

CS 240 – Data Structures and Data Management

Module 3: Sorting, Average-case and Randomization

Arne Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2024

Outline

3 Sorting, Average-case and Randomization

- Analyzing average-case run-time
- SELECTION and *quick-select*
- Randomized Algorithms
- *quick-select* revisited
- SORTING and *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
- SORTING and *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:

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

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 $\{0, \dots, n-1\}$, stored as an array

1. **if** $\pi[0] = 0$ **then for** $j = 1$ **to** n **do** print 'bad case'
2. **else** print 'good case'

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

A simple (contrived) example

silly-test(π, 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'

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

- $(n-1)!$ 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(π, 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'

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

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

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

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 $n = 0$.

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

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

Average-case run-time?

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

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

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*

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

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 $T^{\text{avg}}(n)$:

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

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

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

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 $T^{\text{avg}}(n)$:

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

- 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*

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

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

$$\begin{aligned} T^{avg}(n) &= \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w) \\ &= \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=1}} 1 + \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0}} \left(1 + T(w[0..n-2]) \right) \end{aligned}$$

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

$$\begin{aligned}T^{avg}(n) &= \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w) \\&= \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=1}} 1 + \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0}} \left(1 + T(w[0..n-2])\right) \\&= \frac{1}{2} + \frac{1}{2} + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0 \\ w[0..n-2]=w'}} T(w')\end{aligned}$$

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

$$\begin{aligned}T^{avg}(n) &= \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w) \\&= \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=1}} 1 + \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0}} \left(1 + T(w[0..n-2])\right) \\&= \frac{1}{2} + \frac{1}{2} + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0 \\ w[0..n-2]=w'}} T(w')\end{aligned}$$

Observe: w is uniquely determined by w' and $w[n-1] = 0$

$$= 1 + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} T(w')$$

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

$$\begin{aligned}T^{avg}(n) &= \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w) \\&= \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=1}} 1 + \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0}} \left(1 + T(w[0..n-2])\right) \\&= \frac{1}{2} + \frac{1}{2} + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0 \\ w[0..n-2]=w'}} T(w')\end{aligned}$$

Observe: w is uniquely determined by w' and $w[n-1] = 0$

$$\begin{aligned}&= 1 + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} T(w') \\&= 1 + \frac{|\mathcal{B}_{n-1}|}{|\mathcal{B}_n|} \frac{1}{|\mathcal{B}_{n-1}|} \sum_{w' \in \mathcal{B}_{n-1}} T(w') = 1 + \frac{1}{2} T^{avg}(n-1)\end{aligned}$$

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

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

$$\begin{aligned}T^{avg}(n) &= \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w) \\&= \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=1}} 1 + \frac{1}{|\mathcal{B}_n|} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0}} \left(1 + T(w[0..n-2])\right) \\&= \frac{1}{2} + \frac{1}{2} + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} \sum_{\substack{w \in \mathcal{B}_n \\ w[n-1]=0 \\ w[0..n-2]=w'}} T(w')\end{aligned}$$

Observe: w is uniquely determined by w' and $w[n-1] = 0$

$$\begin{aligned}&= 1 + \frac{1}{|\mathcal{B}_n|} \sum_{w' \in \mathcal{B}_{n-1}} T(w') \\&= 1 + \frac{|\mathcal{B}_{n-1}|}{|\mathcal{B}_n|} \frac{1}{|\mathcal{B}_{n-1}|} \sum_{w' \in \mathcal{B}_{n-1}} T(w') = 1 + \frac{1}{2} T^{avg}(n-1)\end{aligned}$$

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

1. **if** ($n = 0$) **then return** true
2. **if** ($w[n-1] = 1$) **then return** false
3. **if** ($n > 1$) **then** $w[n-2] \leftarrow 0$ // this is the only change
4. *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

1. **if** ($n = 0$) **then return** true
2. **if** ($w[n-1] = 1$) **then return** false
3. **if** ($n > 1$) **then** $w[n-2] \leftarrow 0$ // 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 $T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ n & \text{if } w[n-1] = 0 \end{cases}$.

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** $w[n-2] \leftarrow 0$ // 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
- SORTING and *quick-sort*
- Lower Bound for Comparison-Based Sorting
- Non-Comparison-Based 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.

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.

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.

The encountered sub-routines will also be useful otherwise.

Crucial Subroutines

quick-select and the related *quick-sort* 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

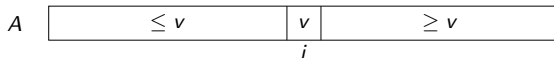
quick-select and the related *quick-sort* rely on two subroutines:

- *choose-pivot*(A): Return an index p in A . We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.
 - ▶ For now simply use $p = A.size - 1$, so v is rightmost item
 - ▶ We will consider more sophisticated ideas later on.

Crucial Subroutines

quick-select and the related *quick-sort* rely on two subroutines:

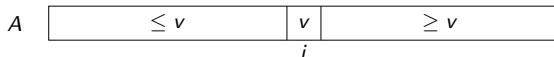
- *choose-pivot*(A): Return an index p in A . We will use the **pivot-value** $v \leftarrow A[p]$ to rearrange the array.
 - ▶ For now simply use $p = A.size - 1$, so v is rightmost item
 - ▶ We will consider more sophisticated ideas later on.
- *partition*(A, p): Rearrange A and return **pivot-index** i so that
 - ▶ the pivot-value v is in $A[i]$,
 - ▶ all items in $A[0, \dots, i-1]$ are $\leq v$, and
 - ▶ all items in $A[i+1, \dots, n-1]$ are $\geq v$.



Crucial Subroutines

quick-select and the related *quick-sort* 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.
 - ▶ For now simply use $p = A.size - 1$, so v is rightmost item
 - ▶ 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 $A[0, \dots, i-1]$ are $\leq v$, and
 - ▶ all items in $A[i+1, \dots, n-1]$ are $\geq v$.



- p = index of pivot-value before *partition* (we choose it)
 i = index of pivot-value after *partition* (no control)

Partition Algorithm

Conceptually easy linear-time implementation:

partition(A, p)

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

1. Create empty lists *smaller*, *equal* and *larger*.
2. $v \leftarrow A[p]$
3. **for** each element x in A **do**
4. **if** $x < v$ **then** *smaller.append*(x)
5. **else if** $x > v$ **then** *larger.append*(x)
6. **else** *equal.append*(x).
7. $i \leftarrow \text{smaller.size}$
8. $j \leftarrow \text{equal.size}$
9. Overwrite $A[0 \dots i-1]$ by elements in *smaller*
10. Overwrite $A[i \dots i+j-1]$ by elements in *equal*
11. Overwrite $A[i+j \dots n-1]$ by elements in *larger*
12. return i

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

Efficient In-Place partition (Hoare) - Example

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	$v=70$

Efficient In-Place partition (Hoare) - Example

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

Efficient In-Place partition (Hoare) - Example

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

Efficient In-Place partition (Hoare) - Example

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	$v=70$

Efficient In-Place partition (Hoare) - Example

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

Efficient In-Place partition (Hoare) - Example

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

Efficient In-Place partition (Hoare) - Example

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

Efficient In-Place partition (Hoare)

Loop invariant: A

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

partition(A, p)

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

1. $\text{swap}(A[n-1], A[p])$
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** $\text{swap}(A[i], A[j])$
8. **end loop**
9. $\text{swap}(A[n-1], A[i])$
10. **return** i

Running time: $\Theta(n)$.

Observe: n **key-comparisons** (comparing two input-items).

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. $0 \leq k < n$

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

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 $k = 0$
 $T^{\text{worst}}(n, 0) \geq n + (n-1) + (n-2) + \dots + 1 \in \Omega(n^2)$

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 $k = 0$
 $T^{\text{worst}}(n, 0) \geq n + (n-1) + (n-2) + \dots + 1 \in \Omega(n^2)$

Best-case run-time: First chosen pivot could be the k th 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 $k = 0$
 $T^{\text{worst}}(n, 0) \geq n + (n-1) + (n-2) + \dots + 1 \in \Omega(n^2)$

Best-case run-time: First chosen pivot could be the k th 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
- SORTING and *quick-sort*
- Lower Bound for Comparison-Based Sorting
- Non-Comparison-Based Sorting

Randomized algorithms

- A **randomized algorithm** is one which relies on some random numbers in addition to the input.
 - ▶ Doing randomization is often a good idea if an algorithm has bad worst-case time but seems to perform much better on most instances.
 - ▶ It can also (with restrictions) be used to bound the avg-case run-time.
- The run-time will depend on the input and the random numbers used.

(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!)

- **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.

Example (again 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 $\{0, 1, 2, \dots, n-1\}$. So $Pr(\text{random}(2) = 0) = \frac{1}{2}$.

Example (again 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 $\{0, 1, 2, \dots, n-1\}$. So $\Pr(\text{random}(2) = 0) = \frac{1}{2}$.

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

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time** $T^{\text{exp}}(I)$ **for instance** I is the expected value:

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

Expected run-time

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time** $T^{\text{exp}}(I)$ **for instance** I is the expected value:

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

Now take the *maximum* over all instances of size n to define the **expected run-time** (or formally: *worst-instance expected-luck run-time*) **of** $\mathcal{A}$.

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

Expected run-time

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time** $T^{\text{exp}}(I)$ **for instance** I is the expected value:

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

Now take the *maximum* over all instances of size n to define the **expected run-time** (or formally: *worst-instance expected-luck run-time*) **of** $\mathcal{A}$.

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

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

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 $T^{\text{exp}}(n)$:

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

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*:

$$T^{\text{exp}}(w) = \sum_R \Pr(R) T(w, R) =$$

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*:

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

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*:

$$\begin{aligned} T^{\text{exp}}(w) &= \sum_R \Pr(R) T(w, R) = \sum_x \sum_{R'} \Pr(x) \Pr(R') T(w, \langle x, R' \rangle) \\ &= \Pr(x=w[n-1]) \sum_{R'} \Pr(R') \cdot 1 \\ &\quad + \Pr(x \neq w[n-1]) \sum_{R'} \Pr(R') (1 + T(w[0..n-2], R')) \end{aligned}$$

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*:

$$\begin{aligned}T^{\text{exp}}(w) &= \sum_R \Pr(R) T(w, R) = \sum_x \sum_{R'} \Pr(x) \Pr(R') T(w, \langle x, R' \rangle) \\&= \Pr(x=w[n-1]) \sum_{R'} \Pr(R') \cdot 1 \\&\quad + \Pr(x \neq w[n-1]) \sum_{R'} \Pr(R') (1 + T(w[0..n-2], R')) \\&= \frac{1}{2} \quad + \frac{1}{2} \quad + \frac{1}{2} \underbrace{\sum_{R'} \Pr(R') \cdot T(w[0..n-2], R')}_{T^{\text{exp}}(\text{some instance of size } n-1)}\end{aligned}$$

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*:

$$\begin{aligned}T^{\text{exp}}(w) &= \sum_R \Pr(R) T(w, R) = \sum_x \sum_{R'} \Pr(x) \Pr(R') T(w, \langle x, R' \rangle) \\&= \Pr(x=w[n-1]) \sum_{R'} \Pr(R') \cdot 1 \\&\quad + \Pr(x \neq w[n-1]) \sum_{R'} \Pr(R') (1 + T(w[0..n-2], R')) \\&= \frac{1}{2} \quad + \frac{1}{2} \quad + \frac{1}{2} \underbrace{\sum_{R'} \Pr(R') \cdot T(w[0..n-2], R')}_{T^{\text{exp}}(\text{some instance of size } n-1)} \\&\leq 1 + \frac{1}{2} \max_{w' \in \mathcal{B}_{n-1}} T^{\text{exp}}(w') = 1 + \frac{1}{2} T^{\text{exp}}(n-1) \quad \text{holds for *all* } w\end{aligned}$$

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

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

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

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

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

shuffled- $\mathcal{A}$ (n)

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

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

Avg-case run-time via expected run-time

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

shuffled- $\mathcal{A}$ (n)

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

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

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

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

Avg-case run-time via expected run-time

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

shuffled- $\mathcal{A}(n)$

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

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

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

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

- 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 $T_{\mathcal{A}}^{avg}(n)$.

Avg-case run-time via expected run-time

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

shuffled-all-0-test(n)

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

randomized-all-0-test(w, n)

1. **for** ($i = n-1; i \geq 0; i--$) **do**
2. **if** ($\text{random}(2)=0$) **then**
 $w[i] = 1 - w[i]$
3. **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):

shuffled-all-0-test(n)

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

randomized-all-0-test(w, n)

1. **for** ($i = n-1; i \geq 0; i--$) **do**
2. **if** ($\text{random}(2)=0$) **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 resp. 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):

shuffled-all-0-test(n)

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

randomized-all-0-test(w, n)

1. **for** ($i = n-1; i \geq 0; i--$) **do**
2. **if** ($\text{random}(2)=0$) **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 resp. 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 $T_{\text{all-0-test}}^{\text{avg}}(n) = T_{\text{shuffled-all-0-test}}^{\text{exp}}(n) = T_{\text{rand-all-0-test}}^{\text{exp}}(n) \in O(1)$
can be deduced without analyzing $T_{\text{all-0-test}}^{\text{avg}}(n)$ 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
$\frac{1}{ \mathcal{I}_n } \sum_{I \in \mathcal{I}_n} T(I)$	$\max_{I \in \mathcal{I}_n} \sum_{\text{outcomes } R} \Pr(R) \cdot T(I, R)$
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
$\frac{1}{ \mathcal{I}_n } \sum_{I \in \mathcal{I}_n} T(I)$	$\max_{I \in \mathcal{I}_n} \sum_{\text{outcomes } R} \Pr(R) \cdot T(I, R)$
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
- SORTING and *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.   $i \leftarrow \text{partition}(A, n-1)$ 
2.  if  $i = k$  then return  $A[i]$ 
3.  else if  $i > k$  then quick-select( $A[0 \dots i-1]$ ,  $k$ )
4.  else if  $i < k$  then quick-select( $A[i+1 \dots n-1]$ ,  $k - (i+1)$ )
```

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.
 - ▶ This is not completely realistic (mostly-sorted orders 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*.

```
shuffled-quick-select(A, k)
```

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

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

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

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)
1. swap A[n-1] with A[random(n)]
2. i ← 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 ... i-1], k)
6. else if i < k then
7.     return randomized-quick-select(A[i+1 ... n-1], k - (i+1))
```

• $T_{\text{rand.-quick-select}}^{\text{exp}}(n) = T_{\text{shuffled-quick-select}}^{\text{exp}}(n).$

(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-index is i ')
 - Observe: $\Pr(\text{pivot-index is } i) = \frac{1}{n}$
 - 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-index is i ')
 - Observe: $\Pr(\text{pivot-index is } i) = \frac{1}{n}$
 - 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 $T^{\text{exp}}(\text{array-size})$.

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

Analysis of *randomized-quick-select*

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

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

Analysis of *randomized-quick-select*

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

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

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 *shuffled-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 *shuffled-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. (→ cs341, maybe)

Outline

3 Sorting, Average-case and Randomization

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

quick-sort

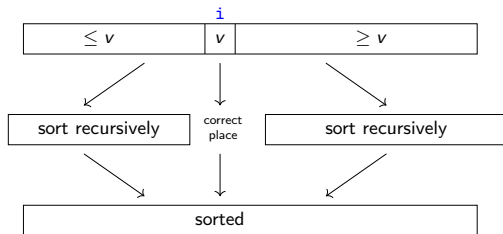
Hoare developed *partition* and *quick-select* in 1960.

He also used them to *sort* based on partitioning:

quick-sort(A)

A: array of size n

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



quick-sort analysis

Set $T(A) := \#$ of key-comparison for *quick-sort* in array A .

Worst-case run-time: $\Theta(n^2)$

- Sub-arrays get smaller $\Rightarrow \leq n$ levels of recursions
- On each level there are $\leq n$ items in total $\Rightarrow \leq n$ key-comparisons
- So run-time in $O(n^2)$; this is tight exactly as for *quick-select*

quick-sort analysis

Set $T(A) := \#$ of key-comparison for *quick-sort* in array A .

Worst-case run-time: $\Theta(n^2)$

- Sub-arrays get smaller $\Rightarrow \leq n$ levels of recursions
- On each level there are $\leq n$ items in total $\Rightarrow \leq n$ key-comparisons
- So run-time in $O(n^2)$; this is tight exactly as for *quick-select*

Best-case run-time: $\Theta(n)$

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

quick-sort analysis

Set $T(A) := \#$ of key-comparison for *quick-sort* in array A .

Worst-case run-time: $\Theta(n^2)$

- Sub-arrays get smaller $\Rightarrow \leq n$ levels of recursions
- On each level there are $\leq n$ items in total $\Rightarrow \leq n$ key-comparisons
- So run-time in $O(n^2)$; this is tight exactly as for *quick-select*

Best-case run-time: $\Theta(n)$

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

Average-case run-time? We again prove this via randomization.

Randomizing quick-sort

randomized-quick-sort(A)

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

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

Randomizing quick-sort

randomized-quick-sort(A)

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

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

Assume the random output was such that the pivot-index 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. $p \leftarrow \text{random}(n)$
3. $i \leftarrow \text{partition}(A, p)$
4. *randomized-quick-sort*(A[0, 1, ..., $i-1$])
5. *randomized-quick-sort*(A[$i+1$, ..., $n-1$])

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

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

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

This implies

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

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

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

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

Proof:

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

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

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

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 $\Theta(n \log n)$.

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 $\Theta(n \log n)$.
- Auxiliary space?
 - ▶ Each nested recursion-call requires $\Theta(1)$ space on the call stack.
 - ▶ As described, *quick-sort/randomized-quick-sort* use $\Omega(n)$ nested recursion-calls in the worst case.
 - ▶ So $\Theta(n)$ auxiliary space (can be improved to $\Theta(\log n)$)

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 $\Theta(n \log n)$.
- Auxiliary space?
 - ▶ Each nested recursion-call requires $\Theta(1)$ space on the call stack.
 - ▶ As described, *quick-sort/randomized-quick-sort* use $\Omega(n)$ nested recursion-calls in the worst case.
 - ▶ So $\Theta(n)$ auxiliary space (can be improved to $\Theta(\log n)$)
- There are numerous tricks to improve *randomized-quick-select*
- With these, this is in practice the fastest solution to SORTING (but *not* in theory).

quick-sort with tricks

randomized-quick-sort-improved(A, n)

1. Initialize a stack S of index-pairs with $\{(0, n-1)\}$
2. **while** S is not empty
3. $(\ell, r) \leftarrow S.\text{pop}()$
4. **while** $(r - \ell + 1 > 10)$ **do**
5. $p \leftarrow \ell + \text{random}(\ell - r + 1)$
6. $i \leftarrow \text{partition}(A, \ell, r, p)$
7. **if** $(i - \ell > r - i)$ **do**
8. $S.\text{push}((\ell, i - 1))$
9. $\ell \leftarrow i + 1$
10. **else**
11. $S.\text{push}((i + 1, r))$
12. $r \leftarrow i - 1$
13. *insertion-sort*(A)

Outline

3 Sorting, Average-case and Randomization

- Analyzing average-case run-time
- SELECTION and *quick-select*
- Randomized Algorithms
- *quick-select* revisited
- SORTING and *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
<i>selection-sort</i>	$\Theta(n^2)$	worst-case
<i>insertion-sort</i>	$\Theta(n^2)$ $\Theta(n)$	worst-case best-case
<i>merge-sort</i>	$\Theta(n \log n)$	worst-case
<i>heap-sort</i>	$\Theta(n \log n)$	worst-case
<i>quick-sort</i>	$\Theta(n \log n)$	average-case
<i>randomized-quick-sort</i>	$\Theta(n \log n)$	expected

Lower bounds for sorting

We have seen many sorting algorithms:

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

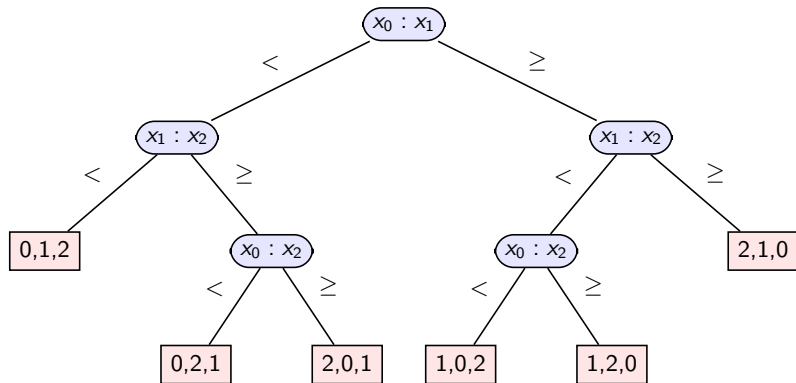
Theorem. Any *comparison-based* sorting algorithm requires in the worst case $\Omega(n \log n)$ comparisons to sort n distinct items.

Proof.

Decision trees

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

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

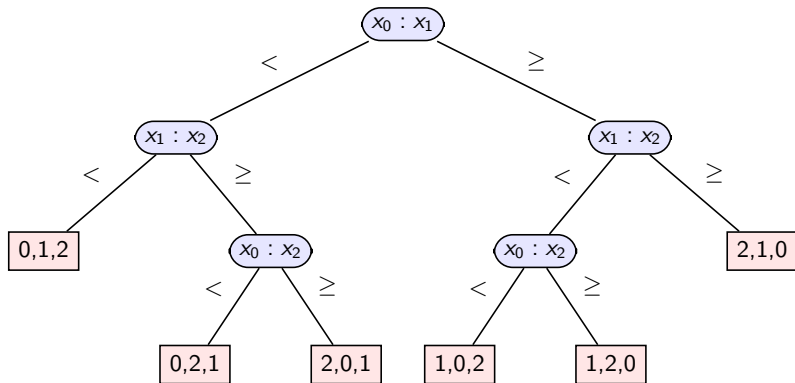


Decision trees

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

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

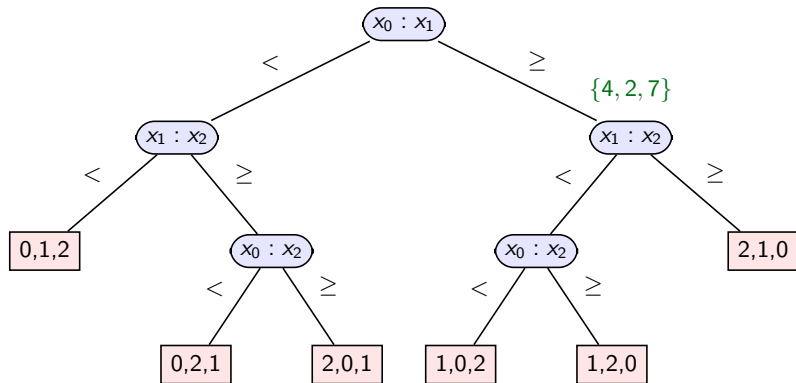
Example: $\{x_0=4, x_1=2, x_2=7\}$



Decision trees

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

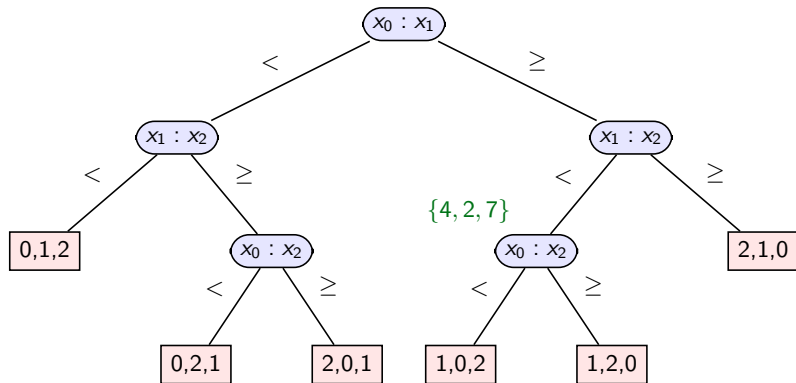
To sort $\{x_0, x_1, x_2\}$:



Decision trees

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

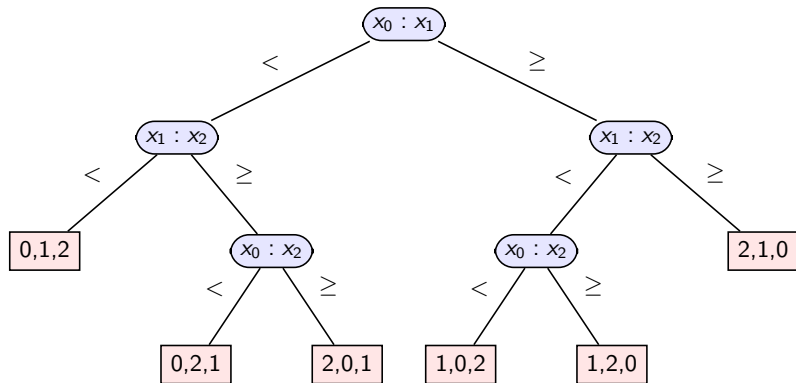
To sort $\{x_0, x_1, x_2\}$:



Decision trees

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

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



$\{4, 2, 7\}$

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
- SORTING and *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 $\{0, \dots, R-1\}$
 - ▶ $R = 2, 10, 128, 256$ 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 m of digits.
 - ▶ Can achieve after padding with leading 0s.
 - ▶ In typical computers, $m = 32$ or $m = 64$, but m need not be constant

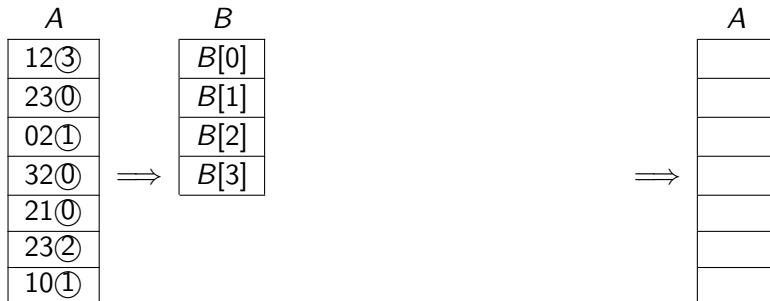
Example ($R = 4$):

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?

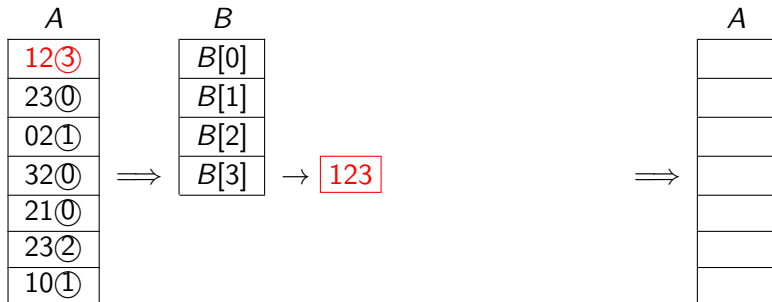
(Single-digit) *bucket-sort*

Sort array A by last digit:



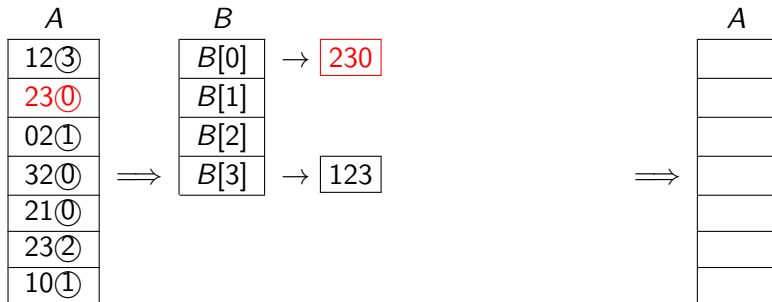
(Single-digit) *bucket-sort*

Sort array A by last digit:



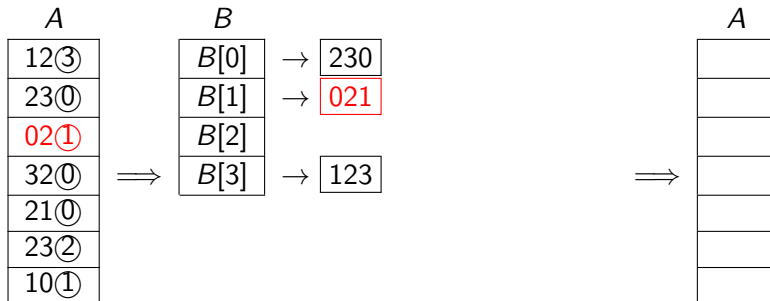
(Single-digit) *bucket-sort*

Sort array A by last digit:



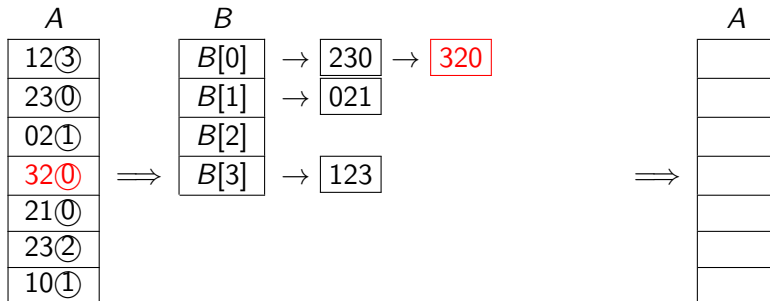
(Single-digit) *bucket-sort*

Sort array A by last digit:



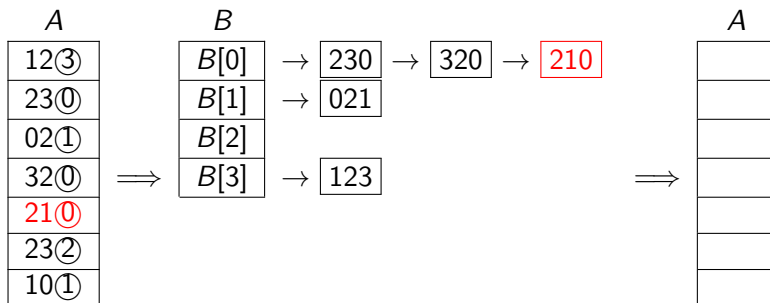
(Single-digit) *bucket-sort*

Sort array A by last digit:



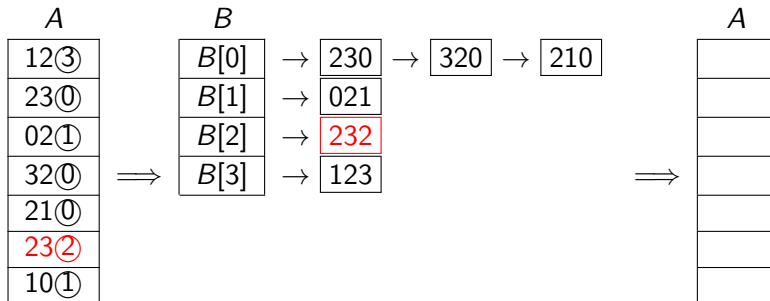
(Single-digit) *bucket-sort*

Sort array A by last digit:



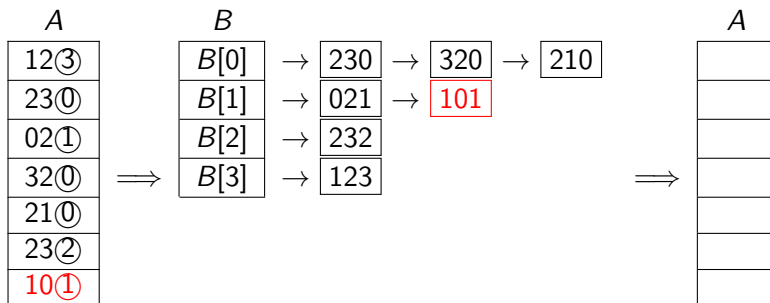
(Single-digit) *bucket-sort*

Sort array A by last digit:



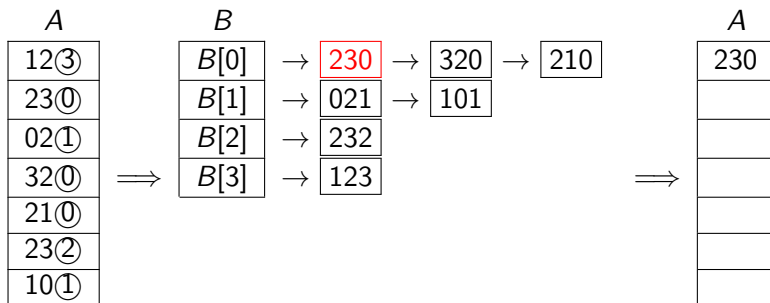
(Single-digit) *bucket-sort*

Sort array A by last digit:



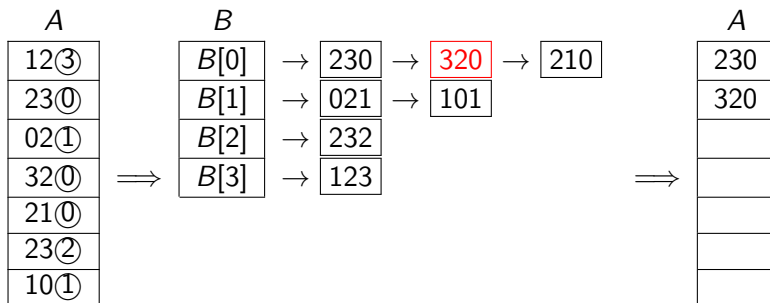
(Single-digit) *bucket-sort*

Sort array A by last digit:



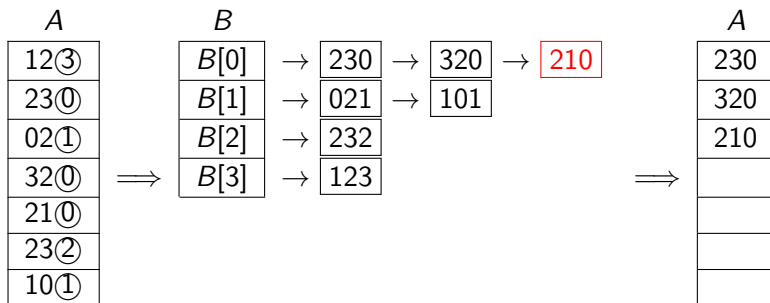
(Single-digit) *bucket-sort*

Sort array A by last digit:



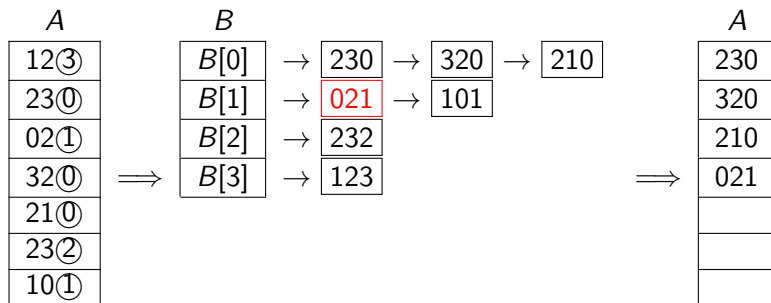
(Single-digit) *bucket-sort*

Sort array A by last digit:



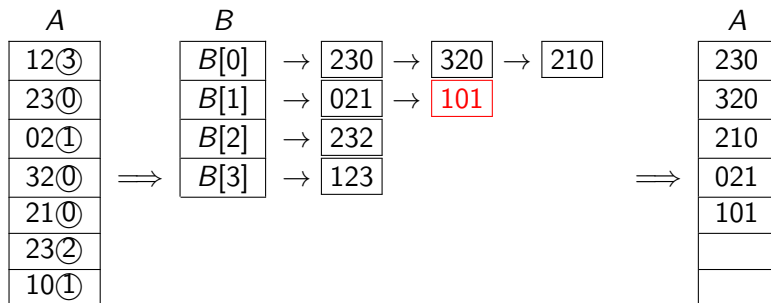
(Single-digit) *bucket-sort*

Sort array A by last digit:



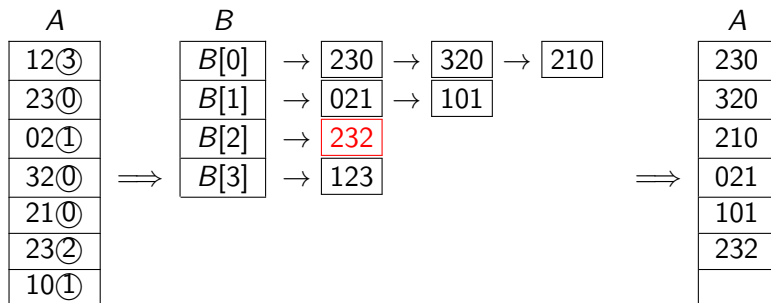
(Single-digit) *bucket-sort*

Sort array A by last digit:



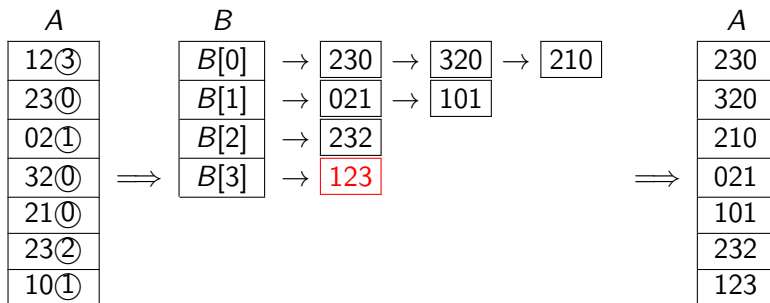
(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

bucket-sort($A, n, \text{sort-key}(\cdot)$)

A : array of size n

sort-key($\cdot$) : maps items of A to $\{0, \dots, R-1\}$

1. Initialize an array $B[0 \dots R-1]$ of empty queues (**buckets**)
2. **for** $i \leftarrow 0$ to $n-1$ **do**
3. Append $A[i]$ at end of $B[\text{sort-key}(A[i])]$
4. $i \leftarrow 0$
5. **for** $j \leftarrow 0$ to $R-1$ **do**
6. **while** $B[j]$ is non-empty **do**
7. 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($A, n, \text{sort-key}(\cdot)$)

A : array of size n

sort-key($\cdot$) : maps items of A to $\{0, \dots, R-1\}$

1. Initialize an array $B[0 \dots R-1]$ of empty queues (**buckets**)
2. **for** $i \leftarrow 0$ to $n-1$ **do**
3. Append $A[i]$ at end of $B[\text{sort-key}(A[i])]$
4. $i \leftarrow 0$
5. **for** $j \leftarrow 0$ to $R-1$ **do**
6. **while** $B[j]$ is non-empty **do**
7. 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).

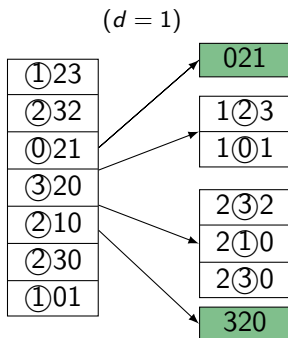
Most-significant-digit(MSD)-radix-sort

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

①23
②32
①01
③20
②10
②30
①01

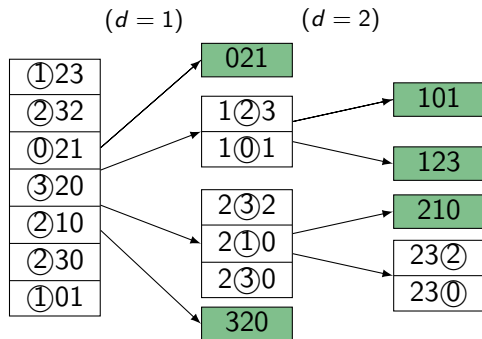
Most-significant-digit(MSD)-radix-sort

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



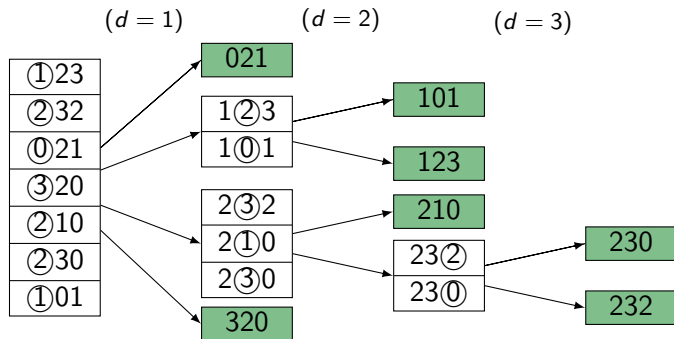
Most-significant-digit(MSD)-radix-sort

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



MSD-radix-sort

MSD-radix-sort($A, n, d \leftarrow 1$)

A : array of size n , contains m -digit radix- R numbers

1. **if** ($d \leq m$ and $(n > 1)$)
2. *bucket-sort*($A, n, \text{'return } d\text{th digit of } A[i]\text{'}$)
3. $\ell \leftarrow 0$ // find sub-arrays and recurse
4. **for** $j \leftarrow 0$ to $R - 1$
5. Let $r \geq \ell - 1$ be maximal s.t. $A[\ell..r]$ have d th digit j
6. *MSD-radix-sort*($A[\ell..r], r - \ell + 1, d + 1$)
7. $\ell \leftarrow r + 1$

Analysis:

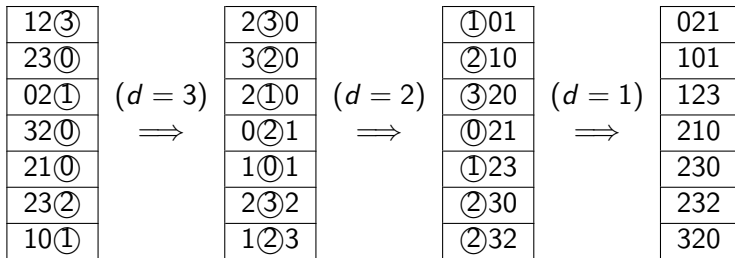
- $\Theta(m)$ levels of recursion in worst-case.
 - $\Theta(n)$ subproblems on most levels in worst-case.
 - $\Theta(R + (\text{size of sub-array}))$ time for each *bucket-sort* call.
- $\Rightarrow$ Run-time $\Theta(mnR)$ — slow. Many recursions and allocated arrays.

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 d th digit of $A[i]$ ')



- Loop-invariant: A is sorted w.r.t. digits $d, \dots, m$ 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
- *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.