

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

Module 3: Sorting, Average-case and Randomization

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version 2026-05-21 01:55

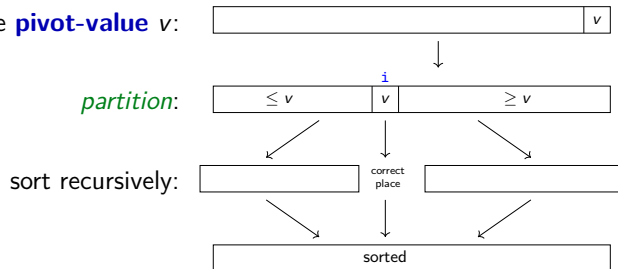
Outline

3 Sorting, Average-case and Randomization

- Review and Outlook
- Analyzing average-case run-time
- Run-time on randomly chosen input
- SELECTION and *quick-select*
- Tips and Tricks for *quick-sort*
- Lower Bound for Comparison-Based Sorting
- Non-Comparison-Based Sorting

Review and outlook

Recall the well-known *quick-sort* algorithm for SORTING:



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

Goal: Analyze *average-case* run-time via *randomization*.

Two detours needed:

- How does one analyze the average-case?
- And what does that have to do with randomization?

Average-case analysis

Recall definition of average-case run-time:

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

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*

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

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** $j \leftarrow 1$ **to** n **do** print 'bad case'
2. **else** print 'good case'

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

- bad case $\Rightarrow$ run-time $\leq c \cdot n$ (for some constant $c > 0$)
- good case $\Rightarrow$ run-time $\leq c$

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

A second (not-so-contrived) recursive 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*

$$T^{\text{avg}}(n) = \frac{1}{|\mathcal{B}_n|} \sum_{w \in \mathcal{B}_n} T(w). \quad (\mathcal{B}_n = \{\text{bitstrings of length } n\}, |\mathcal{B}_n| = 2^n)$$

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*

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

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

Easy induction proof: $T^{\text{avg}}(n) \leq 2 \in O(1)$.

Average-case analysis and recursions

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

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 $T^{\text{avg}}(n) = 1 + \frac{n}{2} \in \Theta(n)$. The “obvious” recursion did not hold.

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

Randomizations of algorithms

```
randomized-all-0-test(w, n)
1. if  $n = 0$  return true
2.  $w[n-1] \leftarrow \text{random}(2)$  // this is the only change
3. if  $w[n-1] = 1$  return false
4. randomized-all-0-test(w,  $n-1$ )
```

This tests whether a randomly chosen bitstring is all-0.

(Input w is actually irrelevant.)

Goal: Develop a recursive formula for $T^{\text{exp}}(n)$.

(This is mostly a “warm-up” for later.)

In each recursion, we use the outcome $x \in \{0, 1\}$ of one coin toss.

We return without recursing if $x = 1$ (this has probability $\frac{1}{2}$).

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

Recall: $T^{\text{exp}}(n) = \max_w T^{\text{exp}}(w) = \max_w \sum_R \mathbb{P}(R) T(w, R)$.

Use here: $T(w, R) = \#$ of bit-comparisons used on input w if the random outcomes are R .

- Recursive formula for one instance:

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

where $R' =$ rest of outcomes.

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

$$T^{\text{exp}}(n) = \underbrace{\frac{1}{2}}_{\mathbb{P}(x=1)} \cdot 1 + \underbrace{\frac{1}{2}}_{\mathbb{P}(x=0)} (1 + T^{\text{exp}}(n-1)) = 1 + \frac{1}{2} T^{\text{exp}}(n-1)$$

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*:

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

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 $\Rightarrow T_{\text{rand-all-0-test}}^{\text{exp}}(n) \in O(1)$.

Recall: *randomized-all-0-test* was very similar to *all-0-test*

(The only difference was to change the input randomly.)

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).

Average-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:

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

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

Average-case run-time via expected run-time

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

shuffle-all-0-test(n)

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

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

1. **for** ($i \leftarrow n-1; i \geq 0; i--$) **do**
2. $w[i] \leftarrow \text{random}(2)$
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, the compared bit is 1 with probability $\frac{1}{2}$.
- So $T_{\text{all-0-test}}^{\text{avg}}(n) = T_{\text{shuffle-all-0-test}}^{\text{exp}}(n) = T_{\text{rand-all-0-test}}^{\text{exp}}(n) \in O(1)$
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?

No!

average-case run-time	expected run-time
$\frac{1}{ \mathcal{I}_n } \sum_{I \in \mathcal{I}_n} T(I)$	$\max_{I \in \mathcal{I}_n} \sum_{\text{outcomes } R} \mathbb{P}(R) \cdot T(I, R)$
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”.

The SELECTION problem

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

(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.

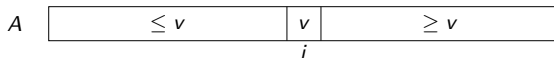
Question: Can we do selection in linear time?

Yes! We will develop algorithm *quick-select* below.

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, n, p): Rearrange A and return **pivot-rank** i so that



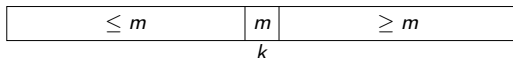
Easy to implement so that it uses at most n key-comparisons.

- ▶ Create three lists *smaller*, *equal*, *larger*
- ▶ Scan through A and add items to appropriate list
- ▶ Copy back from lists to A suitably

We will consider more sophisticated ideas later on.

Algorithm *quick-select*

Goal: Find element m of rank k by rearranging A :



Recall: *partition* method achieves



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

quick-select(A, n, k)

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

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

Analysis of *quick-select*

Let $T(A, k)$ be the number of key-comparisons for *quick-select*(A, k).
partition uses n key-comparisons.

Write A' for rearranged A after *partition*, and i for the pivot-rank.

$$T(A, k) = \begin{cases} n & \text{if } i = k \\ n + T(A'[0..i-1], k) & \text{if } i > k \quad (\text{sub-array has size } i) \\ n + T(A'[i+1..n-1], k-i-1) & \text{if } i < k \quad (\dots \text{size } n-i-1) \end{cases}$$

Worst-case run-time:

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

Best-case run-time: $\Theta(n)$ if $i = k$ in first round.

Average case run-time?

Average-case analysis of quick-select (sketch)

$$T(A, k) = \begin{cases} n & \text{if } i = k \\ n + T(A'[0..i-1], k) & \text{if } i > k \text{ (sub-array has size } i) \\ n + T(A'[i+1..n-1], k-i-1) & \text{if } i < k \text{ (... size } n-i-1) \end{cases}$$

To obtain “obvious” recursive formula:

- Argue: $\frac{1}{n}$ th of the inputs have pivot-rank i
(Easy if we assume that instances are permutations.)
- Argue: the rank-index k does not matter for analysis.
(Not obvious, but analysis works even if we take $\max_k$.)
- Argue: If A is “average”, then A' is also “average”. *Difficult!*
(
 - ▶ Formally: Over all choices of A , we have equally many occurrences of each possibility of A' .
 - ▶ False for some implementations of *partition*. Correct if *partition* only compares to the pivot-value.)

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

Randomizing *quick-select*

To avoid the difficult proof, use randomization instead.

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: Analyze run-time on randomly chosen input, i.e.,
choose n numbers randomly, then do *quick-select*.

This has some problems:

- From what range should we choose the numbers?
- This does not solve the original problem.

We can do something smarter, but need a brief detour.

Sorting permutations

- Make an **assumption**:

*All input numbers are **distinct**.*

(For most problems, this can be forced by using tie-breakers.)

- **Observe:** *quick-select* is **comparison-based**: Data accessed only by
 - ▶ comparing two elements (a *key-comparison*)
 - ▶ moving elements around (e.g. copying, swapping)
- **Observe:** Any comparison-based algorithm has the same run-time on inputs
$$A = [14, 3, 2, 6, 1, 11, 7] \text{ and}$$
$$A' = [14, 4, 2, 6, 1, 12, 8]$$
- The actual numbers do not matter, only their *relative order*.

Sorting permutations

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

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

Example: $A = \begin{bmatrix} 14, & 3, & 2, & 6, & 1, & 11, & 7 \end{bmatrix}$
 $\pi = \begin{bmatrix} 4, & 2, & 1, & 3, & 6, & 5, & 0 \end{bmatrix}$

- Make another **assumption**:
*All $n!$ sorting permutations are **equally frequent** among inputs.*
- Then choosing the input randomly
= change input so that its sorting permutation is randomly chosen
= shuffle the input-array A randomly

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

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

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

(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 .

- 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: $\mathbb{P}(\text{pivot-rank is } i) = \frac{1}{n}$
 - We recurse in an array of size i or $n-i-1$ (or not at all)
- Recursive formula for one instance (and fixed $R = \langle i, R' \rangle$):

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

Analysis of *randomized-quick-select*

Since the expected run-time uses the *worst-case instance*, the recursive formula can now be shown easily:

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

tedious but straightforward

Analysis of *randomized-quick-select*

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

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

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

Proof continued...

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. ($\rightarrow$ cs341, maybe)

Randomizing quick-sort

We analyze the avg-case run-time of *quick-sort* again via randomization.

randomized-quick-sort(A, n)

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

- We use n comparisons in *partition*.
- $\mathbb{P}(\text{pivot has rank } i) = \frac{1}{n}$
- We recurse in two arrays, of size i and $n-i-1$

This implies

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

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:

Proof continued

quick-sort: Summary

- *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 $\Omega(n^2)$
- This implies (with the same detour through *shuffle-quick-sort*):

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).

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. $(\ell, r) \leftarrow S.pop()$ // avoid recursions
4. while $(r - \ell + 1 > 10)$ do // stop recursions early
5. $p \leftarrow \ell + random(\ell - r + 1)$
6. $i \leftarrow Hoare-partition(A, \ell, r, p)$ // use better routine
7. if $(i - \ell > r - i)$ do // reduce aux. space
8. $S.push((\ell, i - 1))$
9. $\ell \leftarrow i + 1$ // remove tail-recursion
10. else
11. $S.push((i + 1, r))$
12. $r \leftarrow i - 1$
13. insertion-sort(A)

```

# Hoare's partition

- *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  | 70  | 80  | 90   |

# Hoare's in-place partition routine

Loop invariant:



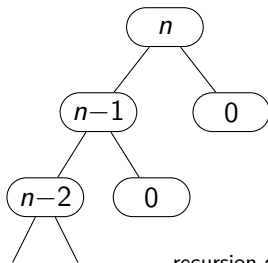
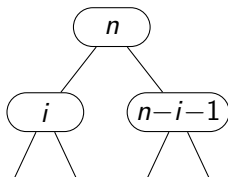
*Hoare-partition*( $A, \ell, r, p$ )

A: array whose index-range includes  $\ell < r$ ,  $p$ : integer s.t.  $\ell \leq p \leq r$

1. *swap*( $A[r], A[p]$ )
2.  $i \leftarrow \ell - 1$ ,  $j \leftarrow r$ ,  $v \leftarrow A[r]$
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*( $A[r], A[i]$ )
10. **return**  $i$

## quick-sort: Improvement ideas

- Every recursive call uses  $O(1)$  auxiliary space to store a record.
- *quick-sort* has nested recursive calls. To analyze its auxiliary space, analyze the height of the recursion tree.



recursion depth can be  $\Omega(n)$

- 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.

## quick-sort: Auxiliary space

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

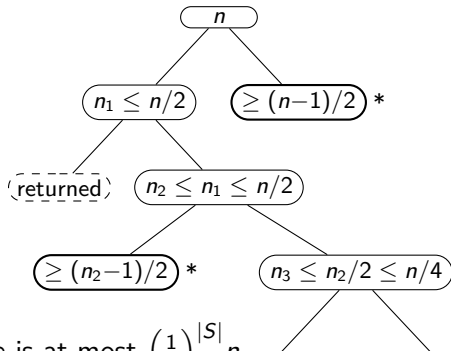
**Proof:** Consider the path in the recursion tree to the current subproblem.

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,  $|S| \leq \log n$ .



## 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>        | $\Theta(n^2)$<br>$\Theta(n)$ | worst-case<br>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

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.**

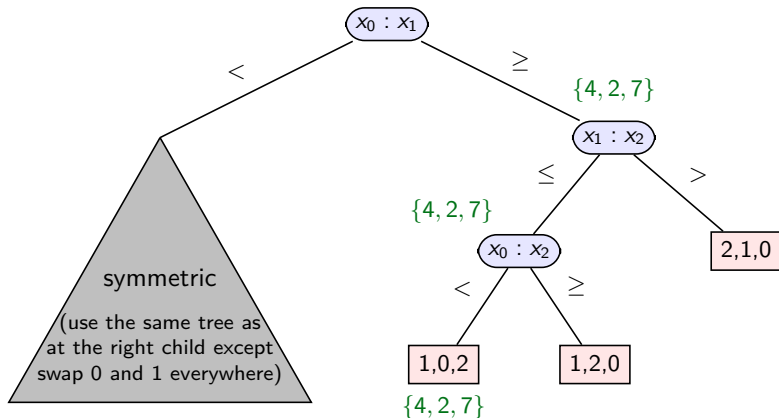
# Proof continued

# Decision trees

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

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

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



Output:  $\{4, 2, 7\}$  has sorting permutation  $\langle 1, 0, 2 \rangle$   
(i.e.,  $x_1=2 \leq x_0=4 \leq x_2=7$ )

# 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  $w$  of digits.
  - ▶ Can achieve after padding with leading 0s.
  - ▶ In typical computers,  $w = 32$  or  $w = 64$ , but  $w$  need not be constant

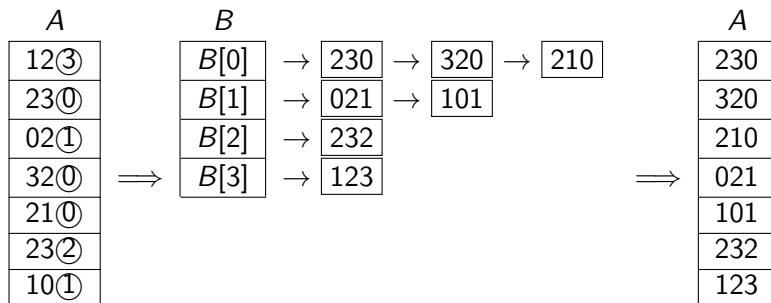
Example ( $R = 4$ ):

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| 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:



## (Single-digit) *bucket-sort*

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

$A$ : array of size  $n$

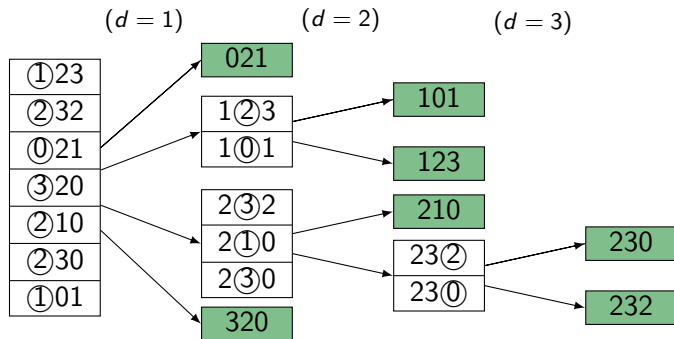
*sort-key*( $\cdot$ ) : maps items of  $A$  to  $\{0, \dots, R-1\}$

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



# MSD-radix-sort

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

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

1. **if** ( $d \leq w$  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:

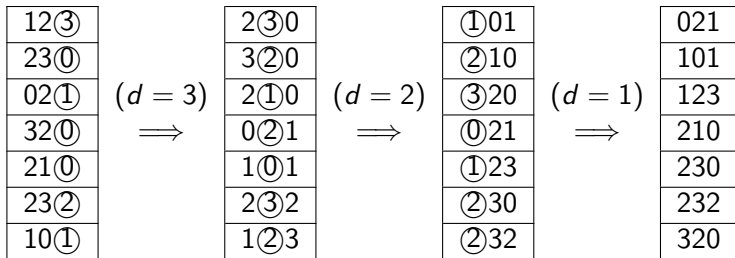
- $\Theta(w)$  levels of recursion in worst-case.
  - $\Theta(n)$  subproblems on most levels in worst-case.
  - $\Theta(R + (\text{size of sub-array}))$  time for each *bucket-sort* call.
- $\Rightarrow$  Run-time  $\Theta(wnR)$  — slow. Many recursions and allocated arrays.

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



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