many sets of $n$ numbers.)
- **Assume**: All input numbers are *distinct*.
  (For most problems, this can be forced by using tie-breakers.)
- **Observe:** **comparison-based** algorithm has the same run-time on inputs

$$A = [\ 14,\ 3,\ 2,\ 6,\ 1,\ 11,\ 7\ ] \quad \text{and}$$
$$A' = [\ 14,\ 4,\ 2,\ 6,\ 1,\ 12,\ 8\ ]$$

- The actual numbers do not matter, only their *relative order*.

# Sorting Permutations

- Characterize relative order via **sorting permutation**: the permutation $\pi \in \Pi_n$ for which

$$A[\pi(0)] \leq A[\pi(1)] \leq \cdots \leq A[\pi(n-1)].$$

Example:
$$A = [\ 14,\ 3,\ 2,\ 6,\ 1,\ 11,\ 7\ ]$$
$$\pi = [\ 4,\ 2,\ 1,\ 3,\ 6,\ 5,\ 0\ ]$$

Observe: $\pi^{-1} = [\ 6,\ 2,\ 1,\ 3,\ 0,\ 5,\ 4\ ]$
has same sorting permutation as $A$.

# Sorting Permutations

- Characterize relative order via **sorting permutation**: the permutation $\pi \in \Pi_n$ for which

$$A[\pi(0)] \leq A[\pi(1)] \leq \cdots \leq A[\pi(n-1)].$$

Example:
$$A = [\ 14,\ 3,\ 2,\ 6,\ 1,\ 11,\ 7\ ]$$
$$\pi = [\ 4,\ 2,\ 1,\ 3,\ 6,\ 5,\ 0\ ]$$

Observe: $\pi^{-1} = [\ 6,\ 2,\ 1,\ 3,\ 0,\ 5,\ 4\ ]$
has same sorting permutation as $A$.

- Assume all $n!$ sorting permutations are *equally likely*.

$\rightsquigarrow$ Average cost is then $\frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$ where

$$\begin{aligned} T(\pi) &= \text{run-time on any instance with sorting-permutation } \pi \\ &= \text{run-time on } \pi^{-1} \end{aligned}$$

# Average-case run-time of *avgCaseDemo*

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi) \quad = \quad \frac{1}{|\Pi_n|} \Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \quad \Big)$$

# Average-case run-time of *avgCaseDemo*

$$T^{avg}(n) = \frac{1}{n!}\sum_{\pi\in\Pi_n} T(\pi) \quad = \quad \frac{1}{|\Pi_n|}\Big(\sum_{\pi\in\Pi_n:\pi \text{ good}} T(\pi) + \sum_{\pi\in\Pi_n:\pi \text{ bad}} T(\pi) \quad \Big)$$

Recursive formula for one instance $\pi$:

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

(You may be tempted to write $1 + T^{avg}(n/2)$ and $1 + T^{avg}(n-2)$ instead, but this is *not* correct. Why?)

# Average-case run-time of *avgCaseDemo*

$$T^{avg}(n) = \frac{1}{n!}\sum_{\pi\in\Pi_n} T(\pi) \quad = \quad \frac{1}{|\Pi_n|}\Big(\sum_{\pi\in\Pi_n:\pi \text{ good}} T(\pi) + \sum_{\pi\in\Pi_n:\pi \text{ bad}} T(\pi)\Big)$$

Recursive formula for one instance $\pi$:

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

(You may be tempted to write $1 + T^{avg}(n/2)$ and $1 + T^{avg}(n-2)$ instead, but this is *not* correct. Why?)

Recursive formula for all instances $\pi$ together:

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

(This is not at all trivial.)

# Average-case run-time of *avgCaseDemo*

$$T^{avg}(n) = \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \Big)$$

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

# Average-case run-time of *avgCaseDemo*

$$
\begin{aligned}
T^{avg}(n) &= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \Big) \\
&= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} (1 + T^{avg}(n-2)) \Big) \\
&= 1 + \frac{1}{|\Pi_n|}\Big( |\{\pi \in \Pi_n : \pi \text{ good}\}| \cdot T^{avg}(n/2) \\
&\qquad + |\{\pi \in \Pi_n : \pi \text{ bad}\}| \cdot T^{avg}(n-2) \Big)
\end{aligned}
$$

# Average-case run-time of *avgCaseDemo*

$$
\begin{aligned}
T^{avg}(n) &= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \Big) \\
&= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} (1 + T^{avg}(n-2)) \Big) \\
&= 1 + \frac{1}{|\Pi_n|}\Big( |\{\pi \in \Pi_n : \pi \text{ good}\}| \cdot T^{avg}(n/2) \\
&\qquad + |\{\pi \in \Pi_n : \pi \text{ bad}\}| \cdot T^{avg}(n-2) \Big)
\end{aligned}
$$

**Observe:** Exactly half of the permutations are good (why?)

# Average-case run-time of *avgCaseDemo*

$$
\begin{aligned}
T^{avg}(n) &= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \Big) \\
&= \frac{1}{|\Pi_n|}\Big( \sum_{\pi \in \Pi_n : \pi \text{ good}} (1 + T^{avg}(n/2)) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} (1 + T^{avg}(n-2)) \Big) \\
&= 1 + \frac{1}{|\Pi_n|}\Big( |\{\pi \in \Pi_n : \pi \text{ good}\}| \cdot T^{avg}(n/2) \\
&\qquad + |\{\pi \in \Pi_n : \pi \text{ bad}\}| \cdot T^{avg}(n-2) \Big)
\end{aligned}
$$

**Observe:** Exactly half of the permutations are good (why?)

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

# Average-case run-time of *avgCaseDemo*

**Claim:** $T^{avg}(n) \leq 2 \log n$.

**Proof:**

# Average-case run-time of *avgCaseDemo*

**Claim:** $T^{avg}(n) \leq 2 \log n$.
**Proof:**

$\Rightarrow$ *avgCaseDemo* has avg-case run-time $O(\log n)$
(compared to $\Theta(n)$ worst-case time).

# Outline

# Randomized algorithms

- If an algorithm has better average-case time than worst-case time, then randomization is often a good idea.
- A **randomized algorithm** is one which relies on some random numbers in addition to the input.

(Computers cannot generate randomness. We assume that there exists a *pseudo-random number generator (PRNG)*, a deterministic program that uses an initial value or *seed* to generate a sequence of seemingly random numbers. The quality of randomized algorithms depends on the quality of the PRNG!)

- The run-time will depend on the input and the random numbers used.
- **Goal:** Shift the dependency of run-time from what we can't control (the input) to what we *can* control (the random numbers).

*No more bad instances, just unlucky numbers.*

# Expected running time

Define $T(I, R)$ to be the running time of a randomized algorithm $\mathcal{A}$ for an instance $I$ and the sequence of random numbers $R$.

The **expected running time** $T^{exp}(I)$ for instance $I$ is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I, R)] = \sum_{R} T(I, R) \cdot \Pr[R]$$

# Expected running time

Define $T(I, R)$ to be the running time of a randomized algorithm $\mathcal{A}$ for an instance $I$ and the sequence of random numbers $R$.

The **expected running time** $T^{exp}(I)$ for instance $I$ is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I, R)] = \sum_{R} T(I, R) \cdot \Pr[R]$$

Now take the *maximum* over all instances of size $n$ to define the **expected running time** of $\mathcal{A}$.

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

# Expected running time

Define $T(I, R)$ to be the running time of a randomized algorithm $\mathcal{A}$ for an instance $I$ and the sequence of random numbers $R$.

The **expected running time** $T^{exp}(I)$ for instance $I$ is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I, R)] = \sum_{R} T(I, R) \cdot \Pr[R]$$

Now take the *maximum* over all instances of size $n$ to define the **expected running time** of $\mathcal{A}$.

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

We can still have good luck or bad luck, so occasionally we also discuss the very worst that could happen, i.e., $\max_I \max_R T(I, R)$.

# Another contrived example

```
expectedDemo(A, n)
A: array of size n with distinct items
1.    if n ≤ 2 return
2.    if random(2) swap A[n−1] and A[n−2]
3.    if A[n−2] ≤ A[n−1]
4.        expectedDemo(A[0..n/2−1], n/2)      // Good case
5.    else expectedDemo(A[0..n−3], n−2)       // Bad case
```

We assume the existence of a function *random*$(n)$ that returns an integer uniformly from $\{0, 1, 2, \ldots, n-1\}$.

# Another contrived example

```
expectedDemo(A, n)
A: array of size n with distinct items
1.    if n ≤ 2 return
2.    if random(2) swap A[n−1] and A[n−2]
3.    if A[n−2] ≤ A[n−1]
4.        expectedDemo(A[0..n/2−1], n/2)      // Good case
5.    else expectedDemo(A[0..n−3], n−2)       // Bad case
```

We assume the existence of a function *random*$(n)$ that returns an integer uniformly from $\{0, 1, 2, \ldots, n-1\}$.

Observe: Pr(good case) $= \frac{1}{2} =$ Pr(bad case).

# Expected run-time of *expectedDemo*

Run-time on array $A$ if random outcomes are $R = \langle x, R' \rangle$:

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

# Expected run-time of *expectedDemo*

Run-time on array $A$ if random outcomes are $R = \langle x, R' \rangle$:

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

Summing up over all sequences of random outcomes:

$$\sum_R \Pr(R)\, T(A,R) = \Pr(X \text{ good}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots \tfrac{n}{2}-1]\,, R')\Big) + \Pr(X \text{ bad}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots n-3]\,, R')\Big)$$

# Expected run-time of *expectedDemo*

Run-time on array $A$ if random outcomes are $R = \langle x, R' \rangle$:

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

Summing up over all sequences of random outcomes:

$$\begin{aligned} \sum_R \Pr(R)\, T(A,R) &= \Pr(X \text{ good}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots \tfrac{n}{2}-1]\,, R')\Big) \\ &\quad + \Pr(X \text{ bad}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots n-3]\,, R')\Big) \\ &= 1 + \tfrac{1}{2} \sum_{R'} \Pr(R') \cdot T(\, A[0 \ldots \tfrac{n}{2}-1]\,, R') + \tfrac{1}{2} \sum_{R'} \Pr(R') \cdot T(\, A[0 \ldots n-3]\,, R') \end{aligned}$$

# Expected run-time of *expectedDemo*

Run-time on array $A$ if random outcomes are $R = \langle x, R' \rangle$:

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

Summing up over all sequences of random outcomes:

$$\begin{aligned} \sum_R \Pr(R)\, T(A,R) &= \Pr(X \text{ good}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots \tfrac{n}{2}-1]\,, R')\Big) \\ &\quad + \Pr(X \text{ bad}) \sum_{R'} \Pr(R') \Big(1 + T(\, A[0 \ldots n-3]\,, R')\Big) \\ &= 1 + \tfrac{1}{2} \sum_{R'} \Pr(R') \cdot T(\, A[0 \ldots \tfrac{n}{2}-1]\,, R') + \tfrac{1}{2} \sum_{R'} \Pr(R') \cdot T(\, A[0 \ldots n-3]\,, R') \\ &\leq 1 + \tfrac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n/2}} \sum_{R'} \Pr(R') \cdot T(A', R')}_{T^{\exp}(\lfloor n/2 \rfloor)} + \tfrac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n-2}} \sum_{R'} \Pr(R') \cdot T(A', R')}_{T^{\exp}(n-2)} \end{aligned}$$

# Expected run-time of *expectedDemo*

- $\sum_R \Pr(R)\,T(A,R) \leq 1 + \frac{1}{2}\,T^{exp}(n/2) + \frac{1}{2}\,T^{exp}(n-2)$ holds for *all* $A$.

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

# Expected run-time of *expectedDemo*

- $\sum_R \Pr(R)\,T(A,R) \leq 1 + \frac{1}{2}\,T^{exp}(n/2) + \frac{1}{2}\,T^{exp}(n-2)$ holds for *all* $A$.

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

- Same recursion as for $T^{avg}_{avgCaseDemo}(n)$
- Same analysis $\rightsquigarrow T^{exp}_{expectedDemo}(n) \in O(\log n)$

# Expected run-time of *expectedDemo*

- $\sum_R \Pr(R)\, T(A,R) \leq 1 + \frac{1}{2} T^{exp}(n/2) + \frac{1}{2} T^{exp}(n-2)$ holds for *all* $A$.

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

- Same recursion as for $T^{avg}_{avgCaseDemo}(n)$
- Same analysis $\rightsquigarrow$ $T^{exp}_{expectedDemo}(n) \in O(\log n)$

- Is this a coincidence? Or does the expected time of a randomized version always have something to do with the average-case time?
- Not in general! (But we will see examples where it does.)

# Outline

# The Selection Problem

The **selection problem**: Given an array $A$ of $n$ numbers, and $0 \leq k < n$, find the element that would be at position $k$ of the sorted array.

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

*select*(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

**Question**: Can we do selection in linear time?

# The Selection Problem

The **selection problem**: Given an array $A$ of $n$ numbers, and $0 \leq k < n$, find the element that would be at position $k$ of the sorted array.

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

*select*(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

**Question**: Can we do selection in linear time?
The *QuickSelect* algorithm answers this question in the affirmative.

# The Selection Problem

The **selection problem**: Given an array $A$ of $n$ numbers, and $0 \leq k < n$, find the element that would be at position $k$ of the sorted array.

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

*select*(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

**Question**: Can we do selection in linear time?
The *QuickSelect* algorithm answers this question in the affirmative.

The encountered sub-routines will also be useful otherwise.

# Crucial Subroutines

*QuickSelect* and the related algorithm *QuickSort* rely on two subroutines:

- *choose-pivot*$(A)$: Return an index $p$ in $A$. We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.

# Crucial Subroutines

*QuickSelect* and the related algorithm *QuickSort* rely on two subroutines:

- *choose-pivot*($A$): Return an index $p$ in $A$. We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.

  Simplest idea: Always select rightmost element in array

```
choose-pivot(A)
1.    return A.size−1
```

  We will consider more sophisticated ideas later on.

# Crucial Subroutines

*QuickSelect* and the related algorithm *QuickSort* rely on two subroutines:

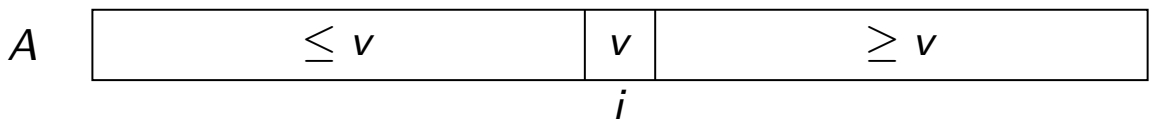
- *choose-pivot*$(A)$: Return an index $p$ in $A$. We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.

  Simplest idea: Always select rightmost element in array

```
choose-pivot(A)
1.    return A.size−1
```

  We will consider more sophisticated ideas later on.

- *partition*$(A, p)$: Rearrange $A$ and return **pivot-index** $i$ so that
  - the pivot-value $v$ is in $A[i]$,
  - all items in $A[0, \ldots, i-1]$ are $\leq v$, and
  - all items in $A[i+1, \ldots, n-1]$ are $\geq v$.



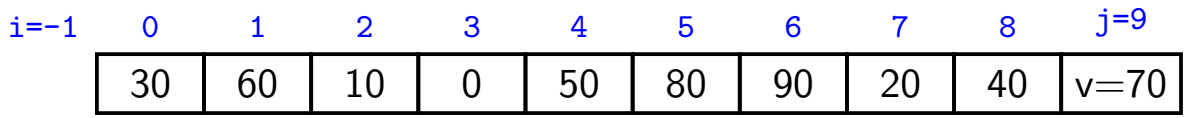
# Partition Algorithm

Conceptually easy linear-time implementation:

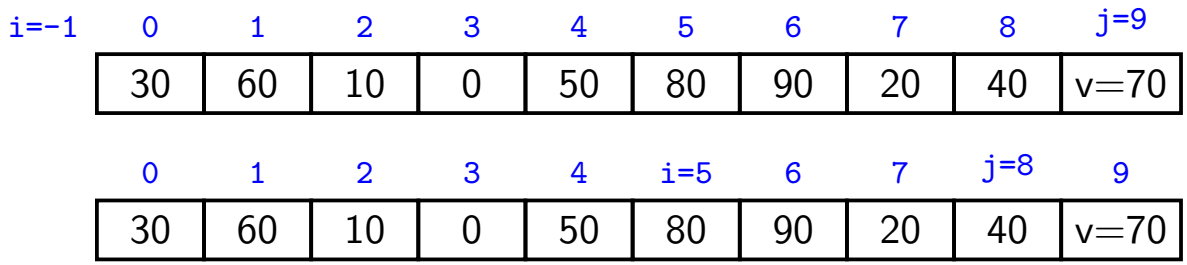
```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
 1.    Create empty lists smaller, equal and larger.
 2.    v ← A[p]
 3.    for each element x in A
 4.        if x < v then smaller.append(x)
 5.        else if x > v then larger.append(x)
 6.        else equal.append(x).
 7.    i ← smaller.size
 8.    j ← equal.size
 9.    Overwrite A[0 . . . i−1] by elements in smaller
10.    Overwrite A[i . . . i+j−1] by elements in equal
11.    Overwrite A[i+j . . . n−1] by elements in larger
12.    return i
```

More challenging: partition **in place** (with $O(1)$ auxiliary space).

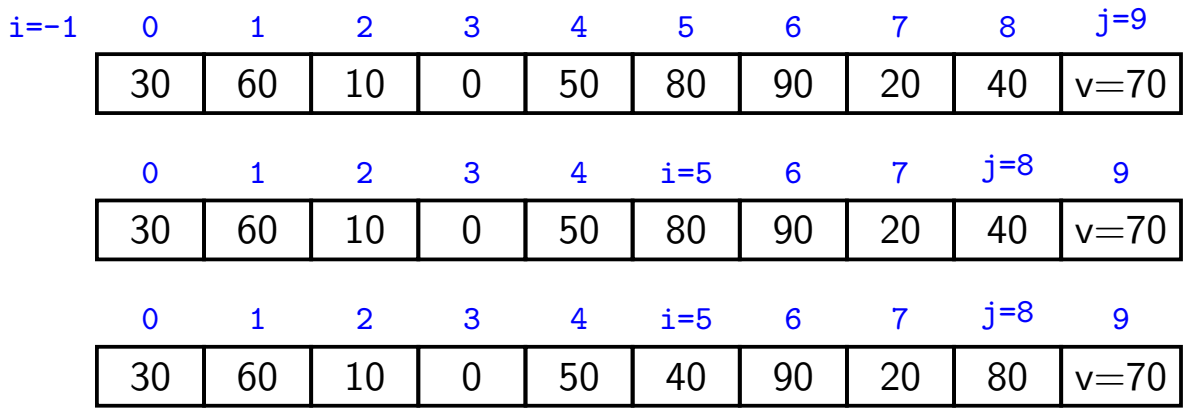
# Efficient In-Place partition (Hoare)



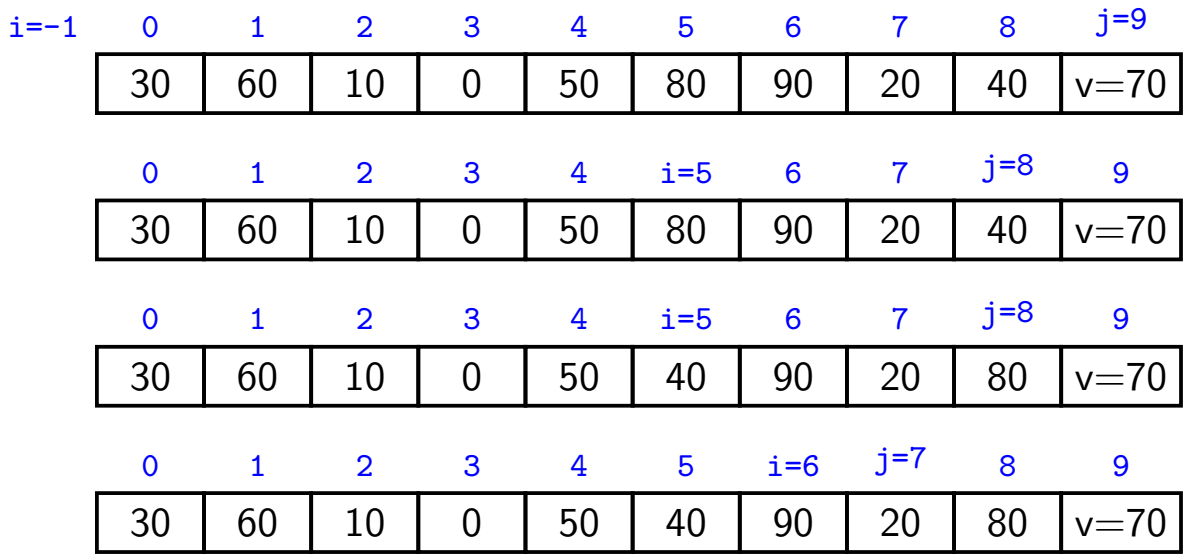
# Efficient In-Place partition (Hoare)



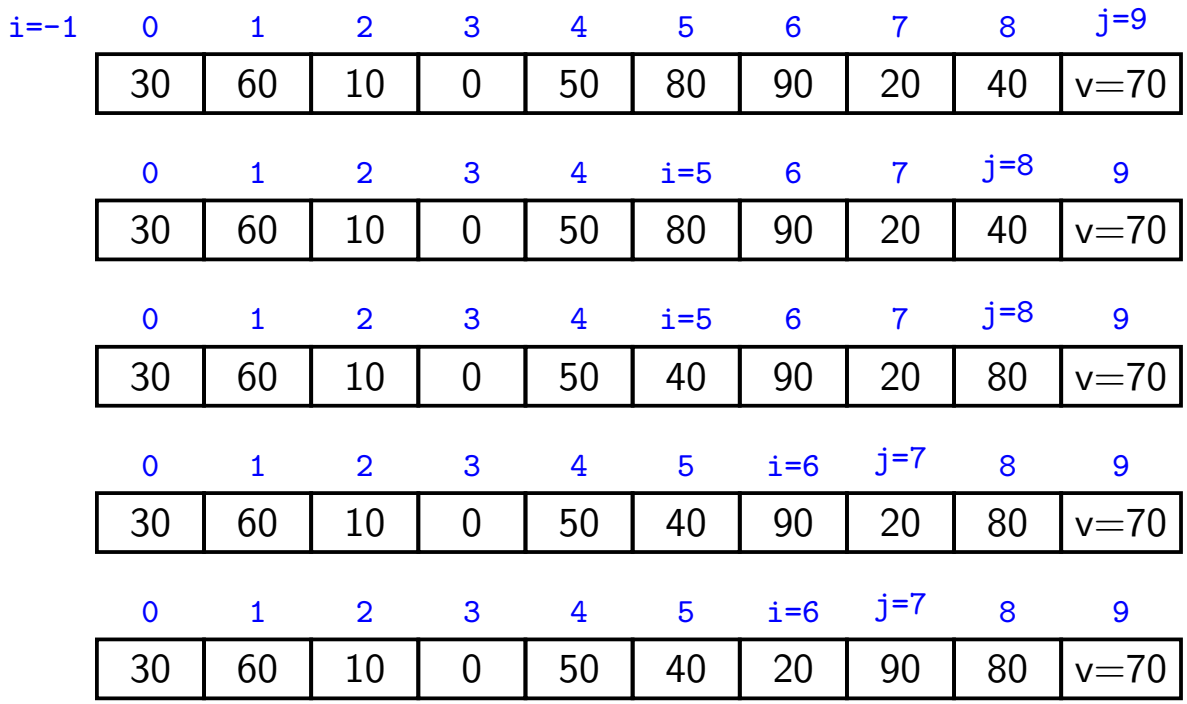
# Efficient In-Place partition (Hoare)



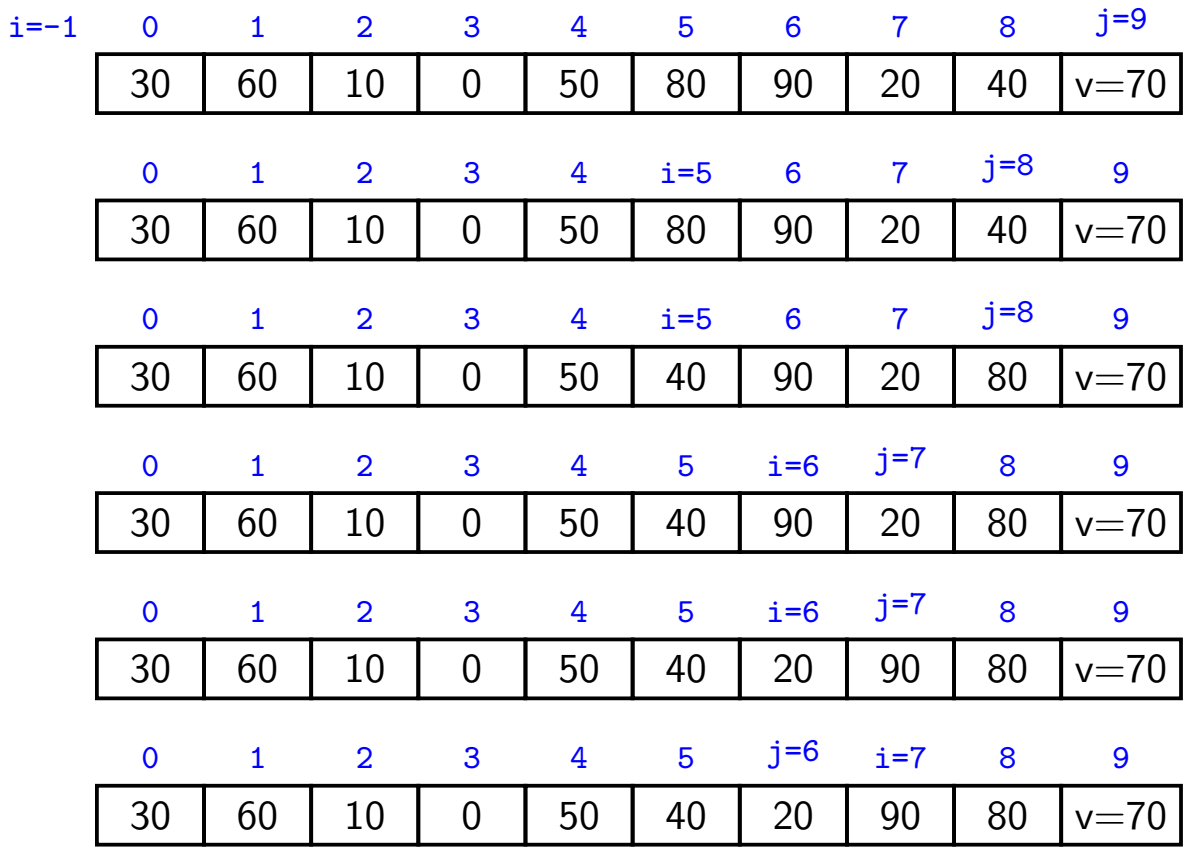
# Efficient In-Place partition (Hoare)



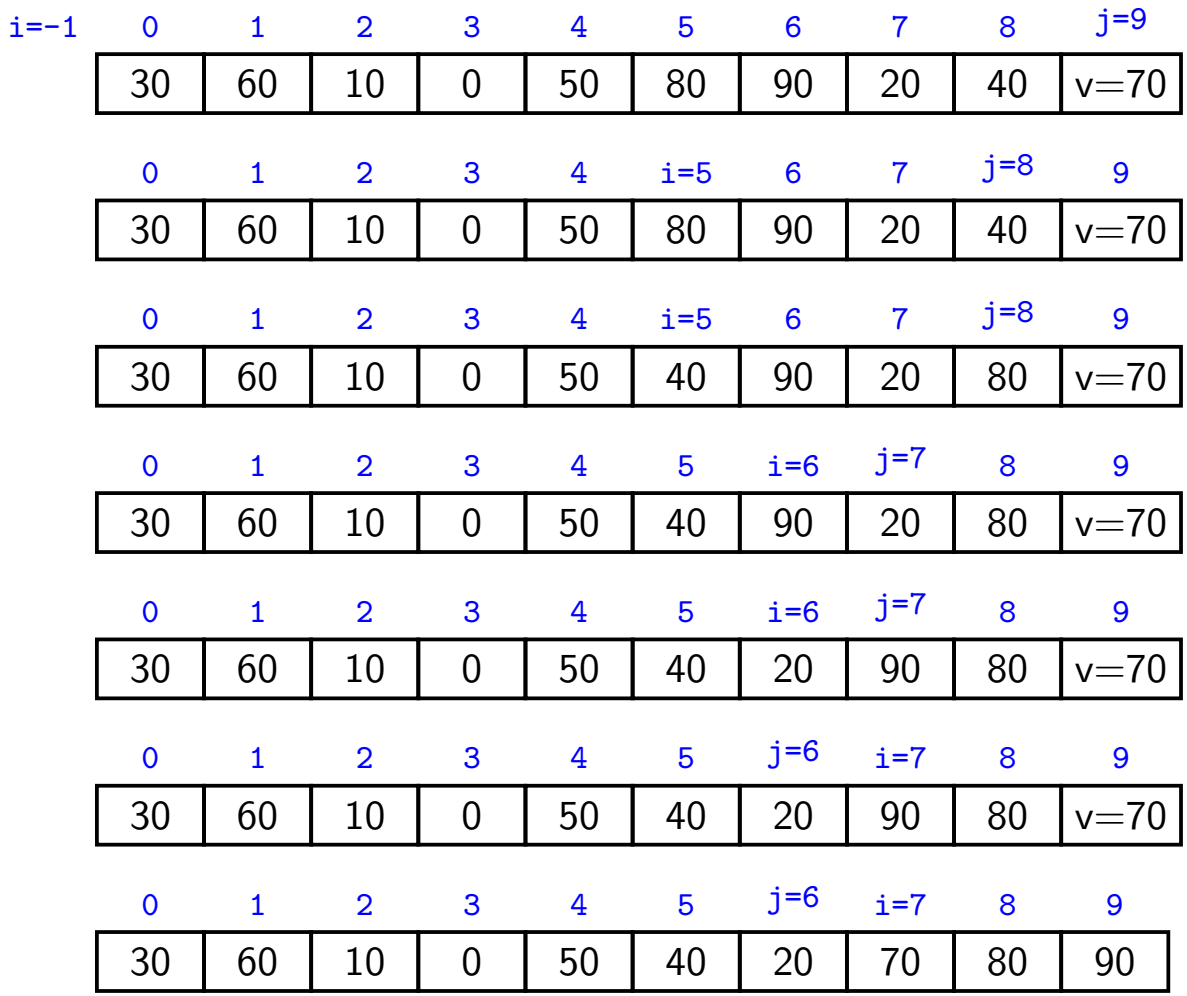
# Efficient In-Place partition (Hoare)



# Efficient In-Place partition (Hoare)



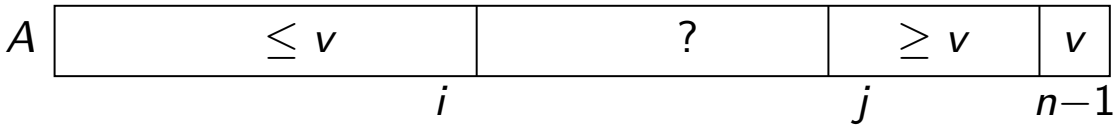
# Efficient In-Place partition (Hoare)



# Efficient In-Place partition (Hoare)

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

Loop invariant:



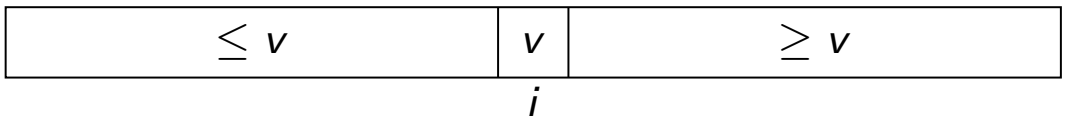

```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
1.    swap(A[n−1], A[p])
2.    i ← −1,   j ← n−1,   v ← A[n−1]
3.    loop
4.        do i ← i+1 while A[i] < v
5.        do j ← j−1 while j ≥ i and A[j] > v
6.        if i ≥ j then break    (goto 9)
7.        else swap(A[i], A[j])
8.    end loop
9.    swap(A[n−1], A[i])
10.   return i
```

Running time: $\Theta(n)$.

# QuickSelect Algorithm

```
QuickSelect(A, k)
A: array of size n,   k: integer s.t. 0 ≤ k < n
 1.     p ← choose-pivot(A)
 2.     i ← partition(A, p)
 3.     if i = k then
 4.         return A[i]
 5.     else if i > k then
 6.         return QuickSelect(A[0, 1, ..., i−1], k)
 7.     else if i < k then
 8.         return QuickSelect(A[i+1, i+2, ..., n−1], k − (i+1))
```

Idea: After partition have




Where is the desired value if $k < i$? If $k = i$? If $k > i$?

# Analysis of *QuickSelect*

Let $T(n, k)$ be the number of **key-comparisons** in a size-$n$ array with parameter $k$. (This is asymptotically the same as the run-time.)

*partition* uses $n$ key-comparisons.

**Worst-case analysis**: Pivot-index is last, $k = 0$
$T(n, 0) \geq n + (n-1) + (n-2) + \cdots + 1 \in \Omega(n^2)$ (and this is tight)

# Analysis of *QuickSelect*

Let $T(n, k)$ be the number of **key-comparisons** in a size-$n$ array with parameter $k$. (This is asymptotically the same as the run-time.)

*partition* uses $n$ key-comparisons.

**Worst-case analysis**: Pivot-index is last, $k = 0$
$T(n, 0) \geq n + (n-1) + (n-2) + \cdots + 1 \in \Omega(n^2)$ (and this is tight)

**Best-case analysis**: First chosen pivot could be the $k$th element
No recursive calls; $T(n, k) = n \in \Theta(n)$

# Analysis of *QuickSelect*

Let $T(n, k)$ be the number of **key-comparisons** in a size-$n$ array with parameter $k$. (This is asymptotically the same as the run-time.)

*partition* uses $n$ key-comparisons.

**Worst-case analysis**: Pivot-index is last, $k = 0$
$T(n, 0) \geq n + (n-1) + (n-2) + \cdots + 1 \in \Omega(n^2)$ (and this is tight)

**Best-case analysis**: First chosen pivot could be the $k$th element
No recursive calls; $T(n, k) = n \in \Theta(n)$

**Average case analysis?**

# Average-Case Analysis of *QuickSelect*

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

(We ignore parameter $k$ here; it turns out not to matter)

# Average-Case Analysis of *QuickSelect*

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

(We ignore parameter $k$ here; it turns out not to matter)

Assume that sorting permutation $\pi$ gives pivot-index is $i$:

- If new array (after *partition*) is $A'$, then

$$T(\pi) \leq n + \max \left\{ T(\underbrace{A'[0..i-1]}_{\text{size } i}), T(\underbrace{A'[i+1..n-1]}_{\text{size } n-i-1}]) \right\}$$

# Average-Case Analysis of *QuickSelect*

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

(We ignore parameter $k$ here; it turns out not to matter)

Assume that sorting permutation $\pi$ gives pivot-index is $i$:

- If new array (after *partition*) is $A'$, then

$$T(\pi) \leq n + \max\left\{ T(\underbrace{A'[0..i-1]}_{\text{size } i}), T(\underbrace{A'[i+1..n-1]}_{\text{size } n-i-1}]) \right\}$$

**Option 1:** Prove that this implies the following:

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

(**Very** complicated proof.)

(And then analyze the recursion, which is not too difficult.)

# Average-Case Analysis of *QuickSelect*

**Option 2:** Prove avg-case run-time *via randomization*

Simpler to do, and randomization is useful in practice.

# Average-Case Analysis of *QuickSelect*

**Option 2:** Prove avg-case run-time *via randomization*

Simpler to do, and randomization is useful in practice.

Need to discuss:

1. How to randomize *QuickSelect*? ($\rightsquigarrow$ *RandomizedQuickSelect*)

# Average-Case Analysis of *QuickSelect*

**Option 2:** Prove avg-case run-time *via randomization*

Simpler to do, and randomization is useful in practice.

Need to discuss:

1. How to randomize *QuickSelect*? ($\rightsquigarrow$ *RandomizedQuickSelect*)
2. What is the expected run-time of *RandomizedQuickSelect*?

# Average-Case Analysis of *QuickSelect*

**Option 2:** Prove avg-case run-time *via randomization*

Simpler to do, and randomization is useful in practice.

Need to discuss:

1. How to randomize *QuickSelect*? ($\rightsquigarrow$ *RandomizedQuickSelect*)
2. What is the expected run-time of *RandomizedQuickSelect*?
3. What does this imply for avg-case run-time of *QuickSelect*?

# Randomizing QuickSelect: Shuffle

**Goal**: Create a randomized version of *QuickSelect*.

**First idea**: Randomly permute the input first using *shuffle*:

```
shuffle(A)
A: array of size n
1.      for i ← 1 to n−1 do
2.              swap( A[i], A[random(i+1)] )
```

This works well, but we can do it directly within the routine.

# Randomizing QuickSelect: Random Pivot

**Second idea**: Change the pivot selection.

*RandomizedQuickSelect*$(A, k)$

1. ...
2. $p \leftarrow$ *random*$(A.$*size*$)$
3. $i \leftarrow$ *partition*$(A, p)$
4. ...

Observe: Pr(pivot has index $i$) $= \frac{1}{n}$

# Randomizing QuickSelect: Random Pivot

**Second idea**: Change the pivot selection.

*RandomizedQuickSelect*$(A, k)$

1. ...
2. $p \leftarrow$ *random*$(A.size)$
3. $i \leftarrow$ *partition*$(A, p)$
4. ...

Observe: Pr(pivot has index $i$) $= \frac{1}{n}$

Assume we know that first *random* gave pivot-index $i$:

- We recurse in an array of size $i$ or $n-i-1$ (or not at all)
- If new array (after *partition*) is $A'$, and $R = \langle i, R' \rangle$ then

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

# Analysis of *RandomizedQuickSelect*

So $T(\pi, k, \langle i, R' \rangle) \leq n + T(\text{some array of size } i \text{ or } n-i-1, \text{some } k', R')$

# Analysis of *RandomizedQuickSelect*

So $T(\pi, k, \langle i, R' \rangle) \leq n + T(\text{some array of size } i \text{ or } n-i-1, \text{some } k', R')$

**Claim:** Over all choices of $i$ and $R'$, this hits the expected values.

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

(Proof similar to *expectedDemo*. Crucial: $T^{exp}(\cdot)$ uses the *maximum* over all instances.)

# Analysis of *RandomizedQuickSelect*

So $T(\pi, k, \langle i, R' \rangle) \leq n + T(\text{some array of size } i \text{ or } n-i-1, \text{some } k', R')$

**Claim:** Over all choices of $i$ and $R'$, this hits the expected values.

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

(Proof similar to *expectedDemo*. Crucial: $T^{exp}(\cdot)$ uses the *maximum* over all instances.)

Note: we get the same bound for all $\pi, k$.

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

# Analysis of *RandomizedQuickSelect*

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

**Claim:** This recursion resolves to $O(n)$.

**Proof:**

# Analysis of *RandomizedQuickSelect*

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

**Claim:** This recursion resolves to $O(n)$.
**Proof:**

$\Rightarrow$ *RandomizedQuickSelect* has expected run-time $O(n)$.

*This is generally the fastest QuickSelect implementation.*

There exists a variation that has worst-case running time $O(n)$, but it uses double recursion and is slower in practice. ($\rightsquigarrow$ *cs*341)

# Expected running time vs. average-case running time

- Assume we have an algorithm $\mathcal{A}$ that solves Selection or Sorting.
- Create a randomized algorithm $\mathcal{B}$ as follows:
  1. Let $I$ be the given instance (an array)
  2. Randomly (and uniformly) permute $I$ to get $I'$
     (We can do this with *shuffle*. For *QuickSelect*, choosing the pivot randomly has the same effect.)
  3. Call algorithm $\mathcal{A}$ on input $I'$

# Expected running time vs. average-case running time

- Assume we have an algorithm $\mathcal{A}$ that solves Selection or Sorting.
- Create a randomized algorithm $\mathcal{B}$ as follows:
  1. Let $I$ be the given instance (an array)
  2. Randomly (and uniformly) permute $I$ to get $I'$
     (We can do this with *shuffle*. For *QuickSelect*, choosing the pivot randomly has the same effect.)
  3. Call algorithm $\mathcal{A}$ on input $I'$

**Claim:** $T_{\mathcal{B}}^{exp}(n) = T_{\mathcal{A}}^{avg}(n)$
**Proof:**

Since *RandomizedQuickSelect* has expected run-time $O(n)$, therefore *QuickSelect* has average-case run-time $O(n)$.

# Outline

# QuickSort

Hoare developed *partition* and *QuickSelect* in 1960.
He also used them to *sort* based on partitioning:

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

# QuickSort analysis

Now set $T(n) :=$ # of key-comparison for *QuickSort* in a size-$n$ array.

**Worst-case analysis**: Recursive call could always have size $n-1$.
$T(n) \geq n + T(n-1) \in \Omega(n^2)$ exactly as for *QuickSelect*
(This is tight since the recursion depth is at most $n$.)

# QuickSort analysis

Now set $T(n) :=$ # of key-comparison for *QuickSort* in a size-$n$ array.

**Worst-case analysis**: Recursive call could always have size $n-1$.
$T(n) \geq n + T(n-1) \in \Omega(n^2)$ exactly as for *QuickSelect*
(This is tight since the recursion depth is at most $n$.)

**Best-case analysis**: If pivot-index is always in the middle, then we recurse in two sub-arrays of size $\leq n/2$.
$T(n) \leq n + 2T(n/2) \in O(n \log n)$ exactly as for *MergeSort*
(This can be shown to be tight.)

# QuickSort analysis

Now set $T(n) :=$ # of key-comparison for *QuickSort* in a size-$n$ array.

**Worst-case analysis**: Recursive call could always have size $n-1$.
$T(n) \geq n + T(n-1) \in \Omega(n^2)$ exactly as for *QuickSelect*
(This is tight since the recursion depth is at most $n$.)

**Best-case analysis**: If pivot-index is always in the middle, then we recurse in two sub-arrays of size $\leq n/2$.
$T(n) \leq n + 2T(n/2) \in O(n \log n)$ exactly as for *MergeSort*
(This can be shown to be tight.)

**Average-case analysis?** We again prove this via randomization.

# Randomizing QuickSort

```
RandomizedQuickSort(A)
1.    ...
2.    p ← random(A.size)
3.    i ← partition(A, p)
4.    ...
```

Observe: $\Pr(\text{pivot has index } i) = \dfrac{1}{n}$

# Randomizing QuickSort

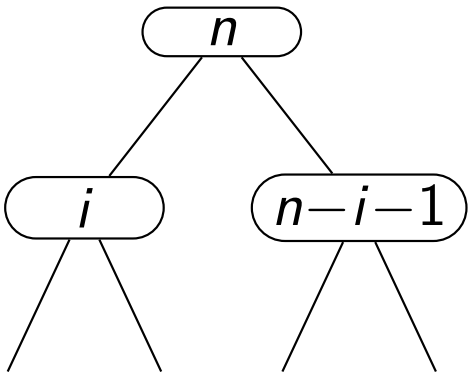
```
RandomizedQuickSort(A)
1.    ...
2.    p ← random(A.size)
3.    i ← partition(A, p)
4.    ...
```

Observe: $\Pr(\text{pivot has index } i) = \dfrac{1}{n}$

Assume we know that pivot-index is $i$:

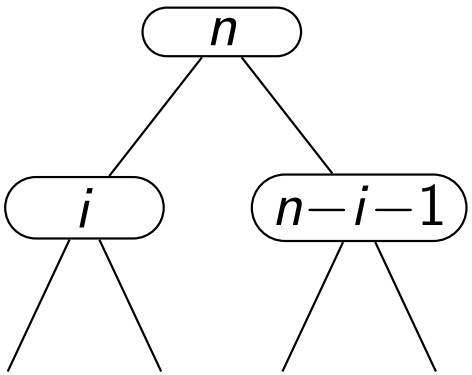
- We recurse in two arrays, of size $i$ and $n-i-1$
- Can use this to show $T^{exp}(n) \leq n + \frac{2}{n}\sum_{i=0}^{n-1} T^{exp}(i)$ (and then show that this is in $O(n \log n)$) but there is an even easier analysis!

# Expected recursion-depth for QuickSort




**Goal:** Analyze expected height of **recursion tree**.

# Expected recursion-depth for QuickSort

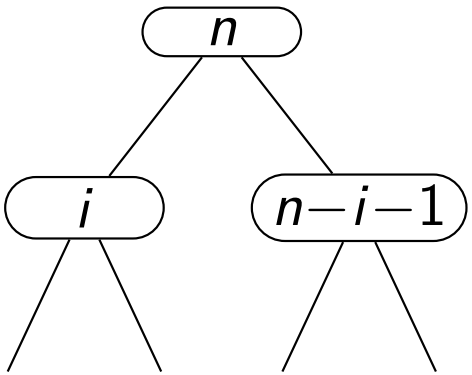



**Goal:** Analyze expected height of **recursion tree**.

Define $H(\pi, R) :=$ its height for instance $\pi$ and outcomes $R$.

$H^{exp}(n) = \max_\pi \sum_R \Pr(R) H(\pi, R)$.

# Expected recursion-depth for QuickSort




**Goal:** Analyze expected height of **recursion tree**.

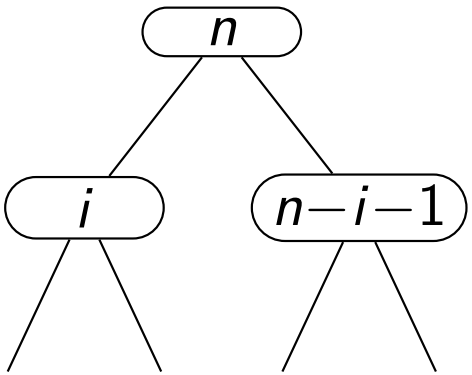
Define $H(\pi, R) :=$ its height for instance $\pi$ and outcomes $R$.

$H^{exp}(n) = \max_\pi \sum_R \Pr(R) H(\pi, R)$.

If $R$ lead to pivot-index $i$ (i.e., $R = \langle i, R' \rangle$) then

$$H(\pi, R) \leq 1 + \max\{H(\text{size-}i\text{-instance}, R'), H(\text{size-}(n-i-1)\text{-instance}, R')\}$$

# Expected recursion-depth for QuickSort




**Goal:** Analyze expected height of **recursion tree**.

Define $H(\pi, R) :=$ its height for instance $\pi$ and outcomes $R$.

$H^{exp}(n) = \max_\pi \sum_R \Pr(R) H(\pi, R)$.

If $R$ lead to pivot-index $i$ (i.e., $R = \langle i, R' \rangle$) then

$$H(\pi, R) \leq 1 + \max\{H(\text{size-}i\text{-instance}, R'), H(\text{size-}(n-i-1)\text{-instance}, R')\}$$

Summing up over all $R$, we can show (similar as for *expectedDemo*):

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

# Expected recursion-depth for QuickSort

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

**Claim:** $H^{exp}(n) \leq O(\log n)$.

**Proof:**

# Expected recursion-depth for QuickSort

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

**Claim:** $H^{exp}(n) \leq O(\log n)$.
**Proof:**

- So expected height of recursion tree is $H(n) \in O(\log n)$.
- We do $\Theta(n)$ work on each level of the recursion tree.
  $\Rightarrow$ Expected run-time of *RandomizedQuickSelect* is $O(n \log n)$.

# Expected recursion-depth for QuickSort

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

**Claim:** $H^{exp}(n) \leq O(\log n)$.
**Proof:**

- So expected height of recursion tree is $H(n) \in O(\log n)$.
- We do $\Theta(n)$ work on each level of the recursion tree.
  $\Rightarrow$ Expected run-time of *RandomizedQuickSelect* is $O(n \log n)$.
  $\Rightarrow$ Avg-case run-time *QuickSelect* is $O(n \log n)$.

# Improvement ideas for QuickSort

- The auxiliary space is $\Omega$(recursion depth).
  - This is $\Theta(n)$ in the worst-case, $\Theta(\log n)$ in avg-case
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.

# Improvement ideas for QuickSort

- The auxiliary space is $\Omega$(recursion depth).
  - This is $\Theta(n)$ in the worst-case, $\Theta(\log n)$ in avg-case
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.

- One should stop recursing when $n \leq 10$.
  Run InsertionSort at the end; this sorts everything in $O(n)$ time since all items are within 10 units of their required position.

# Improvement ideas for QuickSort

- The auxiliary space is $\Omega$(recursion depth).
  - This is $\Theta(n)$ in the worst-case, $\Theta(\log n)$ in avg-case
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.

- One should stop recursing when $n \leq 10$.
  Run InsertionSort at the end; this sorts everything in $O(n)$ time since all items are within 10 units of their required position.

- Arrays with many duplicates can be sorted faster by changing *partition* to produce three subsets

| $\leq v$ | $= v$ | $\geq v$ |
|---|---|---|

# Improvement ideas for QuickSort

- The auxiliary space is $\Omega$(recursion depth).
  - This is $\Theta(n)$ in the worst-case, $\Theta(\log n)$ in avg-case
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.

- One should stop recursing when $n \leq 10$.
  Run InsertionSort at the end; this sorts everything in $O(n)$ time since all items are within 10 units of their required position.

- Arrays with many duplicates can be sorted faster by changing *partition* to produce three subsets

| $\leq v$ | $= v$ | $\geq v$ |
|---|---|---|

- Two programming tricks that apply in many situations:
  - Instead of passing full arrays, pass only the range of indices.
  - Avoid recursion altogether by keeping an explicit stack.

# QuickSort with tricks

```
QuickSortImproved(A, n)
1.    Initialize a stack S of index-pairs with { (0, n−1) }
2.    while S is not empty
3.        (ℓ, r) ← S.pop()
4.        while (r−ℓ+1 > 10) do
5.            p ← choose-pivot-improved(A, ℓ, r)
6.            i ← partition-improved(A, ℓ, r, p)
7.            if (i−ℓ > r−i) do
8.                S.push( (ℓ, i−1) )
9.                ℓ ← i+1
10.           else
11.               S.push( (i+1, r) )
12.               r ← i−1
13.   InsertionSort(A)
```

This is often the most efficient sorting algorithm in practice (but worst-case time is still $\Theta(n^2)$).

# Outline

# 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* | $\Theta(n \log n)$ | average-case |
| *RandomizedQuickSort* | $\Theta(n \log n)$ | expected |

**Question**: Can one do better than $\Theta(n \log n)$ running time?
**Answer**: Yes and no! *It depends on what we allow*.

- No: Comparison-based sorting lower bound is $\Omega(n \log n)$.
- Yes: Non-comparison-based sorting can achieve $O(n)$ (under restrictions!). $\rightarrow$ see below

# The Comparison Model

In the **comparison model** data can only be accessed in two ways:

- comparing two elements
- moving elements around (e.g. copying, swapping)

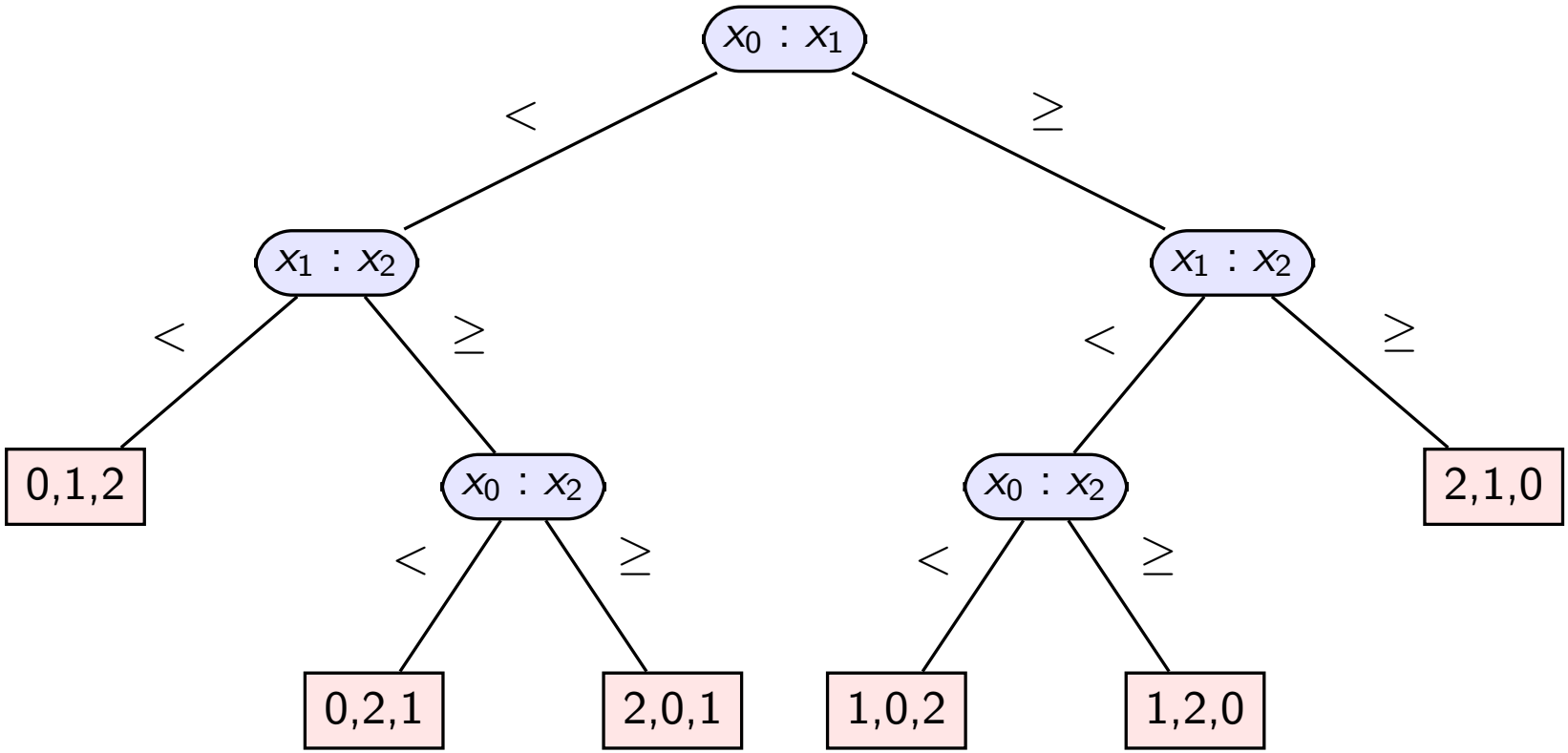
This makes very few assumptions on the kind of things we are sorting.
We count the number of above operations.

All sorting algorithms seen so far are in the comparison model.

# Decision trees

Comparison-based algorithms can be expressed as **decision tree**.

To sort $\{x_0, x_1, x_2\}$:

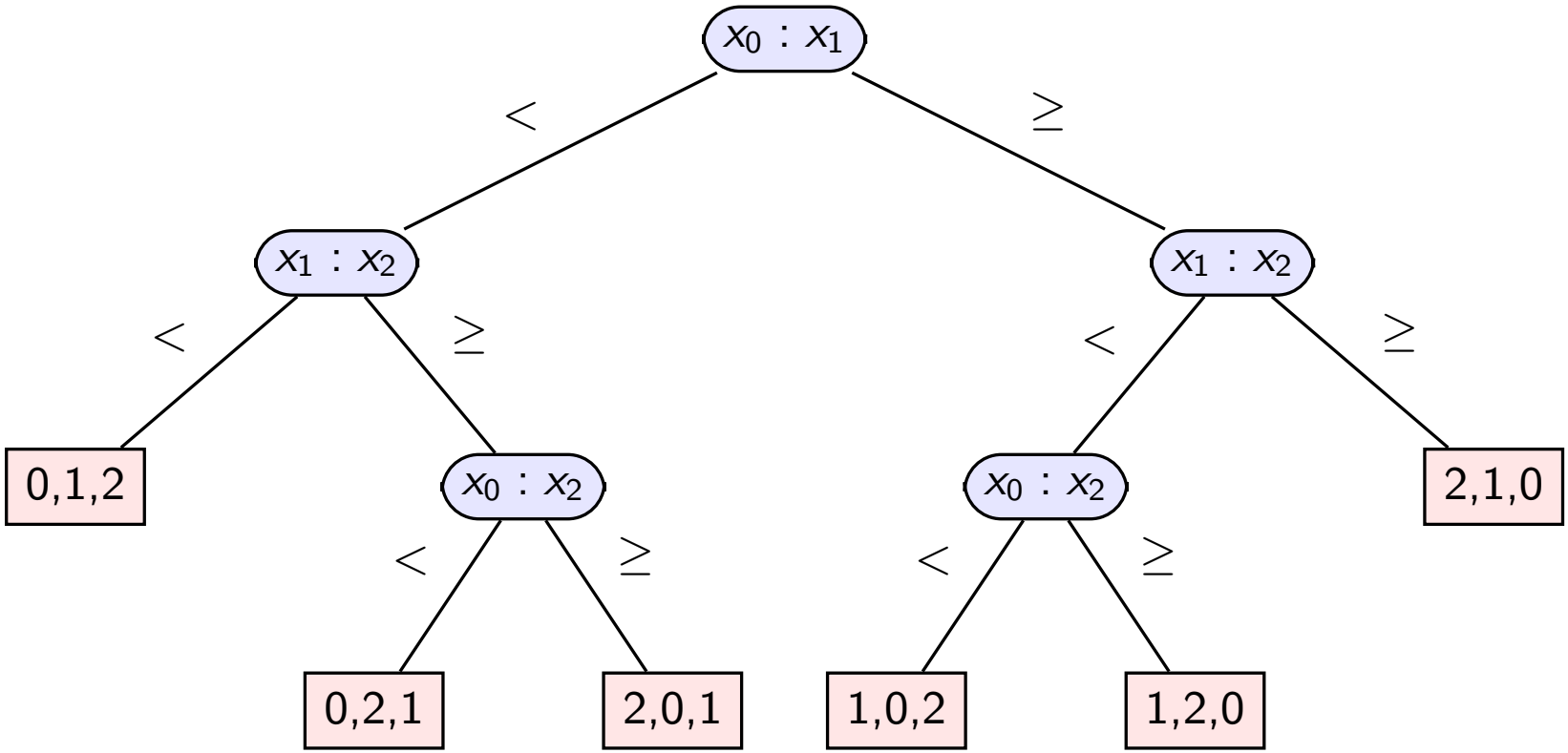


# Decision trees

Comparison-based algorithms can be expressed as **decision tree**.

To sort $\{x_0, x_1, x_2\}$:

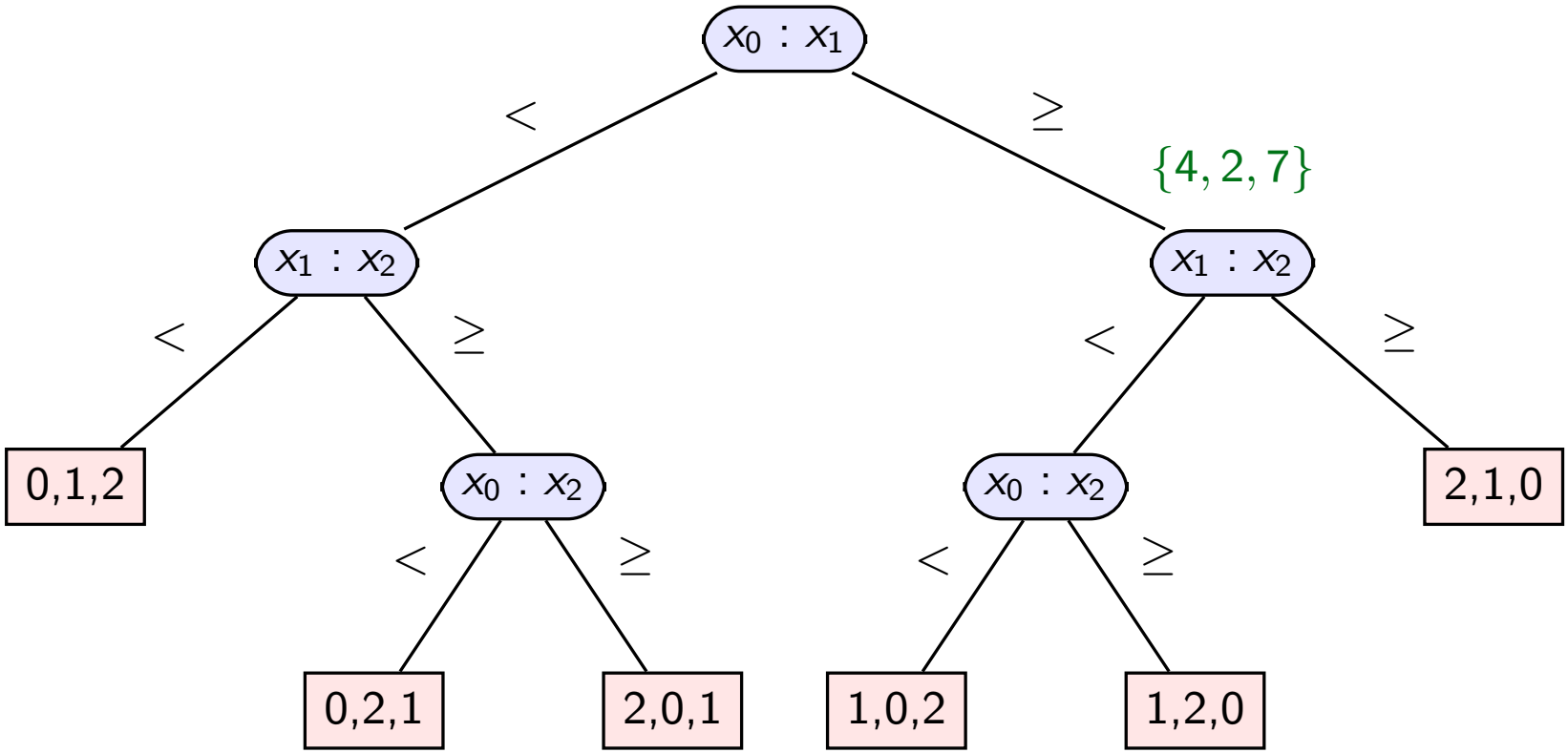
Example: $\{x_0{=}4, x_1{=}2, x_2{=}7\}$



# Decision trees

Comparison-based algorithms can be expressed as **decision tree**.

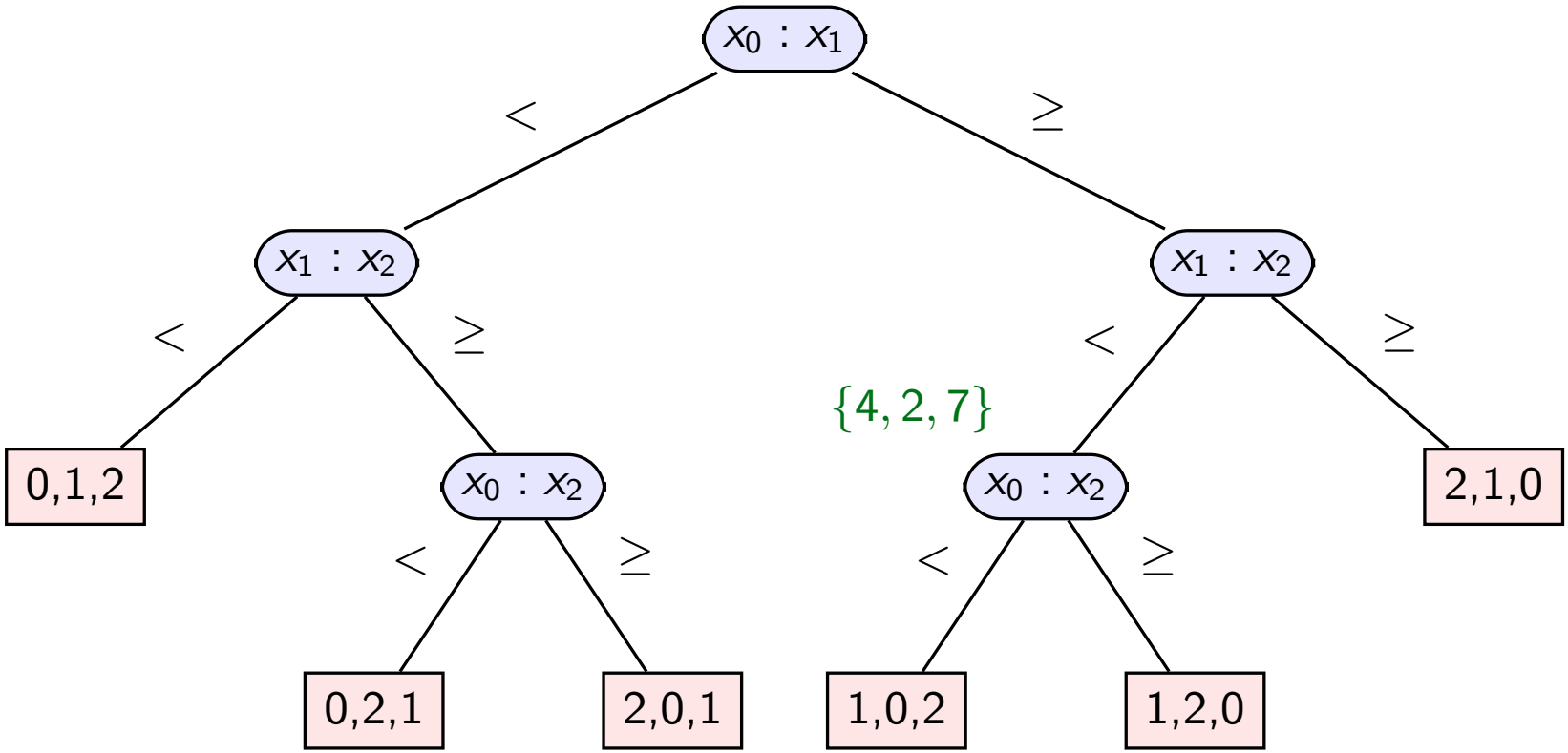
To sort $\{x_0, x_1, x_2\}$:



# Decision trees

Comparison-based algorithms can be expressed as **decision tree**.

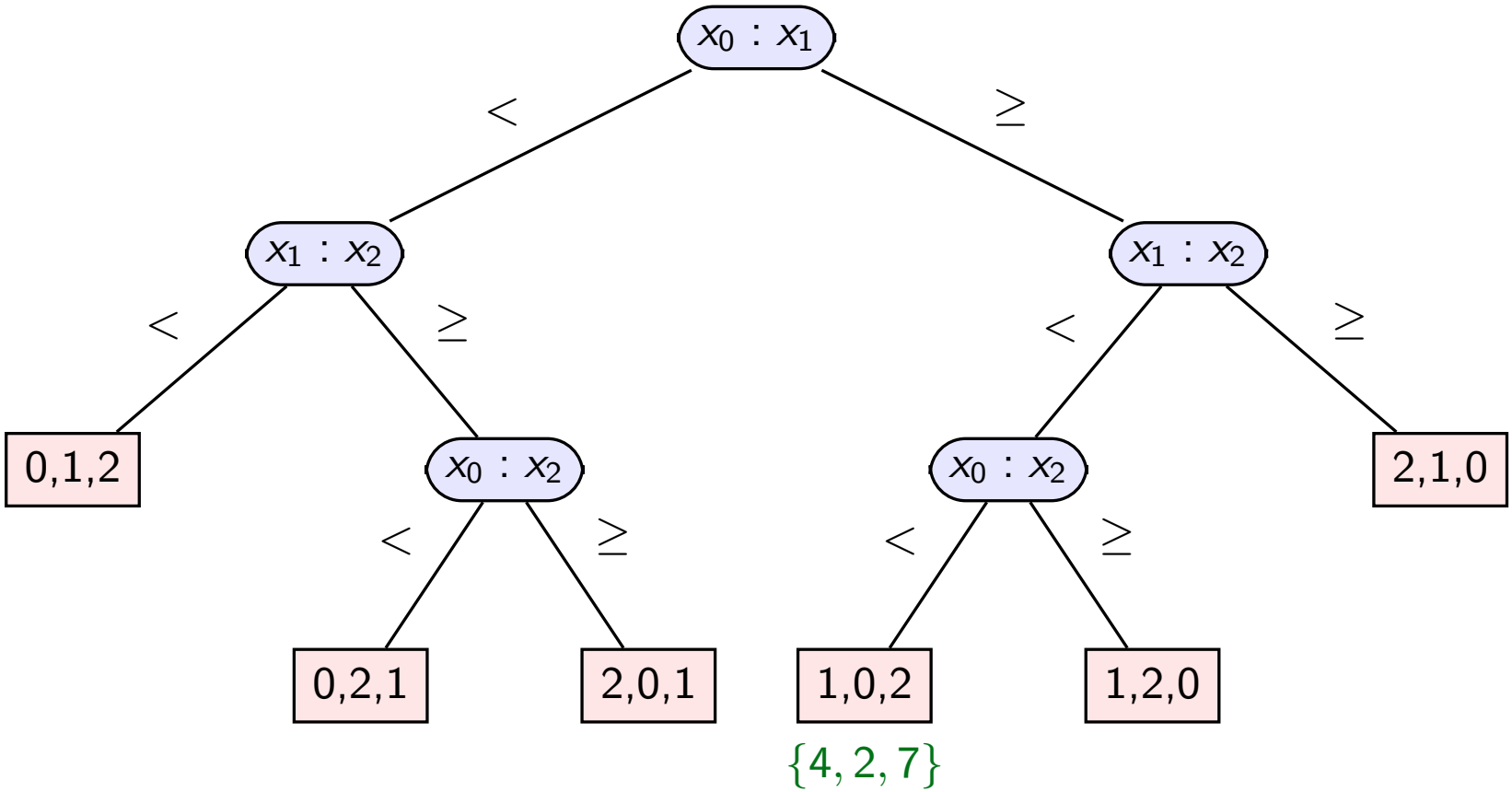
To sort $\{x_0, x_1, x_2\}$:



# Decision trees

Comparison-based algorithms can be expressed as **decision tree**.

To sort $\{x_0, x_1, x_2\}$:




Output: $\{4, 2, 7\}$ has sorting permutation $\langle 1, 0, 2 \rangle$

# Lower bound for sorting in the comparison model

**Theorem**. Any correct *comparison-based* sorting algorithm requires at least $\Omega(n \log n)$ comparison operations to sort $n$ distinct items.

**Proof**.

# Outline

# Non-Comparison-Based Sorting

- Assume keys are numbers in base $R$ ($R$: **radix**)
  - $R = 2, 10, 128, 256$ are the most common.

Example ($R = 4$):

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

- Assume all keys have the same number $m$ of digits.
  - Can achieve after padding with leading 0s.

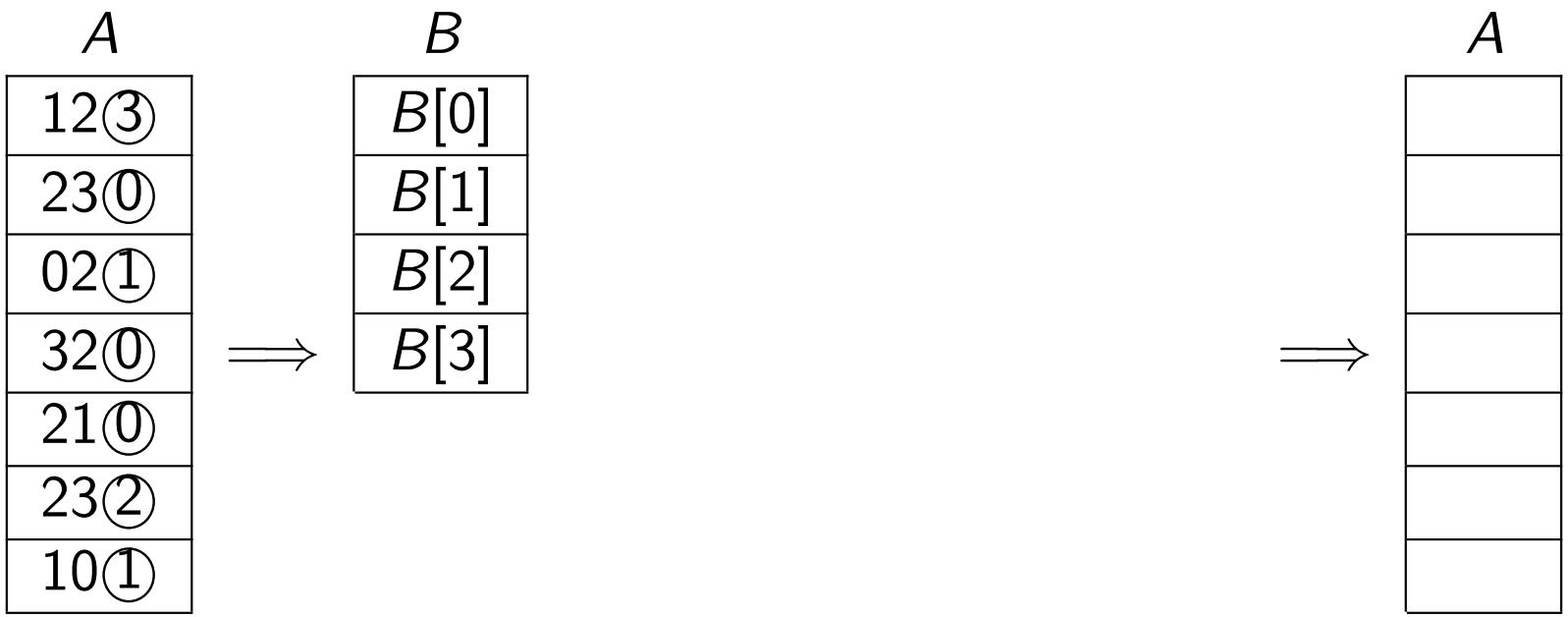
Example ($R = 4$):

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

- Can sort based on individual digits.
  - How to sort 1-digit numbers?
  - How to sort multi-digit numbers based on this?

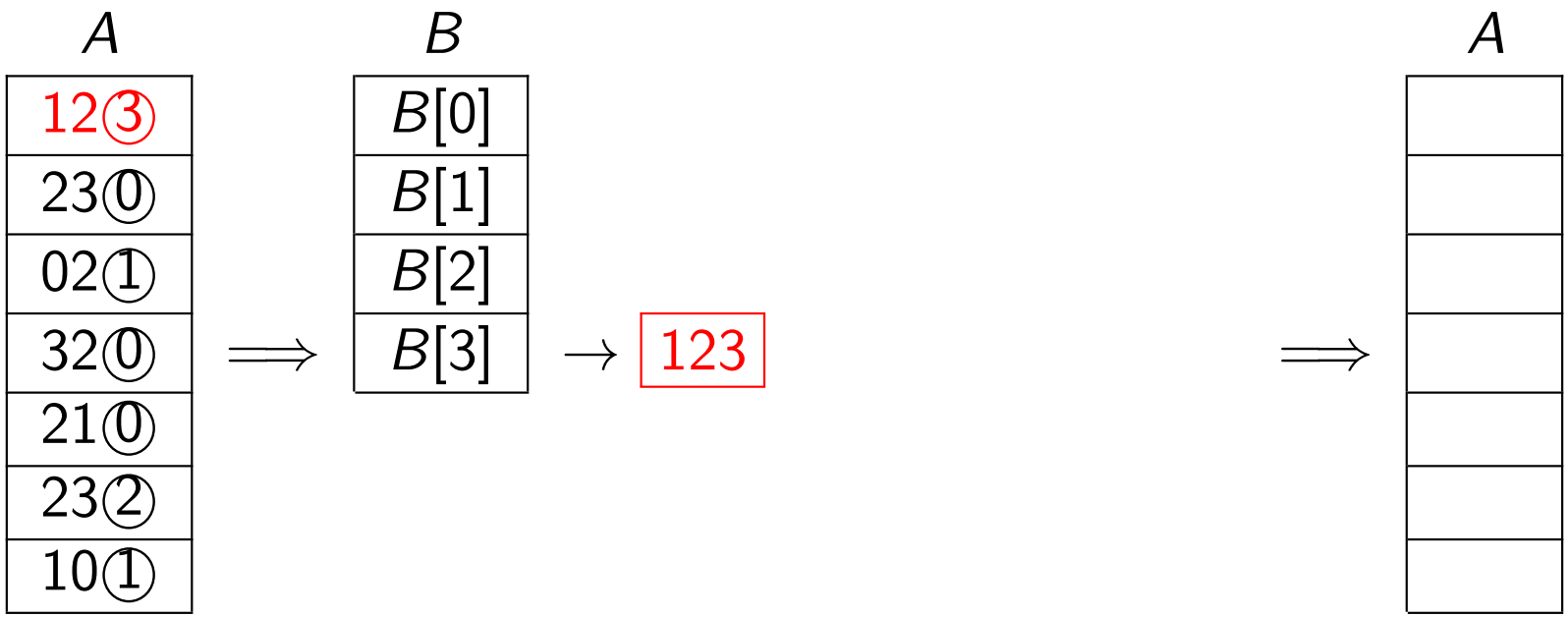
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



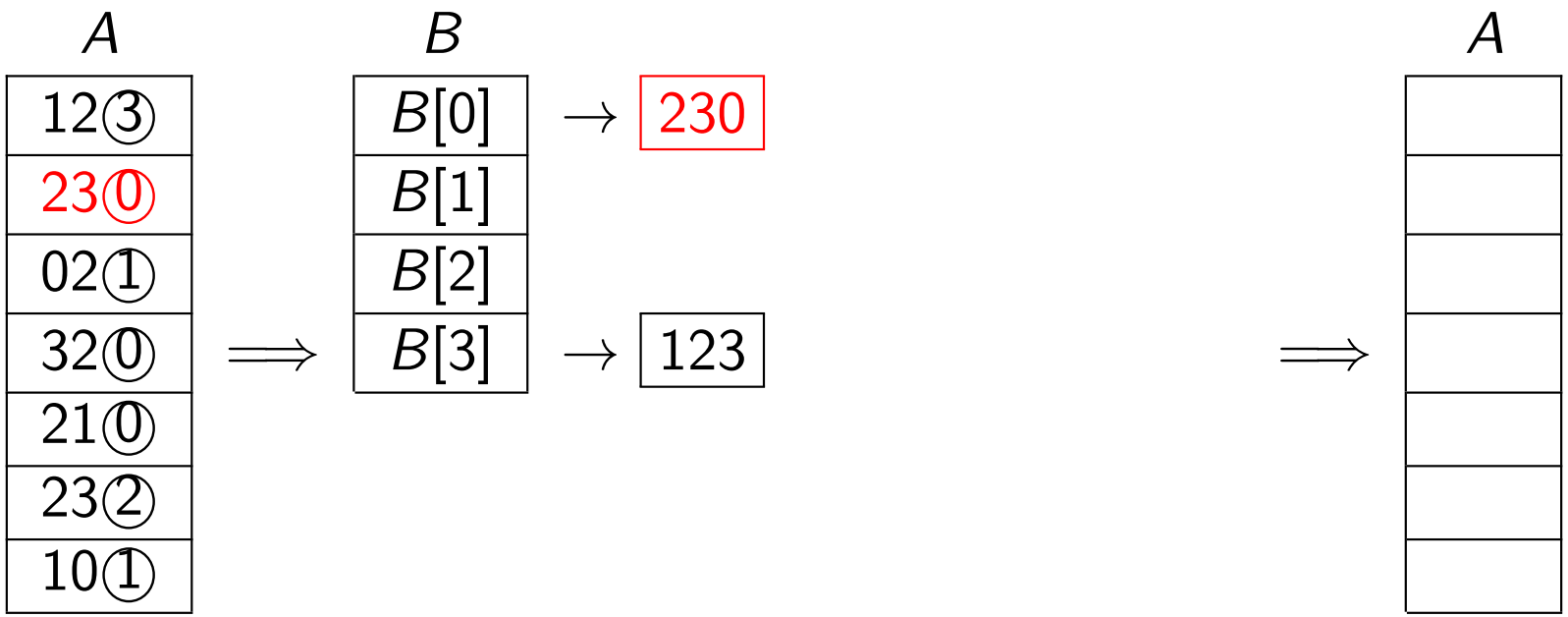
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



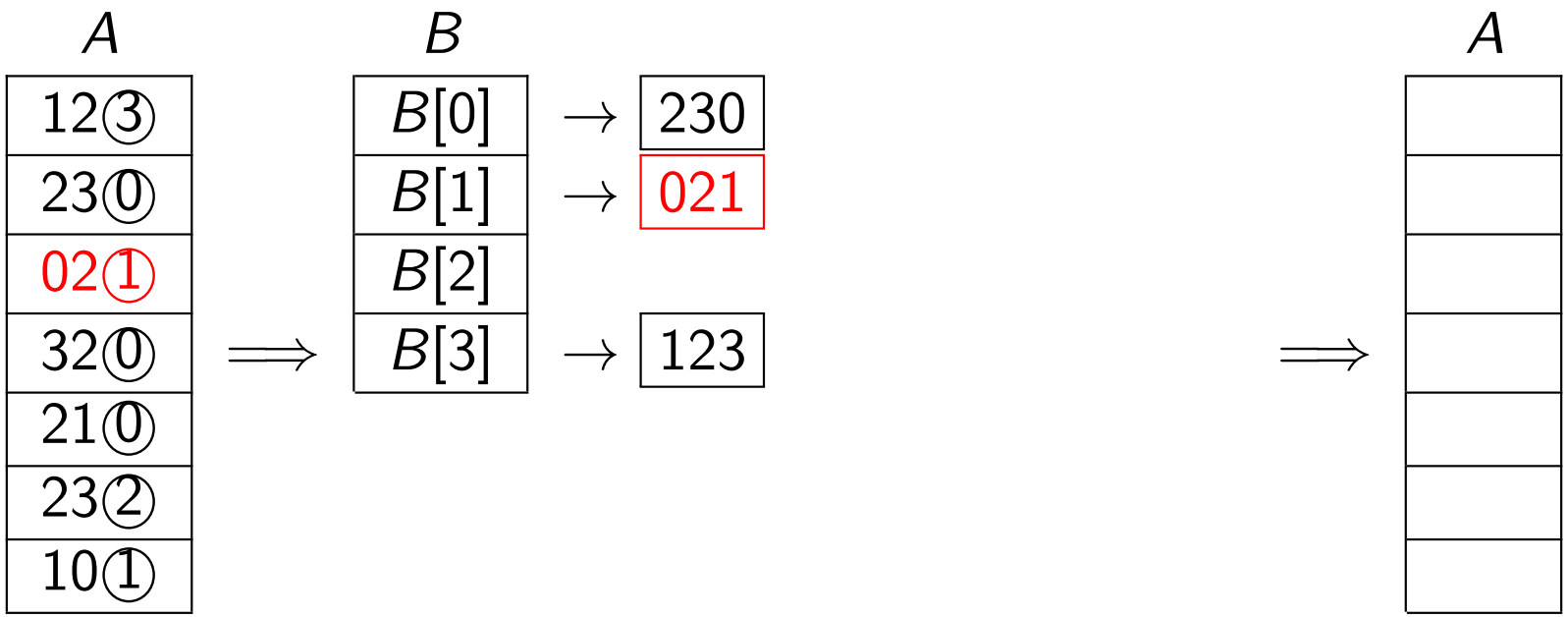
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



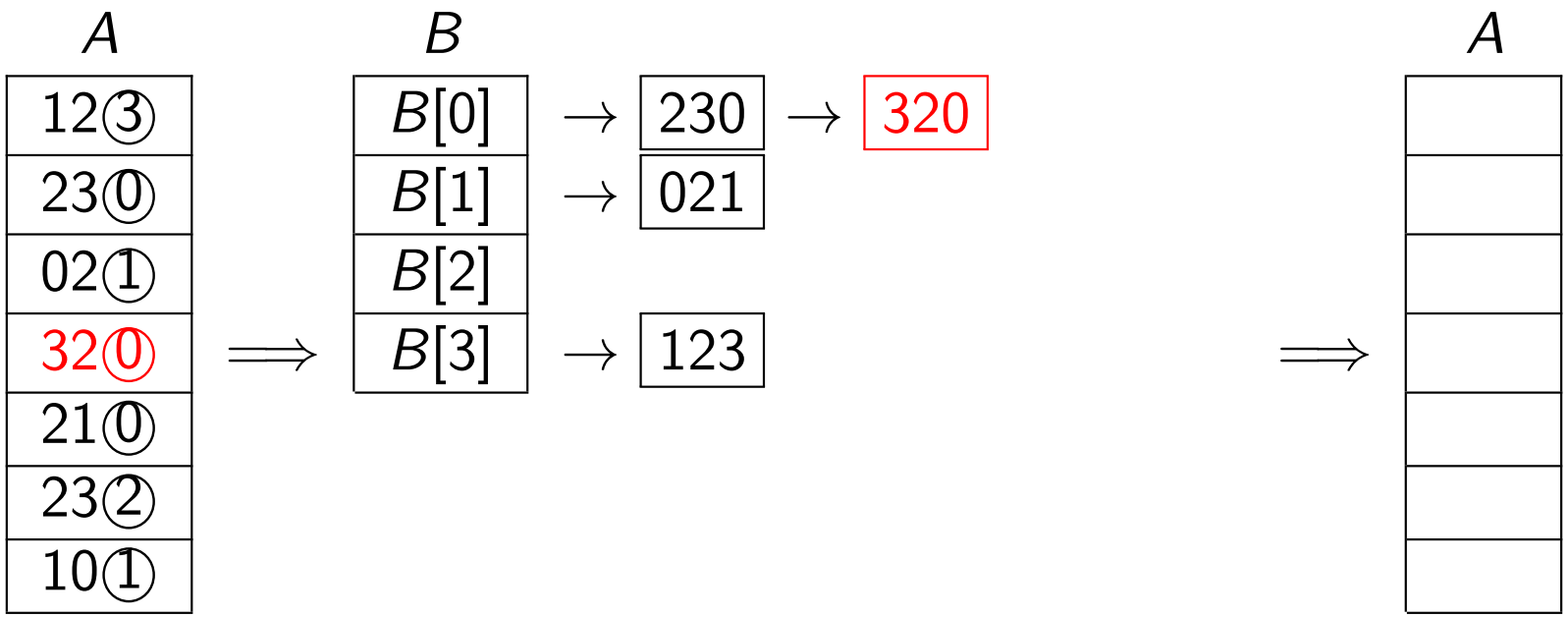
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



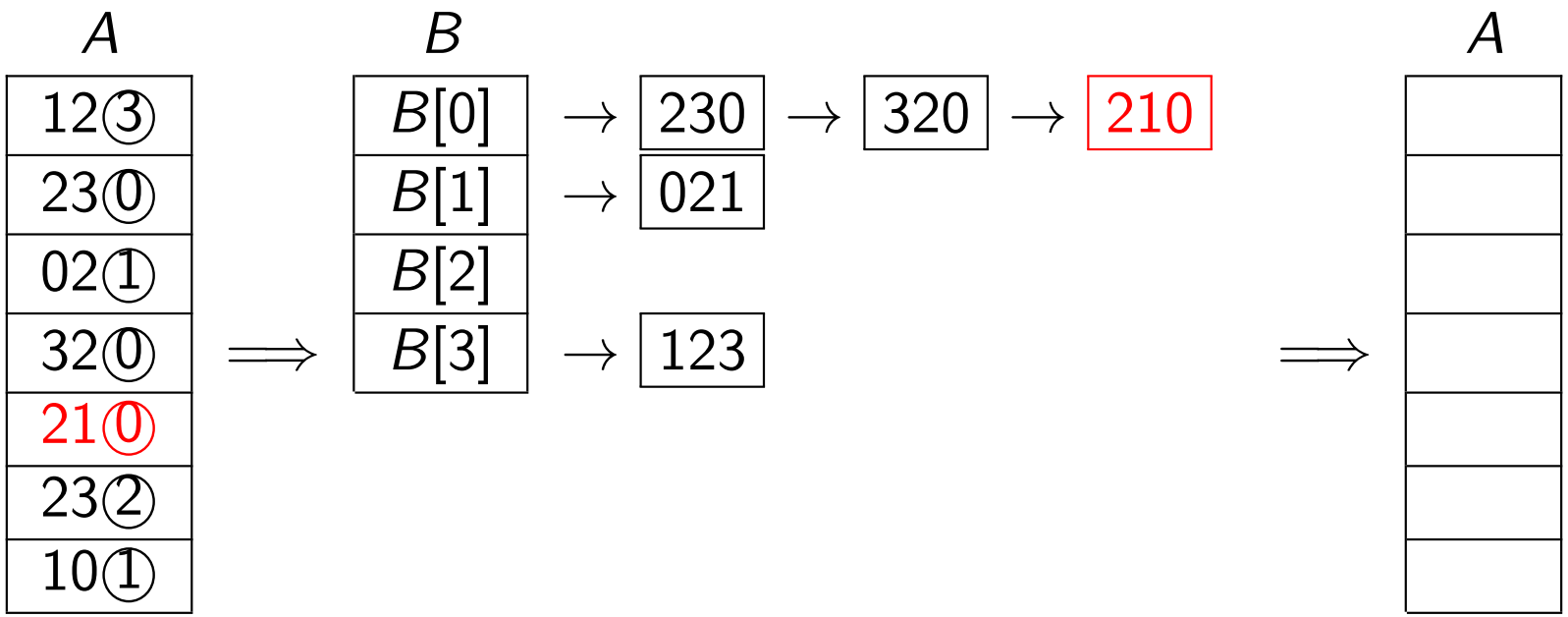
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



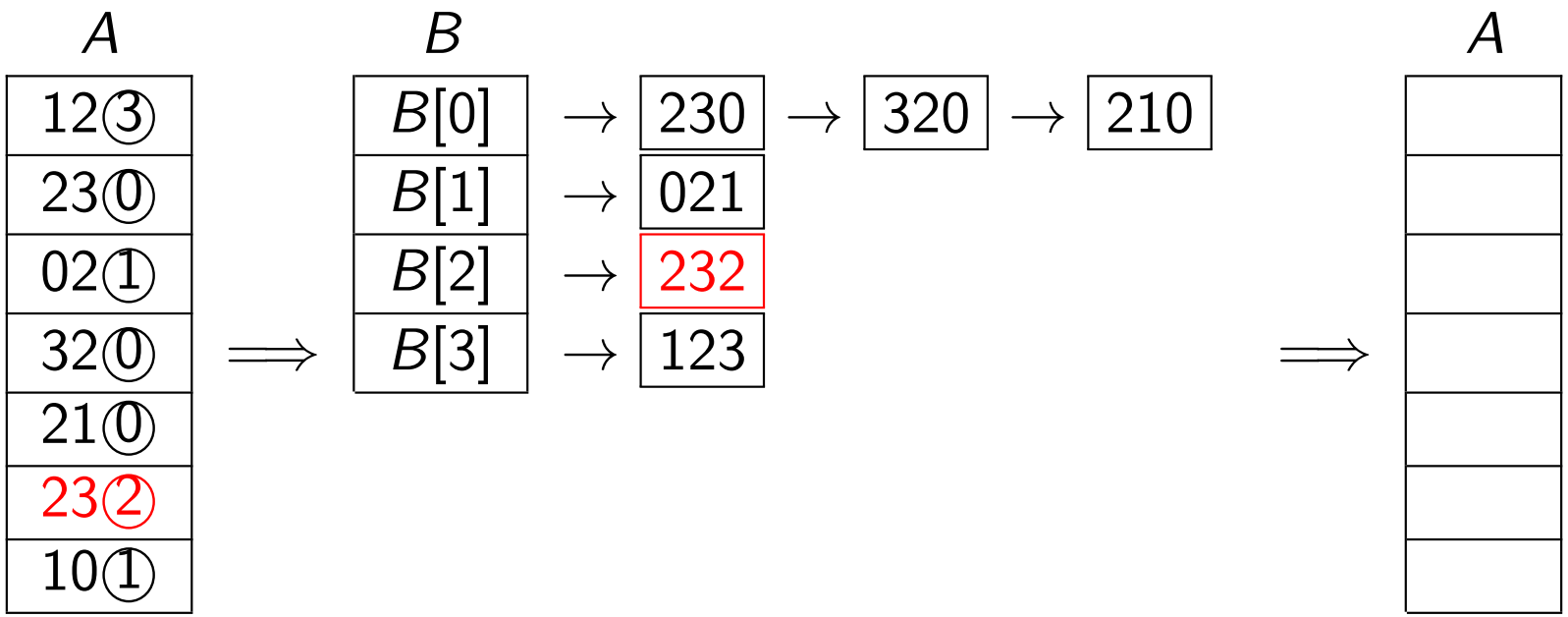
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



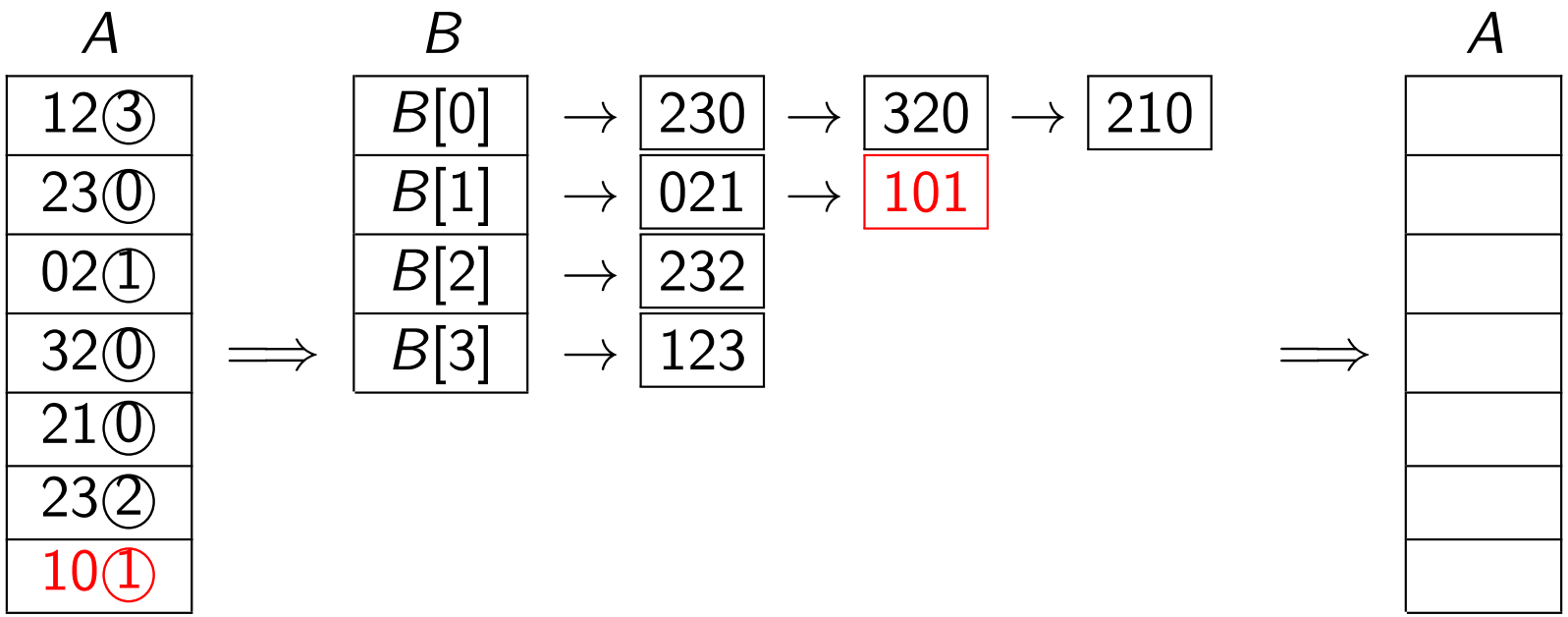
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



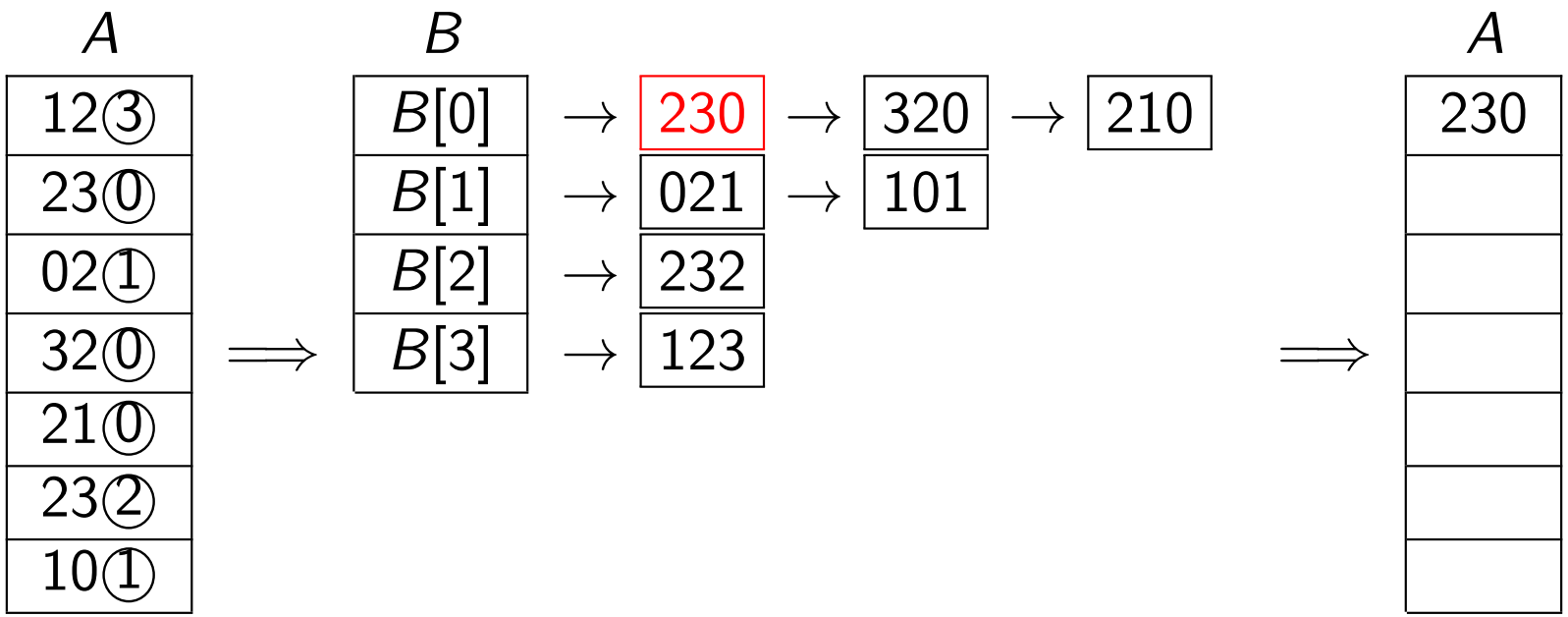
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



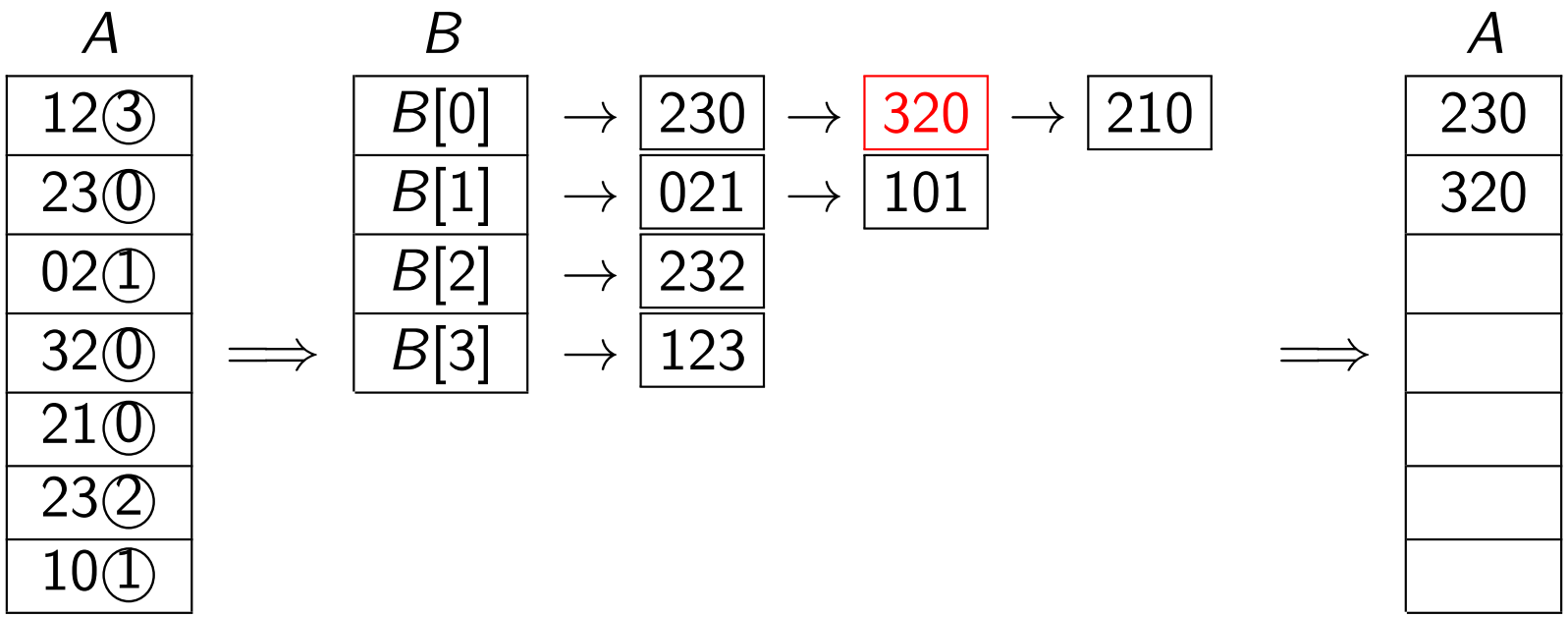
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



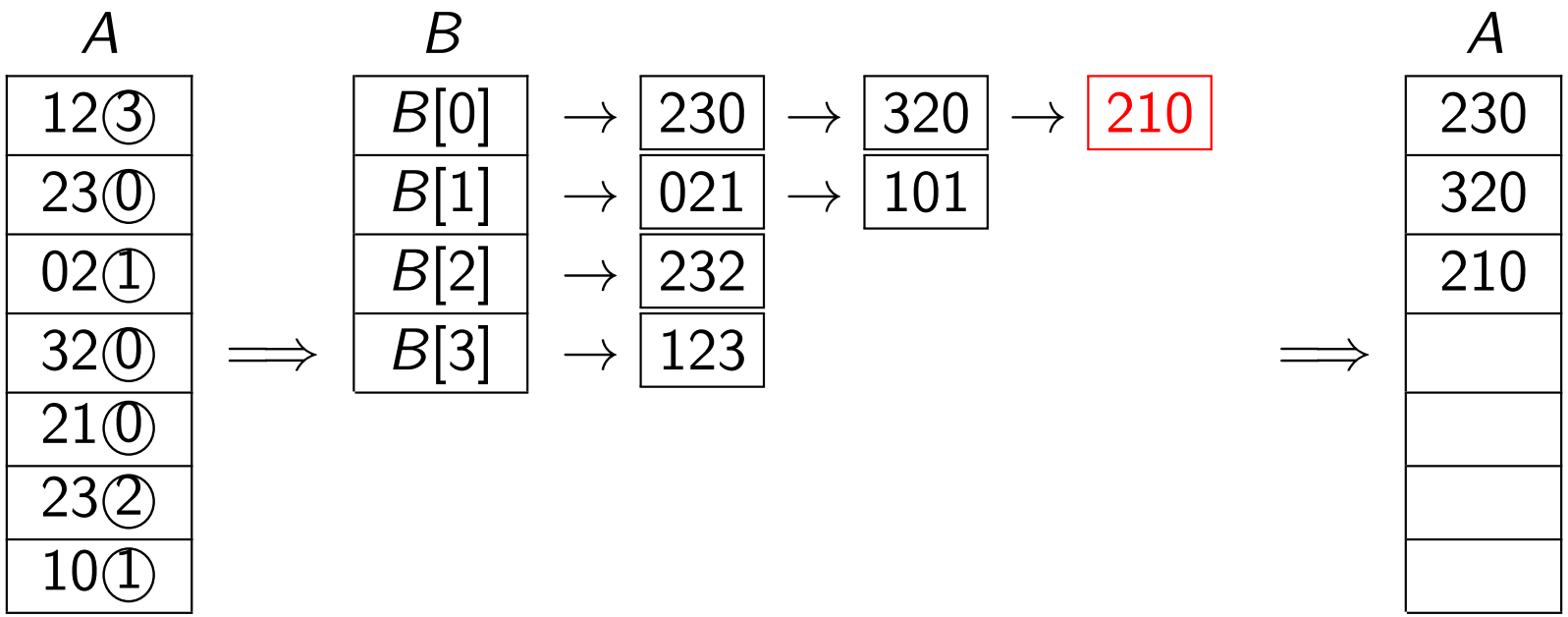
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



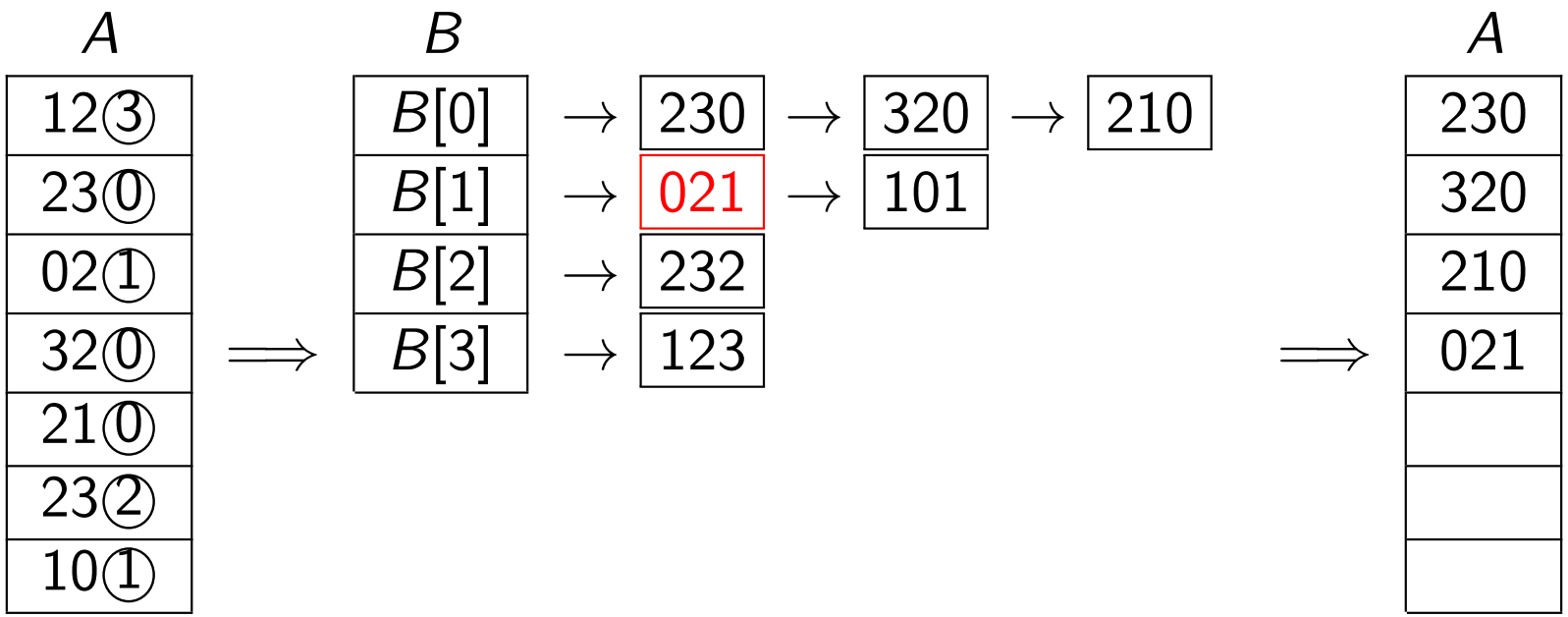
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



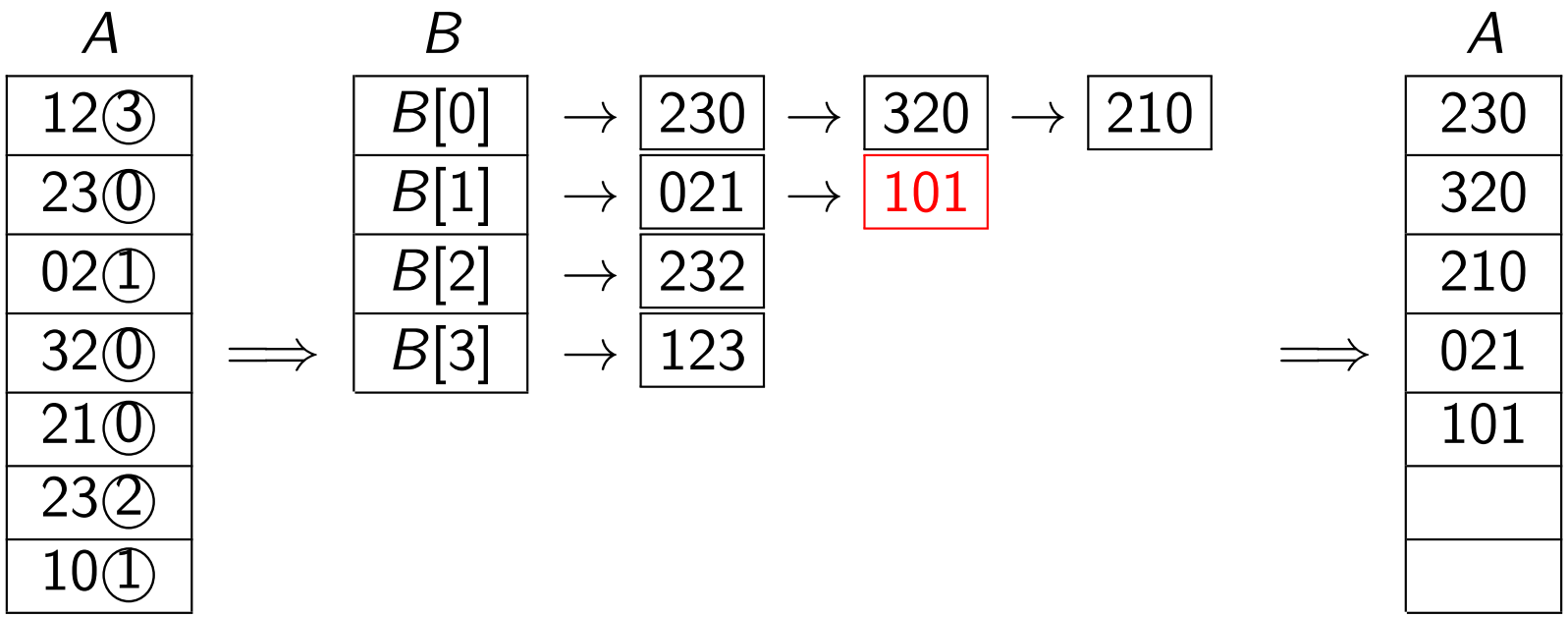
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



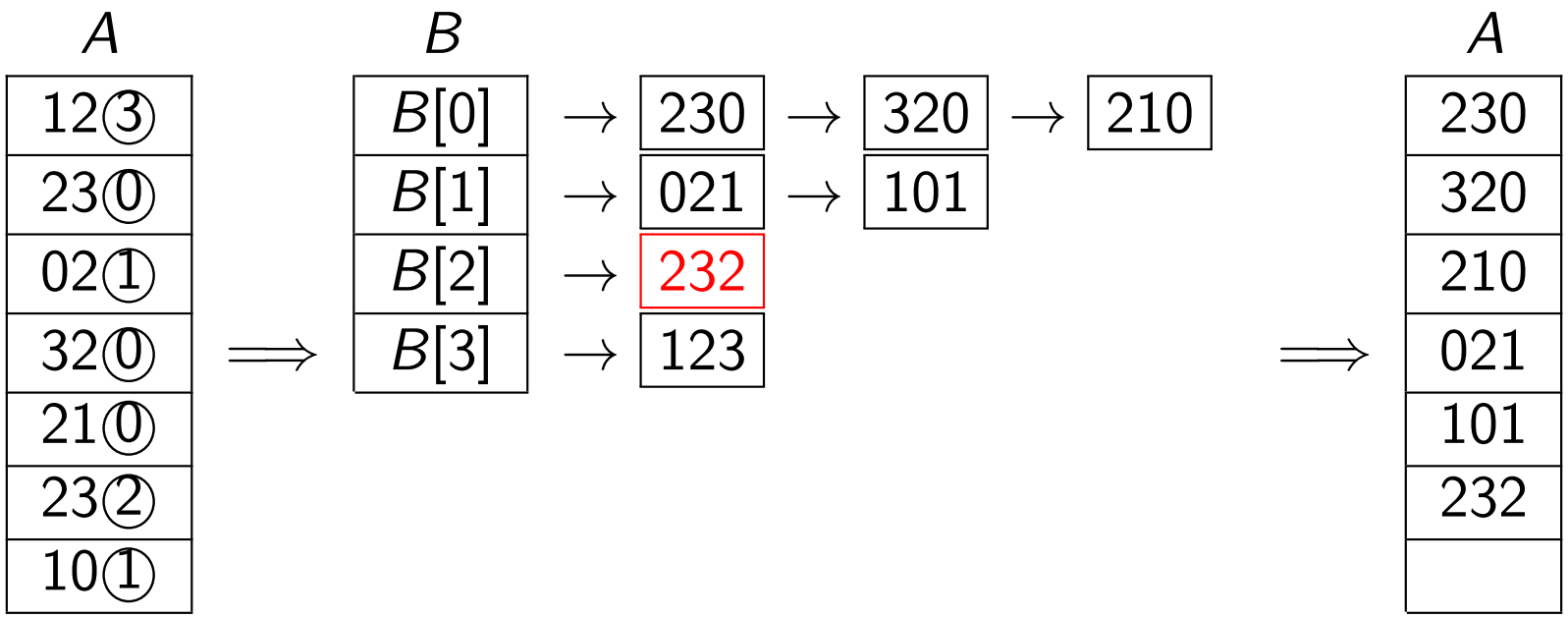
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



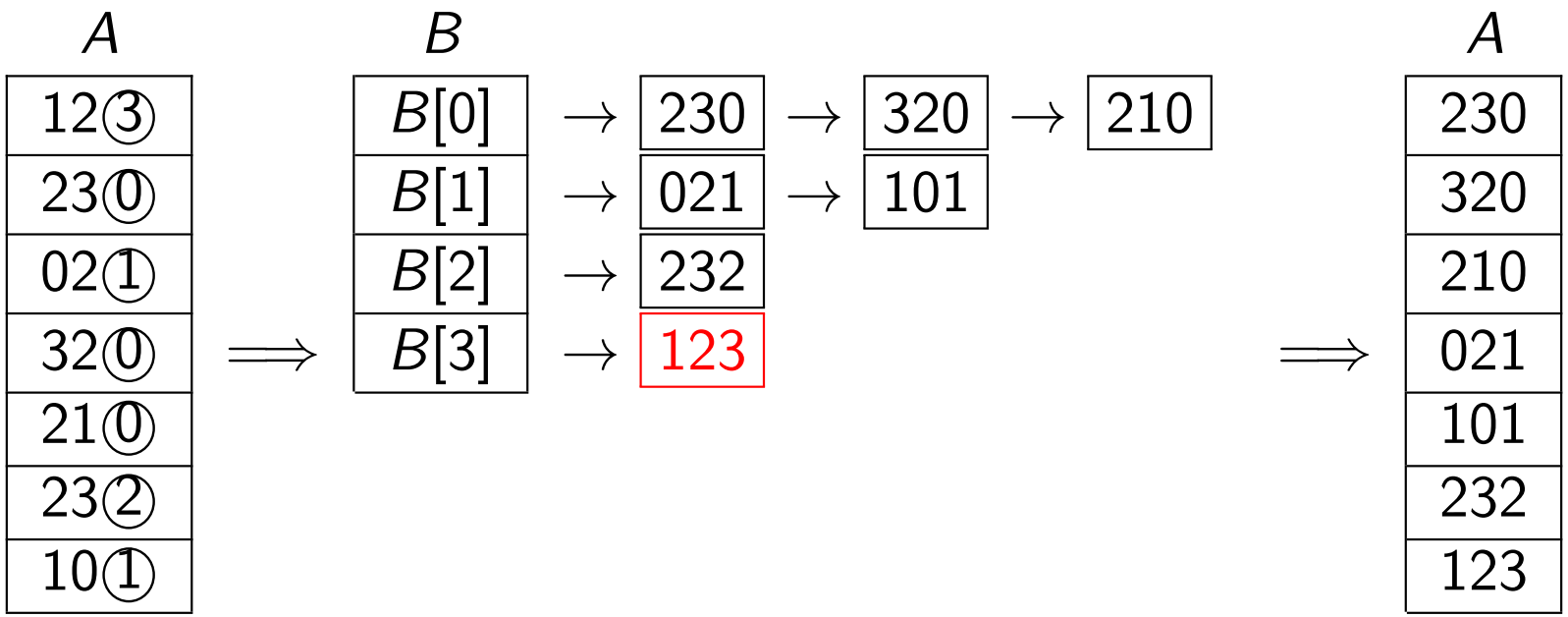
# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



# (Single-digit) Bucket Sort

Sort array $A$ by last digit:



# (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

```
1.    Initialize an array B[0...R − 1] of empty lists (buckets)
2.    for i ← 0 to n−1 do
3.        Append A[i] at end of B[d^th digit of A[i]]
4.    i ← 0
5.    for j ← 0 to R − 1 do
6.        while B[j] is non-empty do
7.            move first element of B[j] to A[i++]
```

- Sorts numbers by single digit (specified by user).
- This is **stable**: equal items stay in original order.
- Run-time $\Theta(n + R)$, auxiliary space $\Theta(n + R)$
- It is possible to replace the lists by two auxiliary arrays of size $R$ and $n \rightsquigarrow$ *count-sort* (no details).

# MSD-Radix-Sort

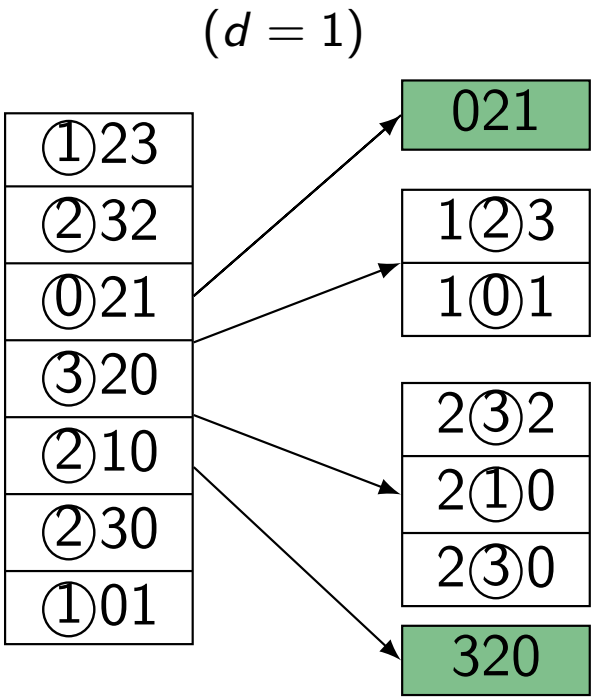
Sorts array of $m$-digit radix-$R$ numbers recursively:
sort by leading digit, then each group by next digit, etc.

```
MSD-Radix-sort(A,  ℓ ← 0,  r ← n−1,  d ← index of leading digit)
ℓ, r: range of what we sort, 0 ≤ ℓ, r ≤ n−1
 1.    if ℓ < r
 2.        bucket-sort(A[ℓ..r], d)
 3.        if there are digits left   // recurse in sub-arrays
 4.            ℓ′ ← ℓ
 5.            while (ℓ′ < r) do
 6.                Let r′ ≥ ℓ′ be maximal s.t. A[ℓ′..r′] all have same dth digit
 7.                MSD-Radix-sort(A, ℓ′, r′, d+1)
 8.                ℓ′ ← r′ + 1
```

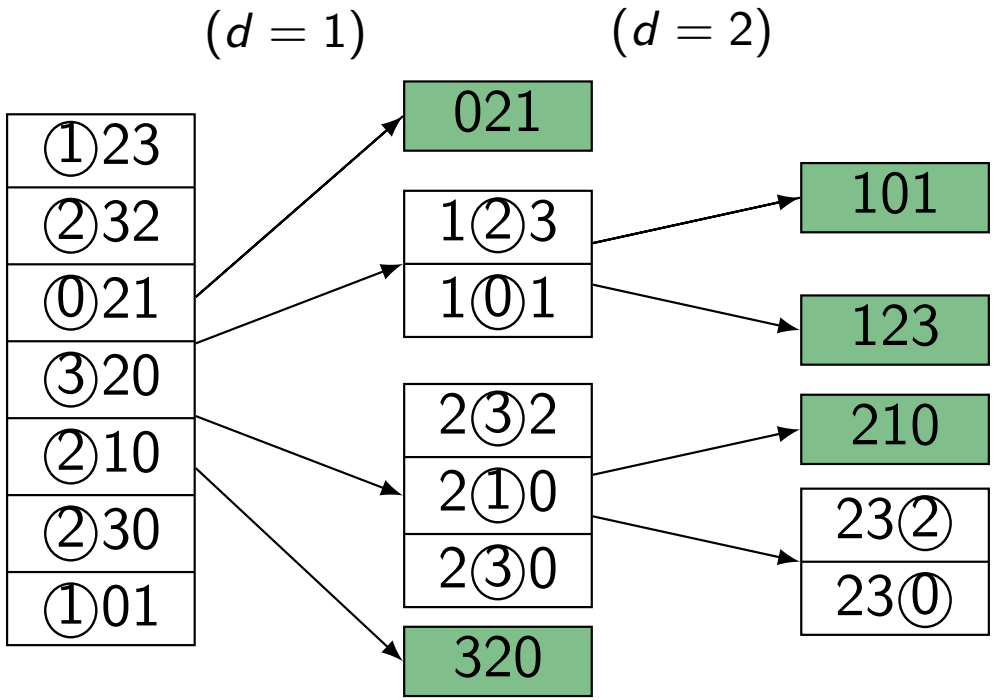
# MSD-Radix-Sort Example

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

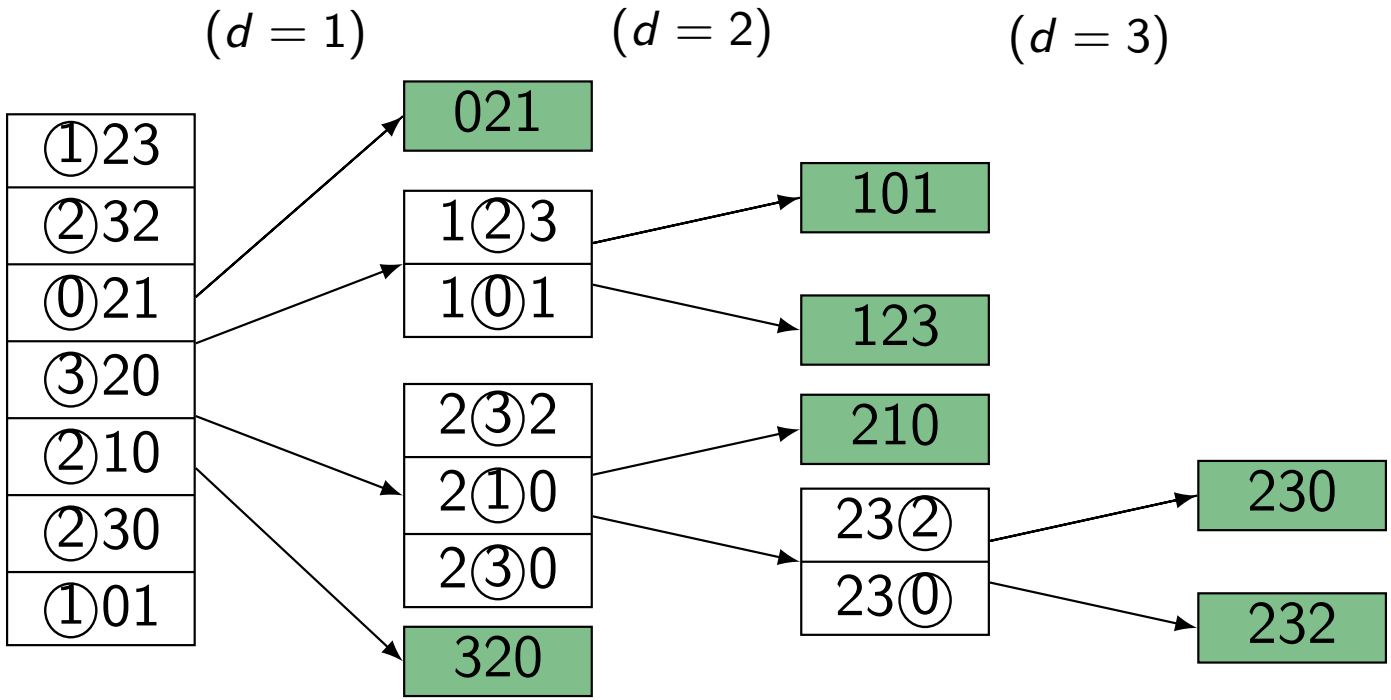
# MSD-Radix-Sort Example



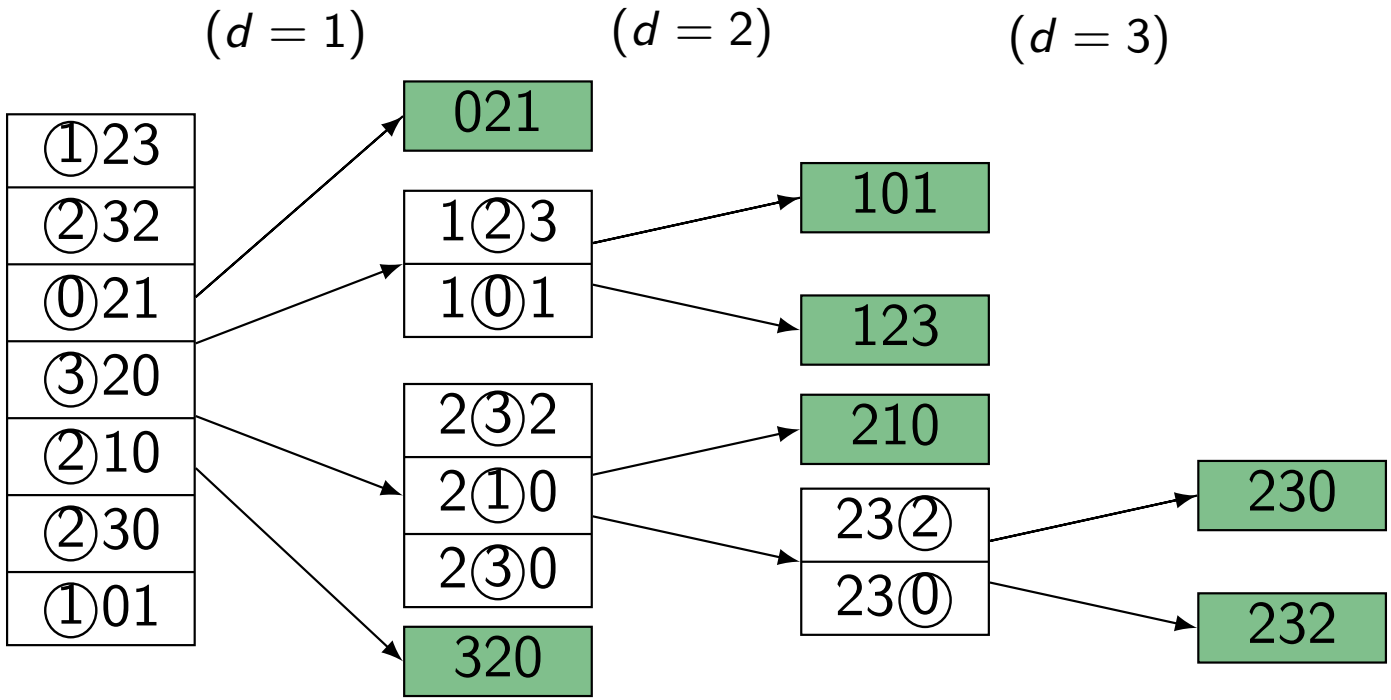
# MSD-Radix-Sort Example



# MSD-Radix-Sort Example



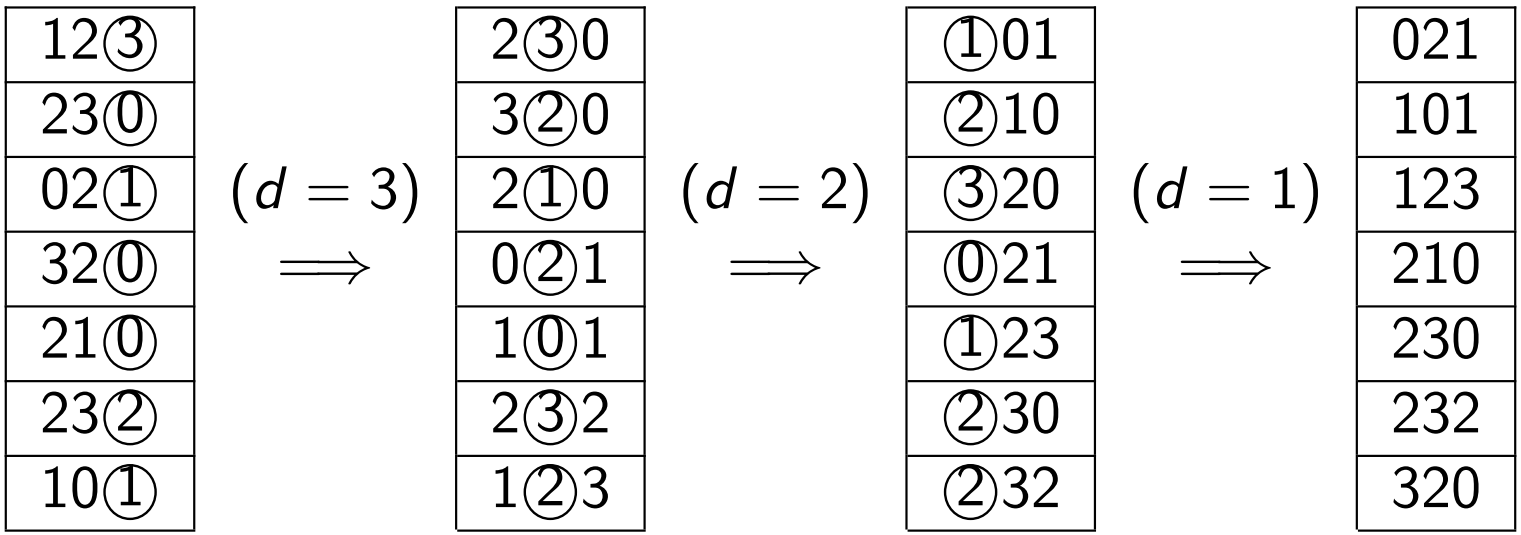
# MSD-Radix-Sort Example




- Drawback of *MSD-Radix-Sort*: many recursions
- **Auxiliary space**: $\Theta(n + R + m)$ (for *bucket-sort* and recursion stack)
- **Run-time**: $\Theta(mnR)$ since we may have $\Theta(mn)$ subproblems.

# LSD-Radix-Sort

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




- Loop-invariant: $A$ is sorted w.r.t. digits $d, \ldots, m$ 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 $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.
- *QuickSort* is worst-case $\Theta(n^2)$, but often the fastest in practice
- *CountSort* and *RadixSort* achieve $o(n \log n)$ if the input is special

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case, expected-case can all differ, but for well-design randomizations of algorithms, the expected case is the same as the average-case of the non-randomized algorithm.