CS 240E – Data Structures and Data Management (Enriched)

Module 3: Sorting, Average-case and Randomization

Therese Biedl

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025–01–16 11:42

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - Selection and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - 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:

$$T_{\mathcal{A}}^{avg}(n) = \sum_{\text{instance } I \text{ of size } n} T_{\mathcal{A}}(I) \cdot \big(\text{relative frequency of } I\big)$$

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:

$$T_{\mathcal{A}}^{avg}(n) = \sum_{\text{instance } I \text{ of size } n} T_{\mathcal{A}}(I) \cdot (\text{relative frequency of } I)$$

For this module:

- Assume that the set $\mathcal{I}_n$ of size-n instances is finite (or can be mapped to a finite set in a natural way)
- Assume that all instances occur equally frequently

Then we can use the following *simplified formula*

$$T^{avg}(n) = \frac{\sum_{I:\text{size}(I)=n} T(I)}{\#\text{instances of size } n} = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I)$$

To learn how to analyze this, we will do simpler examples first.

A simple (contrived) example

silly-test(π, n)
π: a permutation of $\{0, \ldots, n-1\}$, stored as an array
1. **if** $\pi[0] = 0$ **then for** $j = 1$ to n **do** print 'bad case'
2. **else** print 'good case'

$$T^{avg}(n) \qquad = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi) = \frac{1}{n!} \Big(\sum_{\substack{\pi \in \Pi_n \\ \text{in bad case}}} T(\pi) + \sum_{\substack{\pi \in \Pi_n \\ \text{in good case}}} T(\pi) \Big)$$

A simple (contrived) example

silly-test$(\pi,\ n)$
π: a permutation of $\{0,\ldots,n{-}1\}$, stored as an array
1. **if** $\pi[0]=0$ **then for** $j=1$ to n **do** print 'bad case'
2. **else** print 'good case'

$$T^{avg}(n) \qquad = \frac{1}{n!}\sum_{\pi\in\Pi_n} T(\pi) = \frac{1}{n!}\Big(\sum_{\substack{\pi\in\Pi_n \\ \text{in bad case}}} T(\pi) + \sum_{\substack{\pi\in\Pi_n \\ \text{in good case}}} T(\pi)\Big)$$

- $(n-1)!$ permutations have $\pi[0]=0 \Rightarrow$ run-time $c\cdot n$
- The remaining $n!-(n-1)!$ permutations have run-time c.

(for some constant $c>0$)

A simple (contrived) example

silly-test$(\pi,\ n)$
π: a permutation of $\{0,\dots,n-1\}$, stored as an array
1. **if** $\pi[0]=0$ **then for** $j=1$ to n **do** print 'bad case'
2. **else** print 'good case'

$$T^{avg}(n) \qquad = \frac{1}{n!}\sum_{\pi\in\Pi_n} T(\pi) = \frac{1}{n!}\Big(\sum_{\substack{\pi\in\Pi_n\\ \text{in bad case}}} T(\pi) + \sum_{\substack{\pi\in\Pi_n\\ \text{in good case}}} T(\pi)\Big)$$

- $(n-1)!$ permutations have $\pi[0]=0 \Rightarrow$ run-time $c\cdot n$
- The remaining $n!-(n-1)!$ permutations have run-time c.

(for some constant $c>0$)

$$\begin{aligned} T^{avg}(n) &= \frac{1}{n!}\Big(\#\{\pi\in\Pi_n \text{ in bad case}\}\cdot cn + \#\{\pi\in\Pi_n \text{ in good case}\}\cdot c\Big)\\ &= \frac{1}{n!}\Big((n-1)!\cdot cn + (n!-(n-1)!)\cdot c\Big) \le \frac{1}{n}cn + c = 2c \in O(1)\end{aligned}$$

A second (not-so-contrived) example

all-0-test$(w,\ n)$
`// test whether all entries of bitstring` $w[0..n-1]$ `are 0`
1. **if** $(n = 0)$ **return** `true`
2. **if** $(w[n-1] = 1)$ **return** `false`
3. *all-0-test*$(w, n-1)$

(In real life, you would write this non-recursive.)

Define $T(w) = \#$ bit-comparisons (i.e., line 2) on input w. This is asymptotically the same as the run-time.

Worst-case run-time: Always go into the recursion until $n = 0$.
$T(n) = 1 + T(n-1) = 1 + 1 + \cdots + T(0) = n \in \Theta(n)$.

A second (not-so-contrived) example

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

(In real life, you would write this non-recursive.)

Define $T(w) = \#$ bit-comparisons (i.e., line 2) on input w. This is asymptotically the same as the run-time.

Worst-case run-time: Always go into the recursion until $n = 0$.
$T(n) = 1 + T(n-1) = 1 + 1 + \cdots + T(0) = n \in \Theta(n)$.

Best-case run-time: Return immediately. $T(n) = 1 \in \Theta(1)$.

Average-case run-time?

Average-case run-time of *all-0-test*

Recall $T^{\text{avg}}(n) = \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w)$. $\quad$ ($\mathcal{B}_n = \{$bitstrings of length $n\}$)

Recursive formula for one non-empty bitstring w:

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

Average-case run-time of *all-0-test*

Recall $T^{\text{avg}}(n) = \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w)$. $\qquad$ ($\mathcal{B}_n = \{$bitstrings of length $n\}$)

Recursive formula for one non-empty bitstring w:

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

Natural guess for the recursive formula for $T^{avg}(n)$:

$$T^{avg}(n) \quad = \quad \underbrace{\tfrac{1}{2}}_{\substack{\text{half have} \\ w[n-1]=1}} \cdot 1 + \underbrace{\tfrac{1}{2}}_{\substack{\text{half have} \\ w[n-1]=0}} (1 + T^{???}(n{-}1))$$

Average-case run-time of *all-0-test*

Recall $T^{\text{avg}}(n) = \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w)$. $\quad$ ($\mathcal{B}_n = \{$bitstrings of length $n\}$)

Recursive formula for one non-empty bitstring w:

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

Natural guess for the recursive formula for $T^{avg}(n)$:

$$T^{avg}(n) = \underbrace{\tfrac{1}{2}}_{\substack{\text{half have} \\ w[n-1]=1}} \cdot 1 + \underbrace{\tfrac{1}{2}}_{\substack{\text{half have} \\ w[n-1]=0}} (1 + T^{???}(n{-}1))$$

- This holds with $\leq$ (but is useless) if '???' is 'worst'.
- This is *not obvious* if '???' is 'avg'.

Average-case run-time of *all-0-test*

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

$$=$$

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

This recursion resolves to $T^{avg}(n) \in O(1)$.

Average-case analysis and recursions

Why can't we always write 'avg' for '???' in $T^{avg}(n) = 1 + \frac{1}{2}T^{???}(n-1)$?

Consider the following (contrived) example:

silly-all-0-test$(w,\ n)$
w: array of size at least n that stores bits
if $(n = 0)$ **then return** `true`
if $(w[n-1] = 1)$ **then return** `false`
if $(n > 1)$ **then** $w[n-2] \leftarrow 0$ `// this is the only change`
silly-all-0-test$(w, n-1)$

Average-case analysis and recursions

Why can't we always write 'avg' for '???' in $T^{avg}(n) = 1 + \frac{1}{2}T^{???}(n-1)$?

Consider the following (contrived) example:

```
silly-all-0-test(w, n)
w: array of size at least n that stores bits
    if (n = 0) then return true
    if (w[n−1] = 1) then return false
    if (n > 1) then w[n−2] ← 0      // this is the only change
    silly-all-0-test(w, n−1)
```

- Only one more line of code in each recursion, so same formula applies.
- But observe that now $T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ n & \text{if } w[n-1] = 0 \end{cases}$.

Average-case analysis and recursions

Why can't we always write 'avg' for '???' in $T^{avg}(n) = 1 + \frac{1}{2}T^{???}(n-1)$?

Consider the following (contrived) example:

```
silly-all-0-test(w, n)
w: array of size at least n that stores bits
1.  if (n = 0) then return true
2.  if (w[n−1] = 1) then return false
3.  if (n > 1) then w[n−2] ← 0      // this is the only change
4.  silly-all-0-test(w, n−1)
```

- Only one more line of code in each recursion, so same formula applies.
- But observe that now $T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ n & \text{if } w[n-1] = 0 \end{cases}$.
- So $T^{avg}(n) = 1 + \frac{n}{2} \in \Theta(n)$. The "obvious" recursion did not hold.

Average-case analysis is highly non-trivial for recursive algorithms.

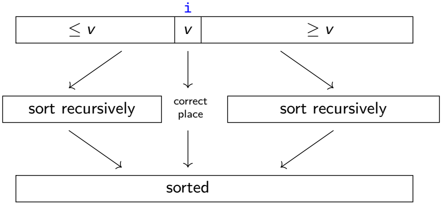
Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Review and Outlook

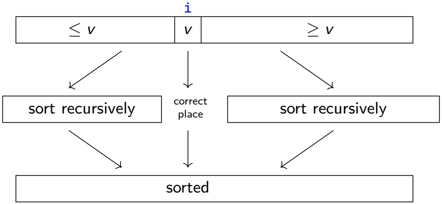
quick-sort(A) `// array of size` n

1. **if** $n \leq 1$ **then return**
2. $i \leftarrow$ *partition*$(A,$ *choose-pivot*$(A))$
3. *quick-sort*$(A[0, 1, \ldots, i{-}1])$
4. *quick-sort*$(A[i{+}1, \ldots, n{-}1])$



Review and Outlook

quick-sort(A) `// array of size` n
1. **if** $n \leq 1$ **then return**
2. $i \leftarrow$ *partition*$(A,$ *choose-pivot*$(A))$
3. *quick-sort*$(A[0, 1, \ldots, i{-}1])$
4. *quick-sort*$(A[i{+}1, \ldots, n{-}1])$



- *choose-pivot*: detemines **pivot-value** v.
  For now, we simply take the rightmost item in A.
- *partition*: achieves top picture, returns **pivot-rank** i
  This takes n **key-comparisons** (compare two items of A).

Review and Outlook

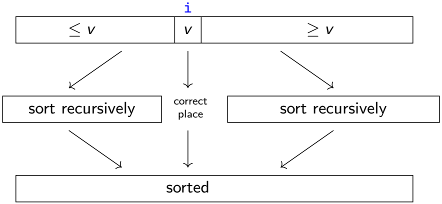
quick-sort(A) `// array of size` n

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

$i \leftarrow$ *partition*$(A,$ *choose-pivot*$(A))$

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

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

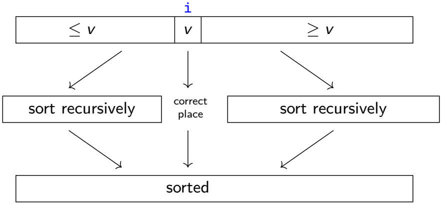


- *choose-pivot*: detemines **pivot-value** v.
  For now, we simply take the rightmost item in A.
- *partition*: achieves top picture, returns **pivot-rank** i
  This takes n **key-comparisons** (compare two items of A).

- *quick-sort* has $\Theta(n^2)$ worst-case run-time.
- *quick-sort* has $\Theta(n \log n)$ best-case run-time.

Review and Outlook

```
quick-sort(A) // array of size n
if n ≤ 1 then return
i ← partition(A, choose-pivot(A))
quick-sort(A[0, 1, ..., i−1])
quick-sort(A[i+1, ..., n−1])
```



- *choose-pivot*: detemines **pivot-value** v.
  For now, we simply take the rightmost item in A.
- *partition*: achieves top picture, returns **pivot-rank** i
  This takes n **key-comparisons** (compare two items of A).

- *quick-sort* has $\Theta(n^2)$ worst-case run-time.
- *quick-sort* has $\Theta(n \log n)$ best-case run-time.

- What is the *average-case* run-time?
- We will study a related problem (with simpler algorithm) first.

The Selection Problem

We saw Selection: 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.

(We also call this the element of **rank** k.)

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.

Selection can be done with heaps in time $\Theta(n + k \log n)$.

Special case: MedianFinding = Selection with $k = \lfloor \frac{n}{2} \rfloor$. With previous approaches, this takes time $\Theta(n \log n)$, no better than sorting.

The Selection Problem

We saw SELECTION: 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.

(We also call this the element of **rank** k.)

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.

SELECTION can be done with heaps in time $\Theta(n + k \log n)$.

Special case: MEDIANFINDING = SELECTION with $k = \lfloor \frac{n}{2} \rfloor$. With previous approaches, this takes time $\Theta(n \log n)$, no better than sorting.

Question: Can we do selection in linear time?
Yes! We will develop algorithm *quick-select* below.

quick-select Algorithm

SELECTION: Want item m such that (after rearranging A) we have

$\leq m$	m	$\geq m$
	k	

quick-select(A, k)
A: array of size n, k: integer s.t. $0 \leq k < n$

```
p ← choose-pivot(A)
i ← partition(A, p)
if i = k then return A[i]
else if i > k then return quick-select(A[0 ... i−1], k)
else if i < k then return quick-select(A[i+1 ... n−1], k − (i+1))
```

Idea: After partition have

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

Where is m if $k = i$? If $k < i$? If $k > i$?

Analysis of *quick-select*

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 run-time:

- Sub-array always gets smaller, so $\leq n$ recursions.
  Each takes $\leq n$ comparisons $\Rightarrow O(n^2)$ time.
- This is tight: If pivot-value is always the maximum and $k = 0$
  $T^{\text{worst}}(n,0) \geq n + (n-1) + (n-2) + \cdots + 1 \in \Omega(n^2)$

Analysis of *quick-select*

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 run-time:

- Sub-array always gets smaller, so $\leq n$ recursions.
  Each takes $\leq n$ comparisons $\Rightarrow O(n^2)$ time.
- This is tight: If pivot-value is always the maximum and $k = 0$
  $T^{\text{worst}}(n,0) \geq n + (n{-}1) + (n{-}2) + \cdots + 1 \in \Omega(n^2)$

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

Analysis of *quick-select*

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 run-time:

- Sub-array always gets smaller, so $\leq n$ recursions.
  Each takes $\leq n$ comparisons $\Rightarrow O(n^2)$ time.
- This is tight: If pivot-value is always the maximum and $k=0$
  $T^{\text{worst}}(n,0) \geq n + (n-1) + (n-2) + \cdots + 1 \in \Omega(n^2)$

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

Average case analysis? Doing this directly would be *very* complicated.
Instead we will do it via a detour into a randomized version.

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Randomized algorithm example (very contrived)

randomized-all-0-test(w, n)
w: array of size at least n that stores bits
1. **if** $n = 0$ **return** `true`
2. **if** (*random*(2)=0) **then**
   $w[n-1] = 1 - w[n-1]$ `// this is the only change`
3. **if** $w[n-1] = 1$ **return** `false`
4. *randomized-all-0-test*($w, n-1$)

This is *all-0-test*, except that we flip last bit based on a coin toss.

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

Randomized algorithm example (very contrived)

```
randomized-all-0-test(w, n)
w: array of size at least n that stores bits
1.  if n = 0 return true
2.  if (random(2)=0) then
            w[n−1] = 1 − w[n−1]      // this is the only change
3.  if w[n−1] = 1 return false
4.  randomized-all-0-test(w, n−1)
```

This is *all-0-test*, except that we flip last bit based on a coin toss.

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

In each recursion, we use the outcome $x \in \{0, 1\}$ of one coin toss.
We return without recursing if $x = w[n-1]$ (this has probability $\frac{1}{2}$).

Expected run-time of *randomized-all-0-test*

Define $T(w,R) := \#$ bit-comparisons used on input w if the random outcomes are R. (This is proportional to the run-time.)

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\Pr(R) = \Pr(x) \cdot \Pr(R')$ (random choices are independent).

Expected run-time of *randomized-all-0-test*

Define $T(w, R) := \#$ bit-comparisons used on input w if the random outcomes are R. (This is proportional to the run-time.)

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\Pr(R) = \Pr(x) \cdot \Pr(R')$ (random choices are independent).

- Recursive formula for one instance:

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

Expected run-time of *randomized-all-0-test*

Define $T(w, R) :=$ # bit-comparisons used on input w if the random outcomes are R. (This is proportional to the run-time.)

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\Pr(R) = \Pr(x) \cdot \Pr(R')$ (random choices are independent).

- Recursive formula for one instance:

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

- Natural guess for the recursive formula for $T^{exp}(n)$:

$$T^{exp}(n) = \underbrace{\tfrac{1}{2}}_{\Pr(x=w[n-1])} \cdot 1 + \underbrace{\tfrac{1}{2}}_{\Pr(x \neq w[n-1])} (1 + T^{exp}(n-1)) = 1 + \tfrac{1}{2} T^{exp}(n-1)$$

Expected run-time of *randomized-all-0-test*

In contrast to average-case analysis, the natural guess usually is correct for the expected run-time.

Proof for *randomized-all-0-test*:

$$T^{exp}(w) \;=\; \sum_{R} \Pr(R)\, T(w,R) =$$

Expected run-time of *randomized-all-0-test*

In contrast to average-case analysis, the natural guess usually is correct for the expected run-time.

Proof for *randomized-all-0-test*:

$$\begin{aligned} T^{exp}(w) &= \sum_R \Pr(R)\, T(w,R) = \sum_x \sum_{R'} \Pr(x)\Pr(R')\, T(w,\langle x, R'\rangle) \\ &= \end{aligned}$$

Expected run-time of *randomized-all-0-test*

In contrast to average-case analysis, the natural guess usually is correct for the expected run-time.

Proof for *randomized-all-0-test*:

$$\begin{aligned} T^{exp}(w) &= \sum_R \Pr(R)\, T(w,R) = \sum_x \sum_{R'} \Pr(x) \Pr(R')\, T(w, \langle x, R' \rangle) \\ &= \end{aligned}$$

Therefore $T^{exp}(n) = \max_{w \in \mathcal{B}_n} T^{exp}(w) \leq 1 + \frac{1}{2} T^{exp}(n-1)$

Expected run-time of *randomized-all-0-test*

- We had $T^{exp}_{rand\text{-}all\text{-}0\text{-}test}(n) \leq 1 + \frac{1}{2} T^{exp}_{rand\text{-}all\text{-}0\text{-}test}(n{-}1)$
- We earlier had $T^{avg}_{all\text{-}0\text{-}test}(n) \leq 1 + \frac{1}{2} T^{avg}_{all\text{-}0\text{-}test}(n{-}1)$
- Same recursion $\Rightarrow$ same upper bound $\Rightarrow$ $T^{exp}_{rand\text{-}all\text{-}0\text{-}test}(n) \in O(1)$.

Expected run-time of *randomized-all-0-test*

- We had $T^{exp}_{\textit{rand-all-0-test}}(n) \le 1 + \frac{1}{2} T^{exp}_{\textit{rand-all-0-test}}(n-1)$
- We earlier had $T^{avg}_{\textit{all-0-test}}(n) \le 1 + \frac{1}{2} T^{avg}_{\textit{all-0-test}}(n-1)$
- Same recursion $\Rightarrow$ same upper bound $\Rightarrow$ $T^{exp}_{\textit{rand-all-0-test}}(n) \in O(1)$.

- Recall: *randomized-all-0-test* was very similar to *all-0-test*
  (The only different was a random bitflip.)
- Is it a coincidence that the two recursive formulas are the same? Or does the expected time of a randomized version always have something to do with the average-case time?

Expected run-time of *randomized-all-0-test*

- We had $T^{exp}_{\textit{rand-all-0-test}}(n) \leq 1 + \frac{1}{2} T^{exp}_{\textit{rand-all-0-test}}(n-1)$
- We earlier had $T^{avg}_{\textit{all-0-test}}(n) \leq 1 + \frac{1}{2} T^{avg}_{\textit{all-0-test}}(n-1)$
- Same recursion $\Rightarrow$ same upper bound $\Rightarrow$ $T^{exp}_{\textit{rand-all-0-test}}(n) \in O(1)$.

- Recall: *randomized-all-0-test* was very similar to *all-0-test*
  (The only different was a random bitflip.)
- Is it a coincidence that the two recursive formulas are the same? Or does the expected time of a randomized version always have something to do with the average-case time?

- Not in general! (It depends how we randomize.)
- Yes if the randomization is a *shuffle* (choose instance randomly).

Avg-case run-time via expected run-time

Consider the following randomization of a deterministic algorithm $\mathcal{A}$.

shuffle-$\mathcal{A}$ (n)
1. Among all instances $\mathcal{I}_n$ of size n for $\mathcal{A}$, choose I randomly
2. $\mathcal{A}(I)$

(*shuffle-$\mathcal{A}$* usually does not solve what $\mathcal{A}$ solves)

Avg-case run-time via expected run-time

Consider the following randomization of a deterministic algorithm $\mathcal{A}$.

shuffle-$\mathcal{A}(n)$
1. Among all instances $\mathcal{I}_n$ of size n for $\mathcal{A}$, choose I randomly
2. $\mathcal{A}(I)$

(*shuffle-*$\mathcal{A}$ usually does not solve what $\mathcal{A}$ solves)

- If we do not count the time for line 1:

$$T^{avg}_{\mathcal{A}}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I) = \sum_{I \in \mathcal{I}_n} Pr(I \text{ chosen}) \cdot T(I) = T^{exp}_{\textit{shuffle-}\mathcal{A}}(n)$$

Avg-case run-time via expected run-time

Consider the following randomization of a deterministic algorithm $\mathcal{A}$.

shuffle-$\mathcal{A}$ (n)
1. Among all instances $\mathcal{I}_n$ of size n for $\mathcal{A}$, choose I randomly
2. $\mathcal{A}(I)$

(*shuffle-$\mathcal{A}$* usually does not solve what $\mathcal{A}$ solves)

- If we do not count the time for line 1:

$$T^{avg}_{\mathcal{A}}(n) = \frac{1}{|\mathcal{I}_n|} \sum_{I \in \mathcal{I}_n} T(I) = \sum_{I \in \mathcal{I}_n} Pr(I \text{ chosen}) \cdot T(I) = T^{exp}_{\textit{shuffle-}\mathcal{A}}(n)$$

- So the average-case run-time of $\mathcal{A}$ is the same as this **run-time of $\mathcal{A}$ on randomly chosen input**.
- This gives us a different way to compute $T^{avg}_{\mathcal{A}}(n)$.

Avg-case run-time via expected run-time

Example: *all-0-test* (rephrased with for-loops):

shuffle-all-0-test(n)
1. **for** $(i = n{-}1; i \geq 0; i{-}{-})$ **do**
2. $\quad w[i] \leftarrow$ *random*(2)
3. **for** $(i = n{-}1; i \geq 0; i{-}{-})$ **do**
4. $\quad$ **if** $(w[i] = 1)$ **return** `false`
5. **return** `true`

randomized-all-0-test(w, n)
1. **for** $(i = n{-}1; i \geq 0; i{-}{-})$ **do**
2. $\quad$ **if** $($*random*$(2){=}0)$ **then**
   $\qquad w[i] = 1 - w[i]$
3. $\quad$ **if** $(w[i] = 1)$ **return** `false`
4. **return** `true`

Avg-case run-time via expected run-time

Example: *all-0-test* (rephrased with for-loops):

shuffle-all-0-test(n)
1. **for** ($i = n{-}1; i \geq 0; i$--) **do**
2. $\quad w[i] \leftarrow$ *random*(2)
3. **for** ($i = n{-}1; i \geq 0; i$--) **do**
4. $\quad$ **if** ($w[i] = 1$) **return** `false`
5. **return** `true`

randomized-all-0-test(w, n)
1. **for** ($i = n{-}1; i \geq 0; i$--) **do**
2. $\quad$ **if** (*random*(2)$=0$) **then**
   $\quad\quad w[i] = 1 - w[i]$
3. $\quad$ **if** ($w[i] = 1$) **return** `false`
4. **return** `true`

- These algorithms are not quite the same.
  - Randomization outside respectively inside the for-loop.
- But this does not matter for the expected number of bit-comparisons.
  - Either way, at time of comparison the bit is 1 with probability $\frac{1}{2}$.

Avg-case run-time via expected run-time

Example: *all-0-test* (rephrased with for-loops):

```
shuffle-all-0-test(n)
 1. for (i = n−1; i ≥ 0; i−−) do
 2.       w[i] ← random(2)
 3. for (i = n−1; i ≥ 0; i−−) do
 4.       if (w[i] = 1) return false
 5. return true
```

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

- These algorithms are not quite the same.
  - Randomization outside respectively inside the for-loop.
- But this does not matter for the expected number of bit-comparisons.
  - Either way, at time of comparison the bit is 1 with probability $\frac{1}{2}$.
- So $T^{avg}_{\text{all-0-test}}(n) = T^{exp}_{\text{shuffle-all-0-test}}(n) = T^{exp}_{\text{rand-all-0-test}}(n) \in O(1)$

  can be deduced without analyzing $T^{avg}_{\text{all-0-test}}(n)$ directly.

Summary: Average-case run-time vs. expected run-time

So: are average-case run-time and expected run-time the same?

Summary: Average-case run-time vs. expected run-time

So: are average-case run-time and expected run-time the same?

No!

average-case run-time	expected run-time
$\frac{1}{\lvert\mathcal{I}_n\rvert}\sum_{I\in\mathcal{I}_n} T(I)$	$\max_{I\in\mathcal{I}_n} \sum_{\text{outcomes } R} \Pr(R)\cdot T(I,R)$
average over instances	weighted average over random outcomes
(usually) applied to a deterministic algorithm	applied only to a randomized algorithm

Summary: Average-case run-time vs. expected run-time

So: are average-case run-time and expected run-time the same?

No!

average-case run-time	expected run-time
$\frac{1}{\lvert\mathcal{I}_n\rvert}\sum_{I\in\mathcal{I}_n} T(I)$	$\max_{I\in\mathcal{I}_n} \sum_{\text{outcomes } R} \Pr(R)\cdot T(I,R)$
average over instances	weighted average over random outcomes
(usually) applied to a deterministic algorithm	applied only to a randomized algorithm

There is a relationship *only* if the randomization effectively achieves "choose the input instance randomly".

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Average-case analysis of quick-select

Recall *quick-select*:

> *quick-select*(A, k)
> 1. $i \leftarrow$ *partition*$(A, n{-}1)$
> 2. **if** $i = k$ **then return** $A[i]$
> 3. **else if** $i > k$ **then** *quick-select*$(A[0 \ldots i{-}1], k)$
> 4. **else if** $i < k$ **then** *quick-select*$(A[i{+}1 \ldots n{-}1], k - (i{+}1))$

For analyzing the average-case run-time, we make two **assumptions**:

- All input-items are distinct.
  - This can be forced using tie-breakers.
- All possible orders of the input-items occur equally often.
  All possible rank-parameters occur equally often.
  - This is not completely realistic (mostly-sorted orders or rank-parameter $\lceil n/2 \rceil$ are more common).
  - But we cannot do average-case analysis without it.

Randomizing quick-select: Shuffling

Goal: Create a randomized version of *quick-select*.

- This will give a proof of the avg-case run-time of *quick-select*.
- This will be a better algorithm in practice.

Randomizing quick-select: Shuffling

Goal: Create a randomized version of *quick-select*.

- This will give a proof of the avg-case run-time of *quick-select*.
- This will be a better algorithm in practice.

First idea: Shuffle the input, then do *quick-select*.

shuffle-quick-select(A, k)
1. **for** $(j \leftarrow 1$ to $n-1)$ **do** *swap*$(\,A[j], A[\mathit{random}(j+1)]\,)$ `// shuffle`
2. *quick-select*(A, k)

- Shuffling (permuting) the input-array is (by assumption) equivalent to randomly choosing an input instance.
- So we know $T^{avg}_{\text{quick-select}}(n) = T^{exp}_{\text{shuffle-quick-select}}(n)$

(Recall: $T(\cdot)$ counts key-comparisons, so shuffling is free.)

Randomizing quick-select: Random Pivot

Second idea: Do the shuffling inside the recursion.
(Equivalently: Randomly choose which value is used for the pivot.)

randomized-quick-select(A, k)
1. swap $A[n{-}1]$ with $A[random(n)]$
2. $i \leftarrow partition(A, n{-}1)$
3. **if** $i = k$ **then return** $A[i]$
4. **else if** $i > k$ **then**
5. **return** *randomized-quick-select*$(A[0 \ldots i{-}1], k)$
6. **else if** $i < k$ **then**
7. **return** *randomized-quick-select*$(A[i{+}1 \ldots n{-}1], k - (i{+}1))$

Randomizing quick-select: Random Pivot

Second idea: Do the shuffling inside the recursion.
(Equivalently: Randomly choose which value is used for the pivot.)

randomized-quick-select(A, k)
```
swap A[n−1] with A[random(n)]
i ← partition(A, n−1)
if i = k then return A[i]
else if i > k then
        return randomized-quick-select(A[0 . . . i−1], k)
else if i < k then
        return randomized-quick-select(A[i+1 . . . n−1], k − (i+1))
```

- $T^{exp}_{\textit{rand.-quick-select}}(n) = T^{exp}_{\textit{shuffle-quick-select}}(n)$.

(This is not completely obvious, but believable. No proof.)

Expected run-time of *randomized-quick-select*

Let $T(A, k, R) = \#$ key-comparisons of *randomized-quick-select* on input $\langle A, k\rangle$ if the random outcomes are R. (This is proportional to the run-time.)

- Write random outcomes R as $R = \langle i, R'\rangle$ (where 'i' stands for 'the first random number was such that the pivot-rank is i')
- Observe: $\Pr(\text{pivot-rank is } i) = \frac{1}{n}$
- We recurse in an array of size i or $n-i-1$ (or not at all)

Expected run-time of *randomized-quick-select*

Let $T(A, k, R) = \#$ key-comparisons of *randomized-quick-select* on input $\langle A, k\rangle$ if the random outcomes are R. (This is proportional to the run-time.)

- Write random outcomes R as $R = \langle i, R'\rangle$ (where 'i' stands for 'the first random number was such that the pivot-rank is i')
- Observe: $\Pr(\text{pivot-rank is } i) = \frac{1}{n}$
- We recurse in an array of size i or $n{-}i{-}1$ (or not at all)
- Recursive formula for one instance (and fixed $R = \langle i, R'\rangle$):

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

Analysis of *randomized-quick-select*

As for *rand-all-0-test*: over all R, the recursions can use T^{exp}(array-size).

$$T^{exp}(A,k) \left(\begin{aligned} &= \sum_{R} P(R)\cdot T(\langle A,k\rangle, R) = \sum_{i=0}^{n-1}\sum_{R'} P(i)\cdot P(R')\cdot T(\langle A,k\rangle, \langle i, R'\rangle) \\ &= \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} P(R')\Big(n + T(\langle A[i{+}1..n{-}1], k{-}i{-}1\rangle, R')\Big) \\ &\qquad + \underbrace{\frac{1}{n}\cdot n}_{i=k} \quad + \frac{1}{n}\sum_{i=k+1}^{n-1}\sum_{R'} P(R')\Big(n + T(\langle A[0..i{-}1, k\rangle, R')\Big) \\ &= n + \frac{1}{n}\sum_{i=0}^{k-1}\sum_{R'} P(R')T(\langle A[i{+}1..n{-}1], k{-}i{-}1\rangle, R') \\ &\qquad + \frac{1}{n}\sum_{i=k+1}^{n-1}\sum_{R'} P(R')T(\langle A[0..i{-}1, k\rangle, R') \\ &= n + \frac{1}{n}\sum_{i=0}^{k-1}\underbrace{T^{\exp}(\langle A[i{+}1..n{-}1], k{-}i{-}1\rangle)}_{\le T^{\exp}(n-i-1)} + \frac{1}{n}\sum_{i=k+1}^{n-1}\underbrace{T^{\exp}(\langle A[0..i{-}1], k\rangle)}_{\le T^{\exp}(i)} \end{aligned} \right) \text{tedious but straightforward}$$

$$\le n + \frac{1}{n}\sum_{i=0}^{n-1}\max\{T^{exp}(i), T^{exp}(n{-}i{-}1)\} \quad \textit{independent} \text{ of } A, k$$

Analysis of *randomized-quick-select*

In summary, the expected run-time of *randomized-quick-select* satisfies:

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

Analysis of *randomized-quick-select*

In summary, the expected run-time of *randomized-quick-select* satisfies:

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

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

Summary of SELECTION

- *randomized-quick-select* has expected run-time $\Theta(n)$.
  - The run-time bound is tight since *partition* takes $\Omega(n)$ time
  - If we're unlucky in the random numbers then the run-time is still $\Omega(n^2)$
- So the expected run-time of *shuffle-quick-select* is $\Theta(n)$.
- So the run-time of *quick-select* on randomly chosen input is $\Theta(n)$.
- So the average-case run-time of *quick-select* is $\Theta(n)$.

Summary of SELECTION

- *randomized-quick-select* has expected run-time $\Theta(n)$.
  - The run-time bound is tight since *partition* takes $\Omega(n)$ time
  - If we're unlucky in the random numbers then the run-time is still $\Omega(n^2)$
- So the expected run-time of *shuffle-quick-select* is $\Theta(n)$.
- So the run-time of *quick-select* on randomly chosen input is $\Theta(n)$.
- So the average-case run-time of *quick-select* is $\Theta(n)$.

- *randomized-quick-select* is generally the fastest solution to SELECTION.

- There exists a variation that solves SELECTION with worst-case run-time $\Theta(n)$, but it uses double recursion and is slower in practice. ($\rightarrow$ *cs*341, maybe)

Randomizing quick-sort

randomized-quick-sort(A)
1. **if** $n \leq 1$ **then return**
2. $p \leftarrow$ *random*(n)
3. $i \leftarrow$ *partition*(A, p)
4. *randomized-quick-sort*$(A[0, 1, \ldots, i{-}1])$
5. *randomized-quick-sort*$(A[i{+}1, \ldots, n{-}1])$

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

Randomizing quick-sort

randomized-quick-sort(A)

```
if n ≤ 1 then return
p ← random(n)
i ← partition(A, p)
randomized-quick-sort(A[0, 1, ..., i−1])
randomized-quick-sort(A[i+1, ..., n−1])
```

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

Assume the random output was such that the pivot-rank is i:

- We use n comparisons in *partition*.
- We recurse in two arrays, of size i and $n-i-1$

Randomizing quick-sort

randomized-quick-sort(A)
1. **if** $n \leq 1$ **then return**
2. $p \leftarrow$ *random*(n)
3. $i \leftarrow$ *partition*(A, p)
4. *randomized-quick-sort*$(A[0, 1, \ldots, i{-}1])$
5. *randomized-quick-sort*$(A[i{+}1, \ldots, n{-}1])$

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

Assume the random output was such that the pivot-rank is i:

- We use n comparisons in *partition*.
- We recurse in two arrays, of size i and $n{-}i{-}1$

This implies

$$T^{exp}(n) = \underbrace{\ldots \; = \; \ldots \; \leq \; \ldots}_{\text{long but straightforward}} = n + \frac{1}{n}\sum_{i=0}^{n-1}\Big(T^{exp}(i) + T^{exp}(n{-}i{-}1)\Big)$$

Expected run-time of *randomized-quick-sort*

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

(since $T(0) = 0$)

Claim: $T^{exp}(n) \in O(n \log n)$.

Proof:

Expected run-time of *randomized-quick-sort*

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

(since $T(0) = 0$)

Claim: $T^{exp}(n) \in O(n \log n)$.

Proof:

Summary of *quick-sort*

- *randomized-quick-sort* has expected run-time $\Theta(n \log n)$.
  - The run-time bound is tight since the best-case run-time is $\Omega(n \log n)$
  - If we're unlucky in the random numbers then the run-time is still $\Omega(n^2)$
- Can show: This has the same expected run-time as *quick-select* on randomly chosen input (no details)
- So the average-case run-time of *quick-sort* is $\Theta(n \log n)$.

- The auxiliary space is not good ($\Theta(n)$) but can be improved ($\rightsquigarrow$ later)

Summary of *quick-sort*

- *randomized-quick-sort* has expected run-time $\Theta(n \log n)$.
  - The run-time bound is tight since the best-case run-time is $\Omega(n \log n)$
  - If we're unlucky in the random numbers then the run-time is still $\Omega(n^2)$
- Can show: This has the same expected run-time as *quick-select* on randomly chosen input (no details)
- So the average-case run-time of *quick-sort* is $\Theta(n \log n)$.
- The auxiliary space is not good ($\Theta(n)$) but can be improved ($\rightsquigarrow$ later)
- There are numerous other tricks to improve *randomized-quick-select*
  - We will see some below.
- With these, this is in practice the fastest solution to SORTING (but *not* in theory).

Outline

3 Sorting, Average-case and Randomization

- Analyzing average-case run-time
- SELECTION and *quick-select*
- Randomized Algorithms
- *quick-select* revisited
- Tips and Tricks for *quick-sort*
- Lower Bound for Comparison-Based Sorting
- Non-Comparison-Based Sorting

quick-sort with tricks

```
randomized-quick-sort-improved(A, n)
Initialize a stack S of index-pairs with { (0, n−1) }
while S is not empty
    (ℓ, r) ← S.pop()                          // avoid recursions
    while (r−ℓ+1 > 10) do                     // stop recursions early
        p ← ℓ + random(ℓ−r+1)
        i ← Hoare-partition(A, ℓ, r, p)       // use better routine
        if (i−ℓ > r−i) do                     // reduce aux. space
            S.push( (ℓ, i−1) )
            ℓ ← i+1                           // remove tail-recursion
        else
            S.push( (i+1, r) )
            r ← i−1
insertion-sort(A)
```

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's Partition Routine

- *partition* is very easy to implement with lists or streams (exercise). This uses $O(n)$ auxiliary space and is rather slow.
- More challenging: partition **in place** (with $O(1)$ auxiliary space).
- **Idea**: Keep swapping the outer-most wrongly-positioned pairs.

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

Hoare's In-Place Partition Routine

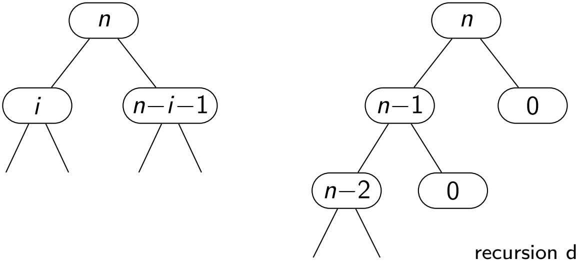
Loop invariant: A

$\leq v$	?	$\geq v$	v
i		j	$n{-}1$

Hoare-partition(A, p)
A: array of size n, p: integer s.t. $0 \leq p < n$
1. *swap*$(A[n{-}1], A[p])$
2. $i \leftarrow -1,\quad j \leftarrow n{-}1,\quad v \leftarrow A[n{-}1]$
3. **loop**
4.   **do** $i \leftarrow i{+}1$ **while** $A[i] < v$
5.   **do** $j \leftarrow j{-}1$ **while** $j > i$ and $A[j] > v$
6.   **if** $i \geq j$ **then break** (goto 9)
7.   **else** *swap*$(A[i], A[j])$
8. **end loop**
9. *swap*$(A[n{-}1], A[i])$
10. **return** i

Improvement ideas for quick-sort

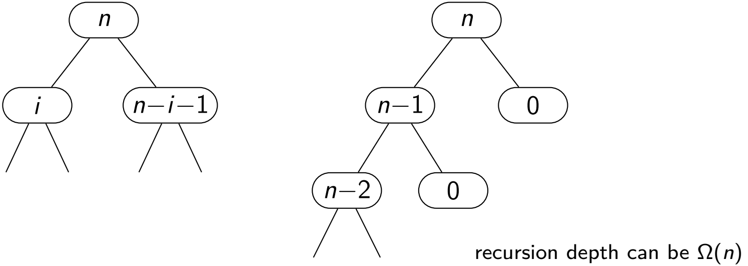
- Every recursive call uses $O(1)$ auxiliary space to store a record.
- *quick-sort* has nested recursive calls. To analyze its auxiliary space, consider the **recursion tree** and analyze its height (**recursion depth**)
  - Write size of subproblem into each node.
  - If $n \geq 2$ then there are two subproblems, hence two children.



recursion depth can be $\Omega(n)$

Improvement ideas for quick-sort

- Every recursive call uses $O(1)$ auxiliary space to store a record.
- *quick-sort* has nested recursive calls. To analyze its auxiliary space, consider the **recursion tree** and analyze its height (**recursion depth**)
  - Write size of subproblem into each node.
  - If $n \geq 2$ then there are two subproblems, hence two children.



- Recursion tree is also useful for analyzing the run-time:
  - On every level, the total number of key-comparisons is $\leq n$.
  - Can argue (later): On average, the height is $O(\log n)$.
  - This gives another proof of $O(n \log n)$ average-case run-time.

Auxiliary space for *quick-sort*

Claim: If we always continue in the *smaller* subproblem first, then the auxiliary space is in $\Theta(\log n)$.

Auxiliary space for *quick-sort*

Claim: If we always continue in the *smaller* subproblem first, then the auxiliary space is in $\Theta(\log n)$.

Proof: Consider the path in the recursion tree to the current subproblem.

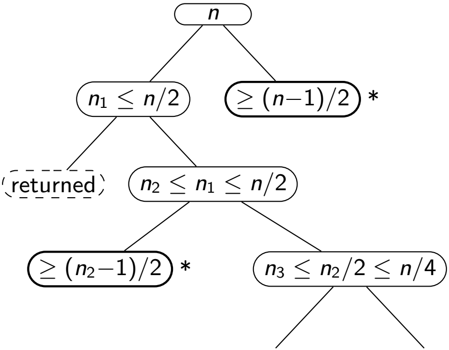
Auxiliary space for *quick-sort*

Claim: If we always continue in the *smaller* subproblem first, then the auxiliary space is in $\Theta(\log n)$.

Proof: Consider the path in the recursion tree to the current subproblem.

For each child:

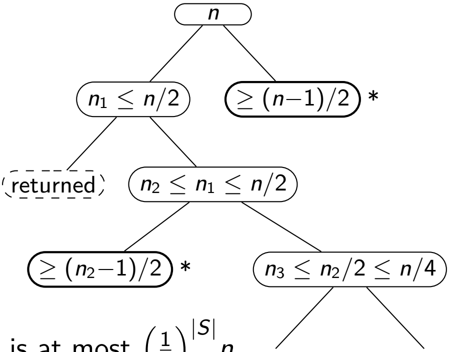
- Either halved the size (or better).
- Or the sibling is done $\Rightarrow$ not on stack



Auxiliary space for *quick-sort*

Claim: If we always continue in the *smaller* subproblem first, then the auxiliary space is in $\Theta(\log n)$.

Proof: Consider the path in the recursion tree to the current subproblem.



For each child:

- Either halved the size (or better).
- Or the sibling is done $\Rightarrow$ not on stack

At all times, the current problem size is at most $\left(\frac{1}{2}\right)^{|S|} n$.

$\Rightarrow$ At all times, $|S| \leq \log n$.

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - 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	$\Theta(n^2)$ $\Theta(n)$	worst-case best-case
merge-sort	$\Theta(n \log n)$	worst-case
heap-sort	$\Theta(n \log n)$	worst-case
quick-sort *randomized-quick-sort*	$\Theta(n \log n)$ $\Theta(n \log n)$	average-case expected

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
	$\Theta(n)$	best-case
merge-sort	$\Theta(n \log n)$	worst-case
heap-sort	$\Theta(n \log n)$	worst-case
quick-sort	$\Theta(n \log n)$	average-case
randomized-quick-sort	$\Theta(n \log n)$	expected

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

Lower bounds for sorting

We have seen many sorting algorithms:

Sort	Running time	Analysis
selection-sort	$\Theta(n^2)$	worst-case
insertion-sort	$\Theta(n^2)$ $\Theta(n)$	worst-case best-case
merge-sort	$\Theta(n \log n)$	worst-case
heap-sort	$\Theta(n \log n)$	worst-case
quick-sort *randomized-quick-sort*	$\Theta(n \log n)$ $\Theta(n \log n)$	average-case 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!). ($\rightarrow$ later)

Lower bound for sorting in the comparison model

All algorithms so far are **comparison-based**: Data is accessed only by

- comparing two elements (a *key-comparison*)
- moving elements around (e.g. copying, swapping)

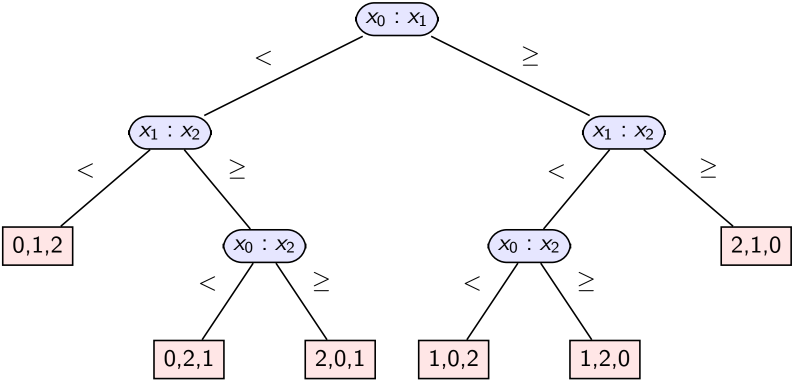
Theorem. Any *comparison-based* sorting algorithm requires in the worst case $\Omega(n \log n)$ comparisons to sort n distinct items.

Proof.

Decision trees

Any comparison-based algorithms can be expressed as **decision tree**.

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

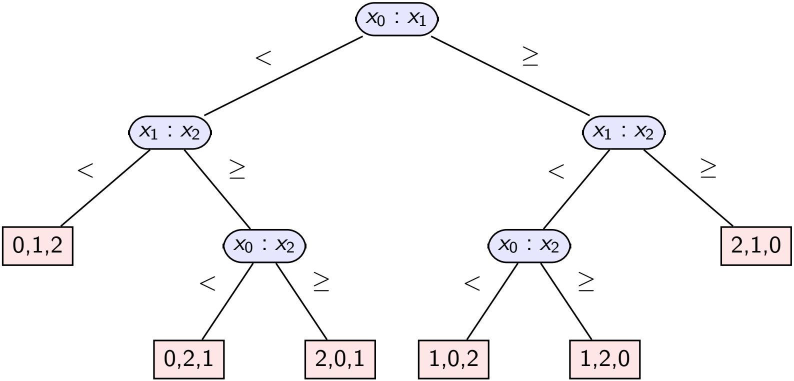


Decision trees

Any comparison-based algorithms can be expressed as **decision tree**.

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

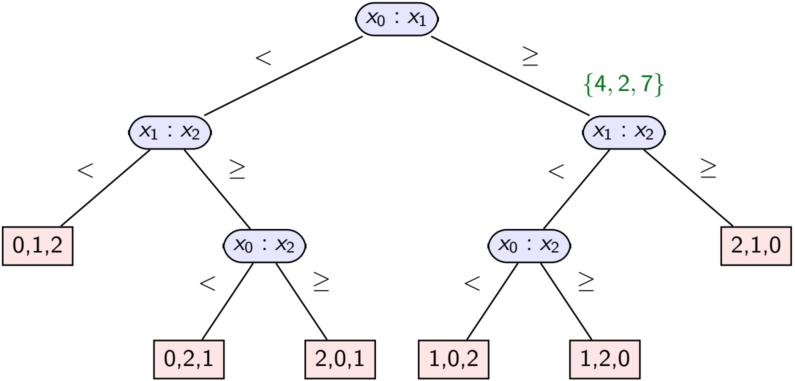
Example: $\{x_0{=}4, x_1{=}2, x_2{=}7\}$



Decision trees

Any comparison-based algorithms can be expressed as **decision tree**.

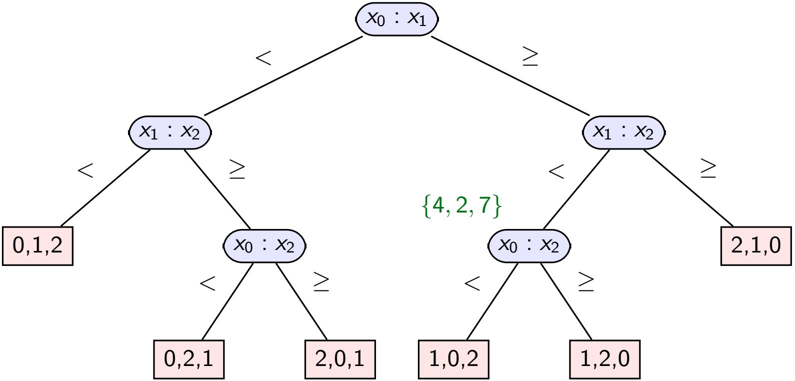
To sort $\{x_0, x_1, x_2\}$:



Decision trees

Any comparison-based algorithms can be expressed as **decision tree**.

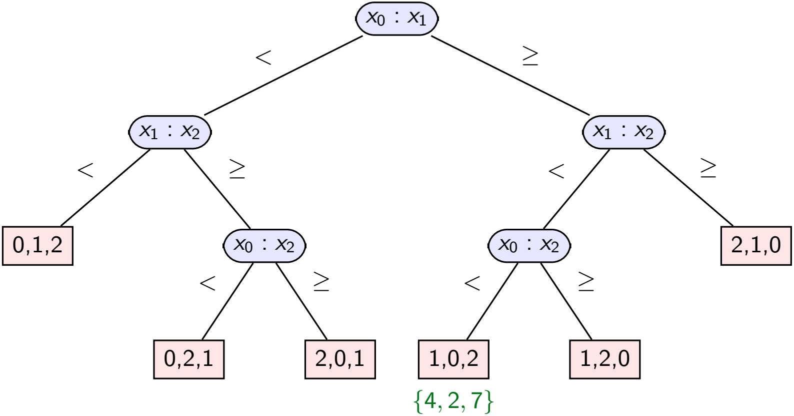
To sort $\{x_0, x_1, x_2\}$:



Decision trees

Any comparison-based algorithms can be expressed as **decision tree**.

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



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

(i.e., $x_1{=}2 \leq x_0{=}4 \leq x_2{=}7$)

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - SELECTION and *quick-select*
   - Randomized Algorithms
   - *quick-select* revisited
   - Tips and Tricks for *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Non-Comparison-Based Sorting

- Assume keys are numbers in base R (R: **radix**)
  - So all digits are in $\{0, \ldots, R-1\}$
  - $R = 2, 10, 128, 256$ are the most common, but R need not be constant

Example ($R = 4$):

123	230	21	320	210	232	101

- Assume all keys have the same number w of digits.
  - Can achieve after padding with leading 0s.
  - In typical computers, $w = 32$ or $w = 64$, but w need not be constant

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:

A
123
230
021
320
210
232
101

$\Longrightarrow$

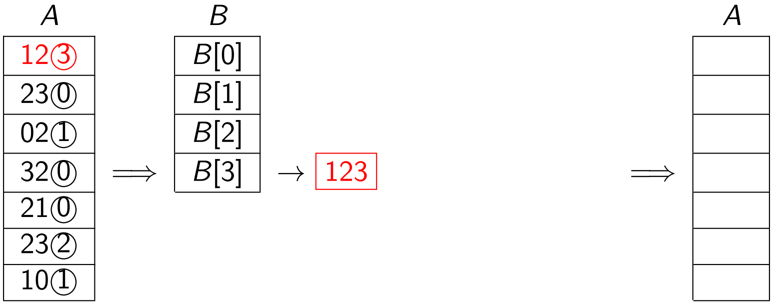
B
$B[0]$
$B[1]$
$B[2]$
$B[3]$

$\Longrightarrow$

A

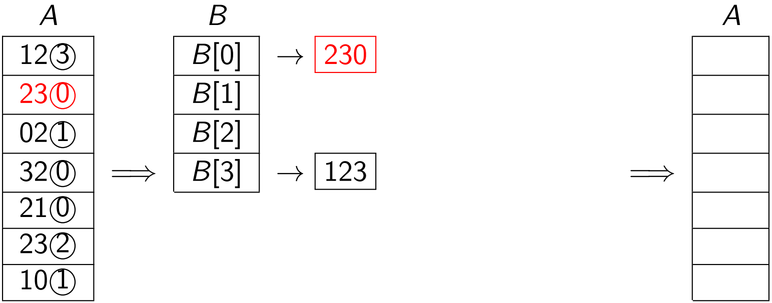
(Single-digit) *bucket-sort*

Sort array A by last digit:



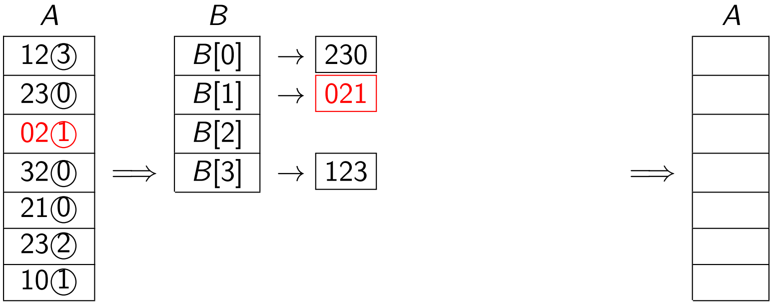
(Single-digit) *bucket-sort*

Sort array A by last digit:



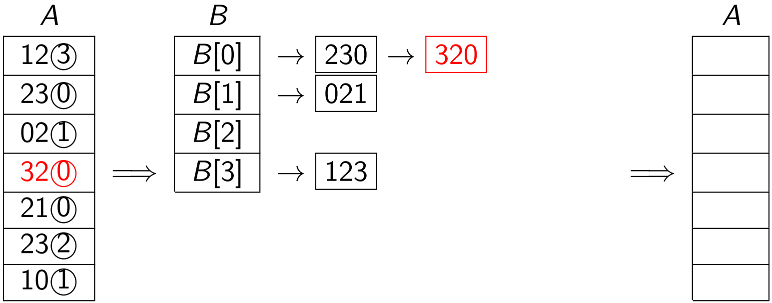
(Single-digit) *bucket-sort*

Sort array A by last digit:



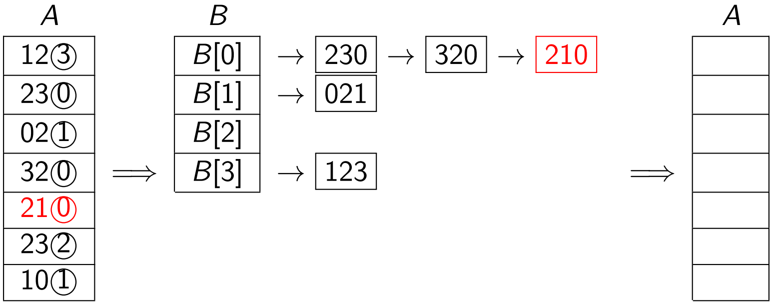
(Single-digit) *bucket-sort*

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

$\Longrightarrow$

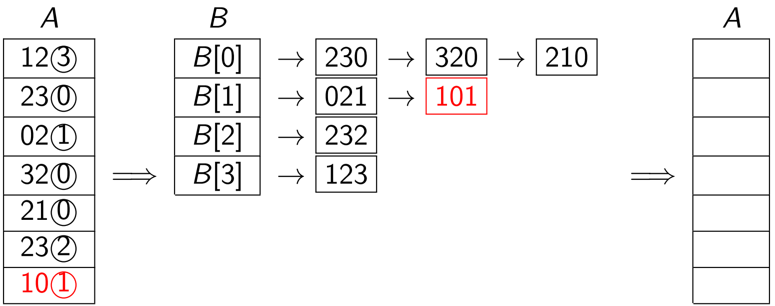
B						
$B[0]$	$\rightarrow$	230	$\rightarrow$	320	$\rightarrow$	210
$B[1]$	$\rightarrow$	021				
$B[2]$	$\rightarrow$	232				
$B[3]$	$\rightarrow$	123				

$\Longrightarrow$

A

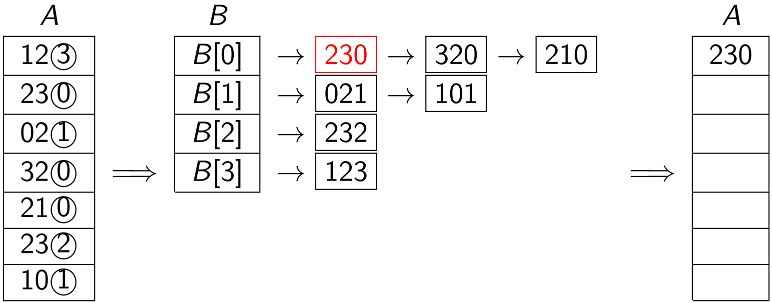
(Single-digit) *bucket-sort*

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

$B[0] \to$ 230 $\to$ 320 $\to$ 210

$B[1] \to$ 021 $\to$ 101

$B[2] \to$ 232

$B[3] \to$ 123

$\Longrightarrow$

A
230
320

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

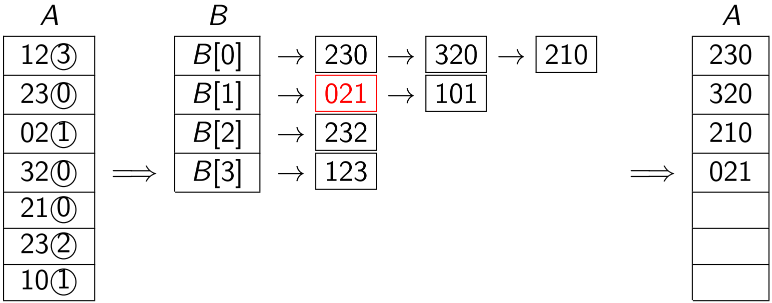
B			
$B[0]$	$\rightarrow$ 230	$\rightarrow$ 320	$\rightarrow$ 210
$B[1]$	$\rightarrow$ 021	$\rightarrow$ 101	
$B[2]$	$\rightarrow$ 232		
$B[3]$	$\rightarrow$ 123		

$\Longrightarrow$

A
230
320
210

(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

(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

(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, n, sort-key(·))
A: array of size n
sort-key(·) : maps items of A to {0, ..., R−1}
    Initialize an array B[0...R − 1] of empty queues (buckets)
    for i ← 0 to n−1 do
        Append A[i] at end of B[sort-key(A[i])]
    i ← 0
    for j ← 0 to R − 1 do
        while B[j] is non-empty do
            move front element of B[j] to A[i++]
```

- In our example *sort-key*$(A[i])$ returns the last digit of $A[i]$

(Single-digit) *bucket-sort*

bucket-sort$(A, n,$ *sort-key*$(\cdot))$
A: array of size n
sort-key$(\cdot)$: maps items of A to $\{0, \ldots, R-1\}$

```
Initialize an array B[0...R − 1] of empty queues (buckets)
for i ← 0 to n−1 do
        Append A[i] at end of B[sort-key(A[i])]
i ← 0
for j ← 0 to R − 1 do
        while B[j] is non-empty do
                move front element of B[j] to A[i++]
```

- In our example *sort-key*$(A[i])$ returns the last digit of $A[i]$
- *bucket-sort* 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 arrays $\rightsquigarrow$ *count-sort* (no details).

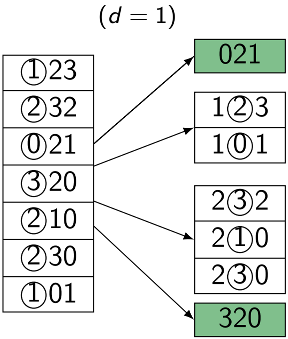
Most-significant-digit(MSD)-radix-sort

Sort array of w-digit radix-R numbers recursively:
sort by 1st digit, then each group by 2nd digit, etc.

123
232
021
320
210
230
101

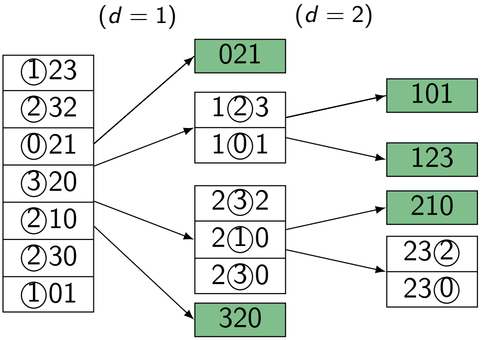
Most-significant-digit(MSD)-radix-sort

Sort array of w-digit radix-R numbers recursively:
sort by 1st digit, then each group by 2nd digit, etc.



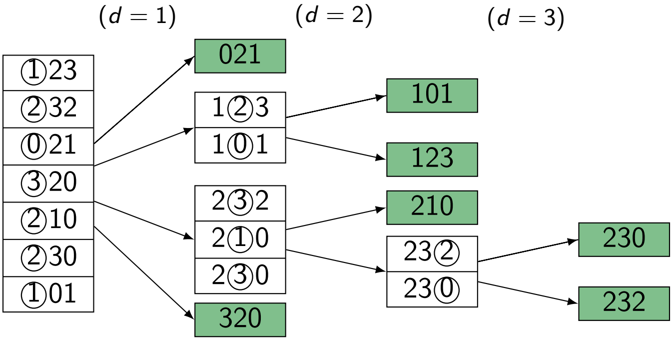
Most-significant-digit(MSD)-radix-sort

Sort array of w-digit radix-R numbers recursively:
sort by 1st digit, then each group by 2nd digit, etc.



Most-significant-digit(MSD)-radix-sort

Sort array of w-digit radix-R numbers recursively:
sort by 1st digit, then each group by 2nd digit, etc.



MSD-radix-sort

MSD-radix-sort$(A, n, \quad d \leftarrow 1)$
A: array of size n, contains w-digit radix-R numbers
  if $(d \leq w$ and $(n > 1))$
    bucket-sort(A, n,'return dth digit of $A[i]$')
    $\ell \leftarrow 0$ `// find sub-arrays and recurse`
    for $j \leftarrow 0$ to $R-1$
      Let $r \geq \ell - 1$ be maximal s.t. $A[\ell..r]$ have dth digit j
      MSD-radix-sort$(A[\ell..r], r-\ell+1, d+1)$
      $\ell \leftarrow r + 1$

Analysis:

- $\Theta(w)$ levels of recursion in worst-case.
- $\Theta(n)$ subproblems on most levels in worst-case.
- $\Theta(R + (\text{size of sub-array}))$ time for each *bucket-sort* call.

$\Rightarrow$ Run-time $\Theta(wnR)$ — slow. Many recursions and allocated arrays.

Least-significant-digit(LSD)-radix-sort

LSD-radix-sort(A, n)
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, n, 'return dth digit of $A[i]$')

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

- Loop-invariant: A is sorted w.r.t. digits $d, \ldots, w$ of each entry.
- **Time cost**: $\Theta(w(n+R))$ **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
- *heap-sort* is the only $\Theta(n \log n)$-time algorithm we have seen with $O(1)$ auxiliary space.
- *merge-sort* is also $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.
- *quick-sort* is worst-case $\Theta(n^2)$, but often the fastest in practice
- *bucket-sort* and *radix-sort* achieve $o(n \log n)$ if the input is special

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case can all differ.
- Often it is easier to analyze the run-time on randomly chosen input rather than the average-case run-time.