CS 240 – Data Structures and Data Management

Module 3: Sorting, Average-case and Randomization

Leili Rafiee Sevyeri   Éric Schost
Based on lecture notes by many previous cs240 instructors

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

Spring 2023

version *2023-06-07 16:29*

Outline

3. 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

3. 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

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
    if n ≤ 2 return
    if A[n−2] < A[n−1]
        avgCaseDemo(A[0..n/2−1], n/2)      // Good case
    else avgCaseDemo(A[0..n−3], n−2)       // Bad case
```

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

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

A contrived example

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

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

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

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

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

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

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

Observe: $\pi^{-1} = \langle\ 6,\ 2,\ 1,\ 3,\ 0,\ 5,\ 4\ \rangle$
seen as an array, π^{-1} 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: $A = [\ 14,\ 3,\ 2,\ 6,\ 1,\ 11,\ 7\ ]$
$\pi = \langle\ 4,\ 2,\ 1,\ 3,\ 6,\ 5,\ 0\ \rangle$

Observe: $\pi^{-1} = \langle\ 6,\ 2,\ 1,\ 3,\ 0,\ 5,\ 4\ \rangle$
seen as an array, π^{-1} has same sorting permutation as A.

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

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

$$\begin{aligned} T(\pi) &= \text{runtime on any instance with sorting permutation } \pi \\ &= \text{runtime on } \pi^{-1} \text{ (seen as an array)} \end{aligned}$$

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 π:

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

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 π:

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

Recursive formula for all instances π 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*

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

Average-case run-time of *avgCaseDemo*

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

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

Average-case run-time of *avgCaseDemo*

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

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

Average-case run-time of *avgCaseDemo*

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

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

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

3. 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

- 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 choices 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 choices 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 choices 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
    if n ≤ 2 return
    if random(2) swap A[n−1] and A[n−2]
    if A[n−2] ≤ A[n−1]
        expectedDemo(A[0..n/2−1], n/2)      // Good case
    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: $\Pr(\text{good case}) = \frac{1}{2} = \Pr(\text{bad case})$.

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:

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

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:

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

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

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

Expected run-time of *expectedDemo*

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

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

Expected run-time of *expectedDemo*

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

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

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

Expected run-time of *expectedDemo*

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

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

- 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

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

The Selection Problem

The **selection problem**: Given an array A of n numbers, and $0 \le k < n$, 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:

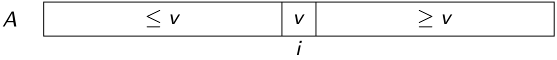
- *choose-pivot*(A): Return an index p in A. We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.

  Simplest idea: Always select rightmost element in array

  ```
  choose-pivot(A)
      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

Easy linear-time implementation:

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

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

QuickSelect Algorithm

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

Idea: After partition have

$\leq v$	v	$\geq v$
	i	

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

Analysis of *QuickSelect*

Let $T(A,k)$ be the number of **key-comparisons** in a size-n array A with parameter k. (This is asymptotically the same as the run-time.) Also written $T(\pi,k)$ for any array with sorting permutation π.

partition uses n key-comparisons.

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

Analysis of *QuickSelect*

Let $T(A, k)$ be the number of **key-comparisons** in a size-n array A with parameter k. (This is asymptotically the same as the run-time.) Also written $T(\pi, k)$ for any array with sorting permutation π.

partition uses n key-comparisons.

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

Best-case analysis: First chosen pivot could be the kth element
No recursive calls; $T^{best}(n, k) = n \in \Theta(n)$

Analysis of *QuickSelect*

Let $T(A, k)$ be the number of **key-comparisons** in a size-n array A with parameter k. (This is asymptotically the same as the run-time.) Also written $T(\pi, k)$ for any array with sorting permutation π.

partition uses n key-comparisons.

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

Best-case analysis: First chosen pivot could be the kth element
No recursive calls; $T^{best}(n, k) = n \in \Theta(n)$

Average case analysis?

Average-Case Analysis of *QuickSelect*

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

Average-Case Analysis of *QuickSelect*

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

Assume that sorting permutation π gives pivot-index i. If new array (after *partition*) is A', then

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

Average-Case Analysis of *QuickSelect*

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

Assume that sorting permutation π gives pivot-index i. If new array (after *partition*) is A', then

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

Claim (very complicated)

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

then (easy)

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

and finally $T^{avg}(n) \leq 4n$
(by induction, using $\sum_{i=0}^{n-1} \max(i, n-i-1) \leq 3/4 n^2$)

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** $i \leftarrow 1$ 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 (this is our prefered implementation)

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

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

Randomizing QuickSelect: Random Pivot

Second idea: Change the pivot selection (this is our prefered implementation)

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

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

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*

Recurrence: with $T^{exp}(n) = \max_{\pi,k} T^{exp}(\pi, k)$, we get

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

Consequence: $T^{exp}(n) \in O(n)$ (and so $\Theta(n)$)

$\Rightarrow$ *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)

Outline

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

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

QuickSort analysis

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

Worst-case analysis: Recursive call could always have size $n{-}1$.
$T^{worst}(n) \geq n + T^{worst}(n{-}1) \in \Omega(n^2)$ 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 $\leq n/2$.
$T^{best}(n) \leq n + 2T^{best}(n/2) \in O(n \log n)$ exactly as for *MergeSort*
(This can be shown to be tight.)

Average-case analysis of *QuickSort*

Let $T^{\mathrm{avg}}(n)$ be the *average-case* number of comparisons for *QuickSort* in a size-n array.

Can prove (complicated)

$$\begin{aligned} T^{\mathrm{avg}}(n) &\leq n + \frac{1}{n}\sum_{i=0}^{n-1}\left(T^{\mathrm{avg}}(i) + T^{\mathrm{avg}}(n-i-1)\right) \\ &\leq n + \frac{2}{n}\sum_{i=0}^{n-1} T^{\mathrm{avg}}(i) \end{aligned}$$

Claim: $T(n) \leq 2n\log(n)$ (by induction on n)

Randomizing QuickSort

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

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

Randomizing QuickSort

RandomizedQuickSort(A)
1. ...
2. $p \leftarrow random(A.size)$
3. $i \leftarrow partition(A, p)$
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 $T^{exp}(n) \leq n + \frac{2}{n}\sum_{i=0}^{n-1} T^{exp}(i)$, so this in $O(n \log n)$

Optional: efficient In-Place partition (Hoare)

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

Optional: efficient In-Place partition (Hoare)

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

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

Optional: efficient In-Place partition (Hoare)

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

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

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

Optional: efficient In-Place partition (Hoare)

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

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

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

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

Optional: efficient In-Place partition (Hoare)

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

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

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

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

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

Optional: efficient In-Place partition (Hoare)

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

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

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

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

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

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

Optional: efficient In-Place partition (Hoare)

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

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

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

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

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

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

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

Optional: Efficient In-Place partition (Hoare)

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

Loop invariant:

A	$\leq v$	?	$\geq v$	v
	i		j	$n-1$

```
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: $\Theta(n)$.

Optional: Improvement ideas for QuickSort

- Pick the median of 3 or so pivot-candidates
- The auxiliary space is Ω(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.

Optional: Improvement ideas for QuickSort

- Pick the median of 3 or so pivot-candidates
- The auxiliary space is Ω(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.

Optional: Improvement ideas for QuickSort

- Pick the median of 3 or so pivot-candidates
- The auxiliary space is Ω(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

$\leq v$	$= v$	$\geq v$

Optional: Improvement ideas for QuickSort

- Pick the median of 3 or so pivot-candidates
- The auxiliary space is Ω(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.

Optional: QuickSort with tricks

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

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

Outline

3. 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	$\Theta(n \log n)$	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)

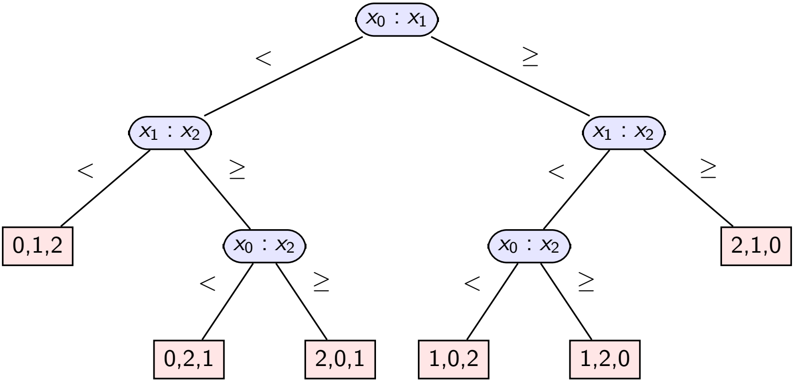
This makes very few assumptions on the kind of things we are sorting.
We count the number of above operations.

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

Decision trees

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

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

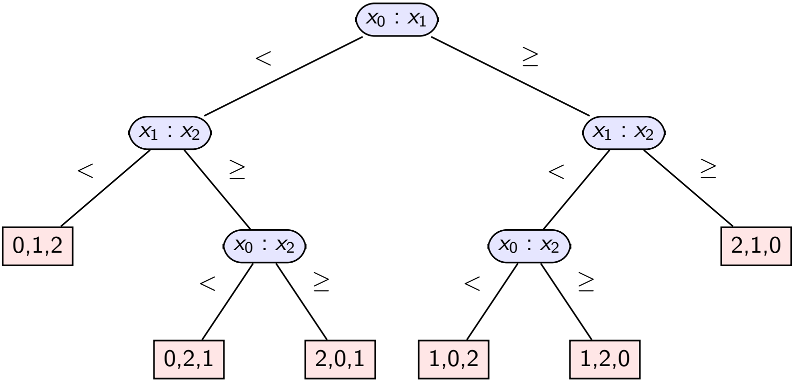


Decision trees

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

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

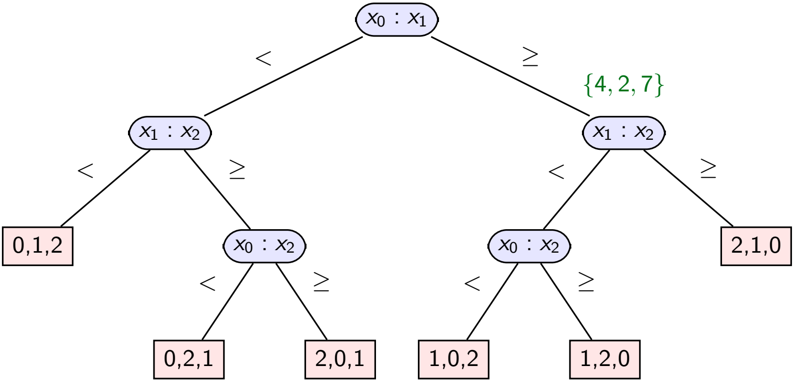
Example: $\{x_0{=}4, x_1{=}2, x_2{=}7\}$



Decision trees

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

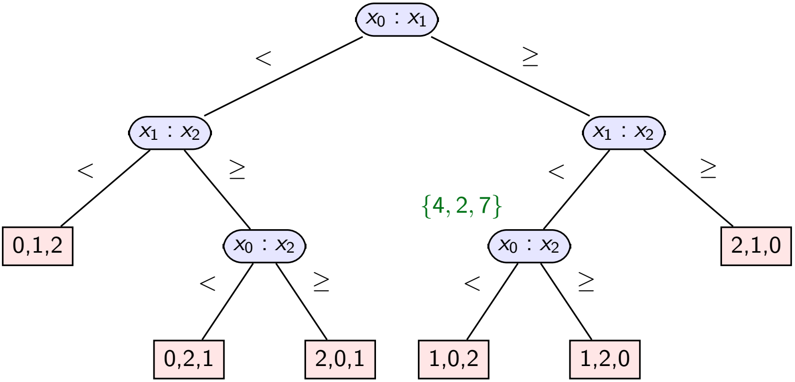
To sort $\{x_0, x_1, x_2\}$:



Decision trees

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

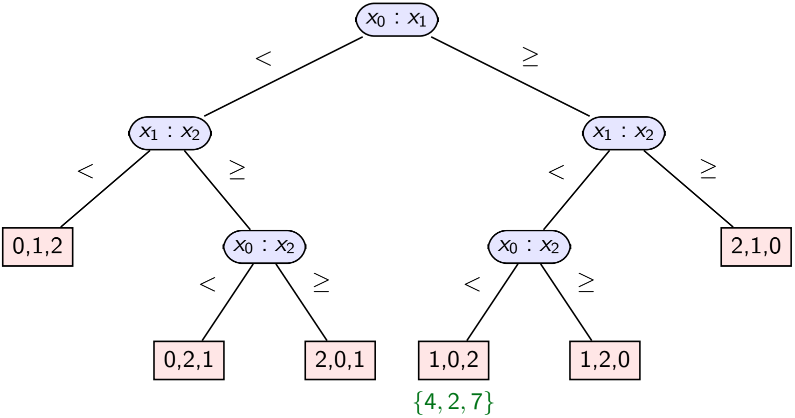
To sort $\{x_0, x_1, x_2\}$:



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

3 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

- 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.

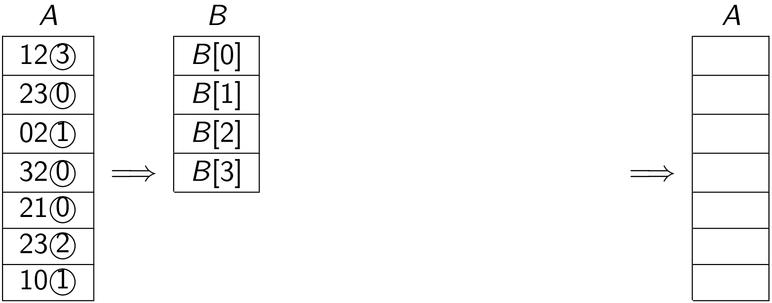
Example ($R = 4$):

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?

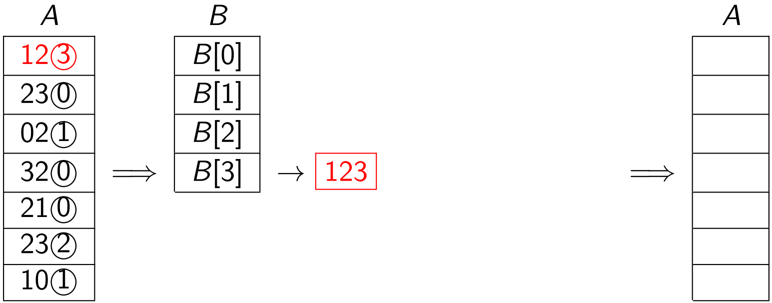
(Single-digit) Bucket Sort

Sort array A by last digit:



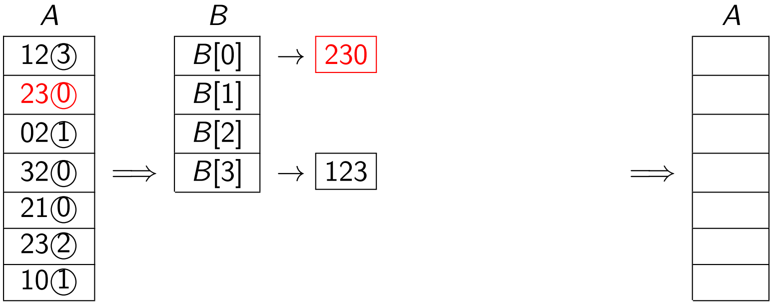
(Single-digit) Bucket Sort

Sort array A by last digit:



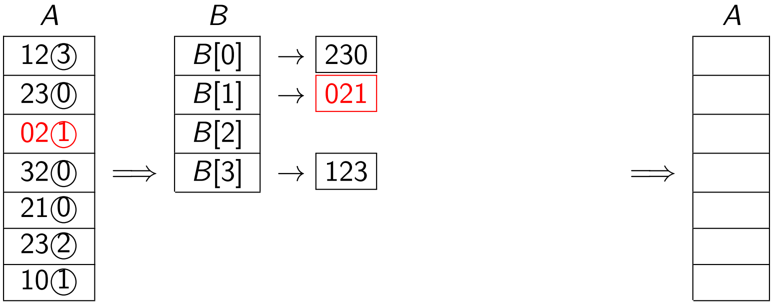
(Single-digit) Bucket Sort

Sort array A by last digit:



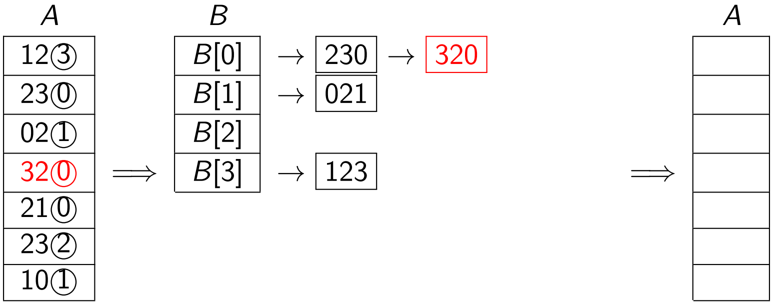
(Single-digit) Bucket Sort

Sort array A by last digit:



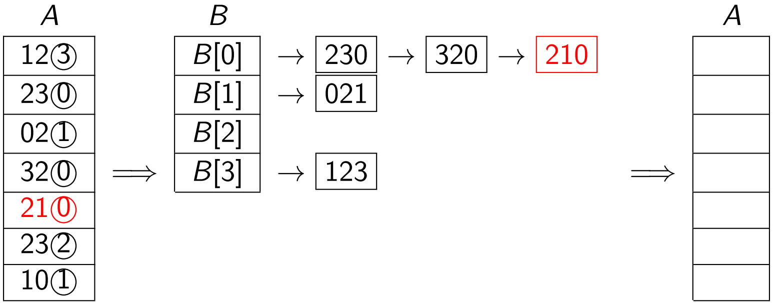
(Single-digit) Bucket Sort

Sort array A by last digit:



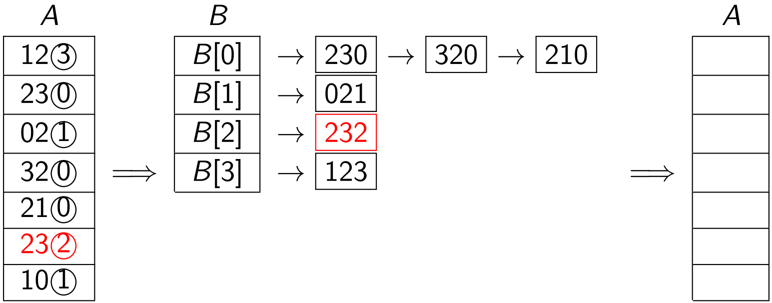
(Single-digit) Bucket Sort

Sort array A by last digit:



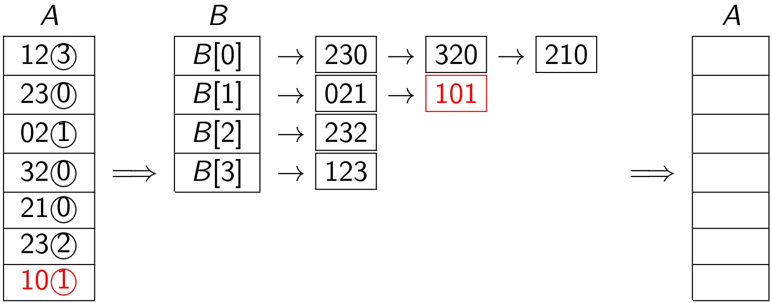
(Single-digit) Bucket Sort

Sort array A by last digit:



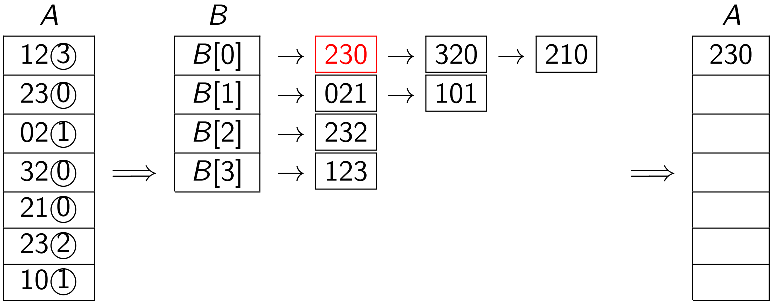
(Single-digit) Bucket Sort

Sort array A by last digit:



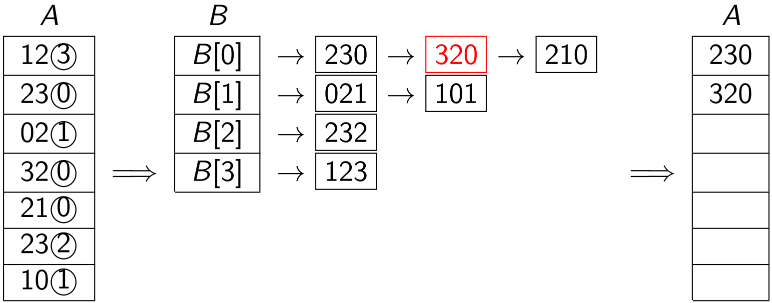
(Single-digit) Bucket Sort

Sort array A by last digit:



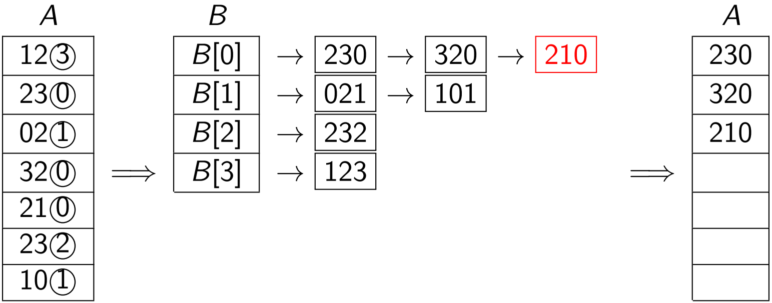
(Single-digit) Bucket Sort

Sort array A by last digit:



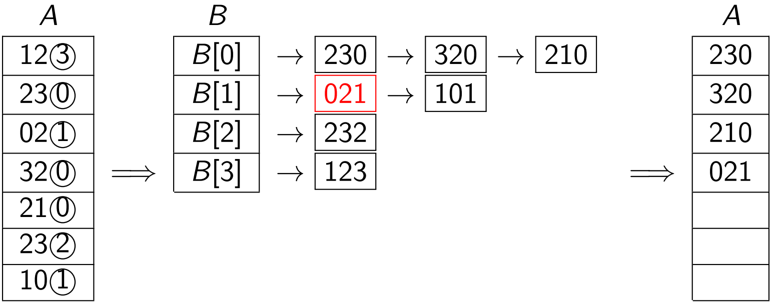
(Single-digit) Bucket Sort

Sort array A by last digit:



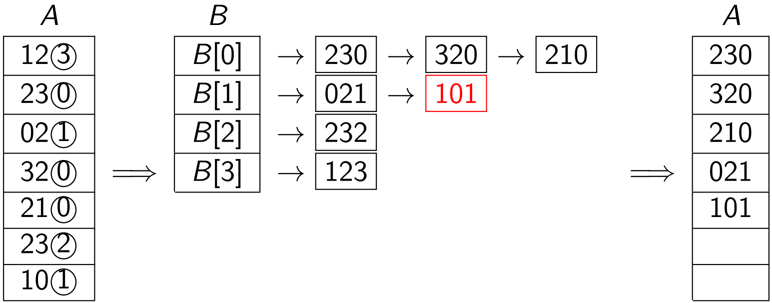
(Single-digit) Bucket Sort

Sort array A by last digit:



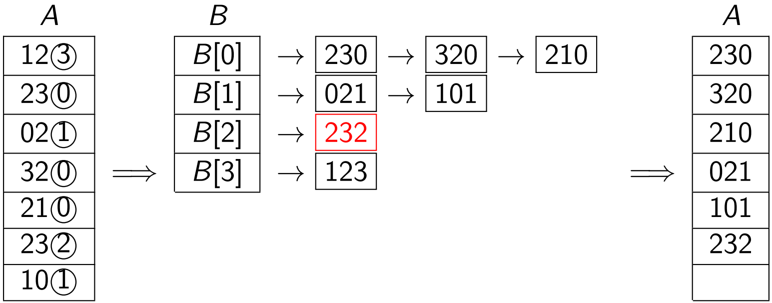
(Single-digit) Bucket Sort

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:

A
12③
23⓪
02①
32⓪
21⓪
23②
10①

$\Longrightarrow$

B			
$B[0]$	$\rightarrow$ 230	$\rightarrow$ 320	$\rightarrow$ 210
$B[1]$	$\rightarrow$ 021	$\rightarrow$ 101	
$B[2]$	$\rightarrow$ 232		
$B[3]$	$\rightarrow$ 123		

$\Longrightarrow$

A
230
320
210
021
101
232
123

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

- 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

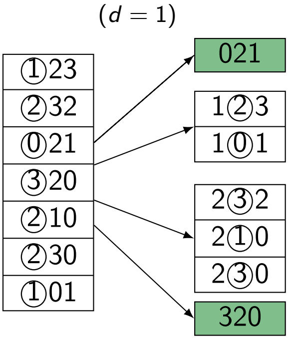
Sorts array of m-digit radix-R numbers recursively:
sort by leading digit, then each group by next digit, etc.

```
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
    if ℓ < r
        bucket-sort(A[ℓ..r], d)
        if there are digits left    // recurse in sub-arrays
            ℓ' ← ℓ
            while (ℓ' < r) do
                Let r' ≥ ℓ' be maximal s.t. A[ℓ'..r'] all have same dth digit
                MSD-Radix-sort(A, ℓ', r', d+1)
                ℓ' ← r' + 1
```

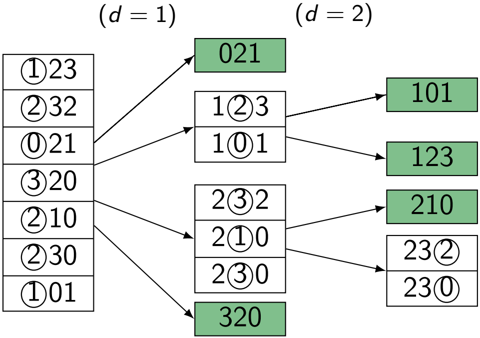
MSD-Radix-Sort Example

(1)23
(2)32
(0)21
(3)20
(2)10
(2)30
(1)01

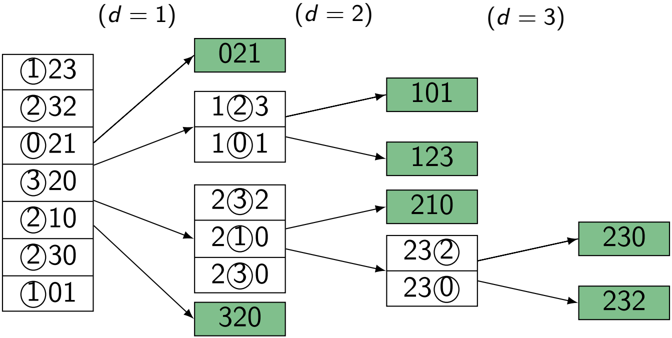
MSD-Radix-Sort Example



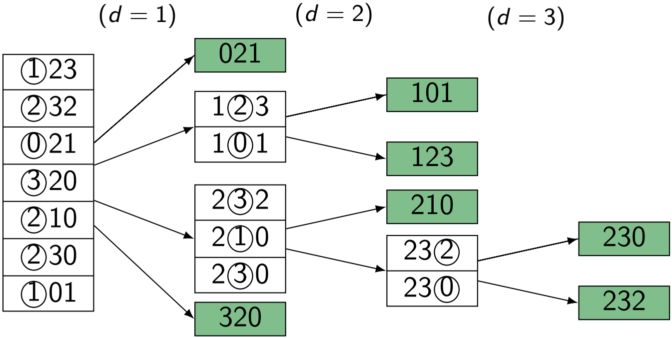
MSD-Radix-Sort Example



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

```
for d ← least significant to most significant digit do
        Bucket-sort(A, d)
```

12③		2③0		①01		021
23⓪		3②0		②10		101
02①	$(d=3)$ $\Longrightarrow$	2①0	$(d=2)$ $\Longrightarrow$	③20	$(d=1)$ $\Longrightarrow$	123
32⓪		0②1		⓪21		210
21⓪		1⓪1		①23		230
23②		2③2		②30		232
10①		1②3		②32		320

- 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.