CS 240 – Data Structures and Data Management

Module 3: Sorting, Average-case and Randomization

Armin Jamshidpey, Éric Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2025

version 2025−05−29 14:31

Outline

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

Outline

Average-case analysis

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

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*

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

First example: finding zero

A simple algorithm

- input: an array $A[0..n-1]$ with all 1 entries, except for one 0
- output: the position $i \in \{0, \ldots, n-1\}$ of the 0
- $\mathcal{I}_n = \{A_0, \ldots, A_{n-1}\}$: n possible arrays of size n, with A_j having 0 at position j

find_zero$(A,\ n)$
1. **for** $i \leftarrow 0$ to $n-1$ **do**
2.   **if** $A[i] = 0$ **then**
3.   **return** i

Set $T(A)$ = number of comparisons (step 2) on input A

- **worst-case**: $A[n-1] = 0$, we visit the whole array: $T(A) = n$
- **best-case**: $A[0] = 0$: $T(A) = 1$
- **average-case?**

A simple algorithm

- input: an array $A[0..n-1]$ with all 1 entries, except for one 0
- output: the position $i \in \{0, \ldots, n-1\}$ of the 0
- $\mathcal{I}_n = \{A_0, \ldots, A_{n-1}\}$: n possible arrays of size n, with A_j having 0 at position j

```
find_zero(A, n)
    for i ← 0 to n − 1 do
        if A[i] = 0 then
            return i
```

$$T^{\text{avg}}(n) = \frac{1}{n} \sum_{A \in \mathcal{I}_n} T(A) = \frac{1}{n} \sum_{j=0}^{n-1} T(A_j)$$

$$= \frac{1}{n} \sum_{j=0}^{n-1} (j+1) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n}{2} + \Theta(1)$$

Recursive version

find_zero_rec(A, n)
1. **if** $A[0] = 0$ **then return** 0
2. **else return** $1+$*find_zero_rec*$(A[1..n-1],\ n-1)$

Set $T_{\rm rec}(A)$ = number of comparisons (step 1) on input A:

$$T_{\rm rec}(A) \quad = \quad \begin{cases} 1 & \text{if } A[0] = 0 \\ 1 + T_{\rm rec}(\underbrace{A[1..n{-}1]}_{\text{length } n-1}) & \text{otherwise} \end{cases}$$

Recursive version

```
find_zero_rec(A, n)
1.  if A[0] = 0 then return 0
2.  else return 1+find_zero_rec(A[1..n − 1], n − 1)
```

Set $T_{\mathrm{rec}}(A)$ = number of comparisons (step 1) on input A:

$$T_{\mathrm{rec}}(A) = \begin{cases} 1 & \text{if } A[0] = 0 \\ 1 + T_{\mathrm{rec}}(\underbrace{A[1..n-1]}_{\text{length } n-1}) & \text{otherwise} \end{cases}$$

Recurrence:

$$T_{\mathrm{rec}}^{\mathrm{avg}}(n) = \frac{1}{n} \sum_{A \in \mathcal{I}_n} T_{\mathrm{rec}}(A) = \frac{1}{n} \left(1 + \sum_{A \in \mathcal{I}_n, A[0]!=0} (1 + T_{\mathrm{rec}}(A[1..n-1])) \right)$$

$$= 1 + \frac{1}{n} \sum_{A \in \mathcal{I}_n, A[0]!=0} T_{\mathrm{rec}}(A[1..n-1]) = 1 + \frac{n-1}{n} T_{\mathrm{rec}}^{\mathrm{avg}}(n-1)$$

Recursive version

Solving the recurrence:

$$n T^{\text{avg}}_{\text{rec}}(n) = n + (n-1)\, T^{\text{avg}}_{\text{rec}}(n-1)$$

Set $U_n = n T^{\text{avg}}_{\text{rec}}(n)$:

$$U_n = n + U_{n-1}, \quad U_0 = 0$$

so $U_n = 1 + \cdots + n = n(n+1)/2$ and

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

Conclusion:

- same result as the iterative version
- but we had to do some work to obtain a recurrence relation on the average runtime

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - Randomized Algorithms
   - SELECTION and *quick-select*
   - 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.

(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!)

```
randomized-silly(A, n)
A: array of size n
    sum ← 0
    if (random(3)>0) then
        for i ← 0 to n − 1 do
            sum ← sum + A[i]
    return sum
```

Randomized algorithms

- A **randomized algorithm** is one which relies on some random numbers in addition to the input.

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

```
randomized-silly(A, n)
A: array of size n
    sum ← 0
    if (random(3)>0) then
        for i ← 0 to n − 1 do
            sum ← sum + A[i]
    return sum
```

- We assume the existence of a function *random*(n) that returns an integer uniformly from $\{0, 1, 2, \ldots, n-1\}$. So $Pr(random(3)=0) = \frac{1}{3}$.

Expected run-time

The run-time depends on the random numbers, as well as the input.

Definition: $T_{\mathcal{A}}(I, R)$ is the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.
(in the example, R is either 0, 1 or 2)

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

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

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

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

We can also (rarely) discuss the very worst that could happen:
$\max_I \max_R T(I, R)$.

Expected run-time: Example

```
randomized-silly(A, n)
A: array of size n
    sum ← 0
    if (random(3)>0) then
        for i ← 0 to n − 1 do
            sum ← sum + A[i]
    return sum
```

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

- $R \in \{0,1,2\}$ with $Pr(0) = Pr(1) = Pr(2) = \frac{1}{3}$
- If outcome is 0: $O(1)$ time, say c time units (for some constant c)
- If outcome is 1 or 2: $O(n)$ time, say cn time units
- $T^{\exp}(A) = \frac{1}{3}c + \frac{1}{3}cn + \frac{1}{3}cn \in \Theta(n)$
- All instances have the same expected run-time, so $T^{\exp}(n) \in \Theta(n)$

Why randomized algorithms?

- Doing randomization is often a good idea if an algorithm has bad worst-case time but seems to perform much better on most instances.
- **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.
- Not all randomizations achieve this automatically; it must be proved.

Randomizations of algorithms

$randomized\_find\_rec(A, n)$

1. $r \leftarrow random(n)$
2. **if** $A[r] = 0$ **then return** r
3. **swap** $A[r]$ and $A[n-1]$
4. $p \leftarrow randomized\_find\_rec(A[0..n-2], n-1)$
5. **if** $p = r$ **then return** $n-1$ **else return** p

- $T(A, R) = \#$ of comparisons on input A and random outcomes R
- random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first random choice $\in \{r = 0, r = 1, \ldots, r = n-1\}$
  - R': random outcomes (if any) for the recursions
- $Pr(R) = Pr(x)\, Pr(R')$ (random choices are independent).
- recurrence (we let k be the position of 0 in A)

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

where R' = rest of outcomes and A' = array after swap.

Expected run-time of *randomized_find_rec*

$$
\begin{aligned}
T^{\exp}(A) &= \sum_R T(A,R)\ Pr(R) \\
&= 1 \cdot Pr(r = k) \\
&\quad + \sum_{j \neq k} \sum_{R'} (1 + T(A'[0..n-2], R'))\ Pr(r = j)\ Pr(R')
\end{aligned}
$$

Expected run-time of *randomized_find_rec*

$$
\begin{aligned}
T^{\exp}(A) &= \sum_{R} T(A,R)\ Pr(R) \\
&= 1 \cdot Pr(r = k) \\
&\quad + \sum_{j \neq k} \sum_{R'} (1 + T(A'[0..n-2], R'))\ Pr(r = j)\ Pr(R') \\
&= \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} \left(\sum_{R'} Pr(R') + \sum_{R'} T(A'[0..n-2], R')\ Pr(R') \right)
\end{aligned}
$$

Expected run-time of *randomized_find_rec*

$$
\begin{aligned}
T^{\exp}(A) &= \sum_{R} T(A,R)\ Pr(R) \\
&= 1 \cdot Pr(r=k) \\
&\quad + \sum_{j \neq k} \sum_{R'} (1 + T(A'[0..n-2], R'))\ Pr(r=j)\ Pr(R') \\
&= \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} \left(\sum_{R'} Pr(R') + \sum_{R'} T(A'[0..n-2], R')\ Pr(R') \right) \\
&= \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} (1 + T^{\exp}(A'[0..n-2]))
\end{aligned}
$$

Expected run-time of *randomized_find_rec*

$$
\begin{aligned}
T^{\exp}(A) &= \sum_R T(A,R)\ Pr(R) \\
&= 1 \cdot Pr(r=k) \\
&\quad + \sum_{j \neq k} \sum_{R'} (1 + T(A'[0..n-2], R'))\ Pr(r=j)\ Pr(R') \\
&= \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} \left(\sum_{R'} Pr(R') + \sum_{R'} T(A'[0..n-2], R')\ Pr(R') \right) \\
&= \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} (1 + T^{\exp}(A'[0..n-2])) \\
&\leq \frac{1}{n} + \frac{1}{n} \sum_{j \neq k} (1 + T^{\exp}(n-1)) = 1 + \frac{n-1}{n} T^{\exp}(n-1)
\end{aligned}
$$

True for **any** A: $T^{\exp}(n) \leq 1 + \frac{n-1}{n} T^{\exp}(n-1)$ and so $T^{\exp}(n) \leq \frac{n+1}{2}$.

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

Are average-case run-time and expected run-time the same?

No!

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

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

Outline

3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - Randomized Algorithms
   - SELECTION and *quick-select*
   - 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 $0 \le k < n$, 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 (but we won't do the worst-case linear time version)

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

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

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

  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

  Note: can be done in place in one pass over A

Algorithm *quick-select*

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

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

Recall: *partition* method achieves

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

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

Analysis of *quick-select*

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

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

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

Analysis of *quick-select*

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

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

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

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

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

Average analysis?

What do we average on? Infinitely many arrays with e.g. integer entries...

- 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

$$\begin{array}{llllllllll} A = [& 14, & 3, & 2, & 6, & 1, & 11, & 7 &] & \text{and} \\ A' = [& 14, & 4, & 2, & 6, & 1, & 12, & 8 &] & \end{array}$$

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

Sorting permutations

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

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

($\pi[0] =$ index of the smallest element, then $\pi[1], \ldots$)

Example:

$$\begin{array}{rl} A = [& 14, \quad 3, \quad 2, \quad 6, \quad 1, \quad 11, \quad 7 \quad] \\ \pi = [& 4, \quad 2, \quad 1, \quad 3, \quad 6, \quad 5, \quad 0 \quad] \end{array}$$

Sorting permutations

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

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

  ($\pi[0]$ = index of the smallest element, then $\pi[1], \ldots$)

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

- Make another **assumption**:
  All $n!$ sorting permutations are equally frequent among inputs.

- Then average run-time is

  $$T^{\text{avg}}(n) = \frac{1}{n!}\frac{1}{n} \sum_{A \text{ permutation } of \{0,\ldots,n-1\}} \sum_k T(A,k)$$

  We will estimate $T^{\text{avg}}(n)$ via randomization.

Randomizing quick-select

First idea: shuffle the array at random (does not change the result)

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

Second idea: do the shuffling inside the recursion

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

Difficult: $T^{\text{avg}}_{\textit{quick-select}}(n) \leq T^{\text{exp}}_{\textit{shuffle-quick-select}}(n) = T^{\text{exp}}_{\textit{rand.-quick-select}}(n)$

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

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

Analysis of *rand.-quick-select* (do not read)

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

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

independent of A, k

Analysis of *randomized-quick-select*

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

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

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

Summary of SELECTION

- *randomized-quick-select* has expected run-time $\Theta(n)$
  - The run-time bound is tight since *partition* takes $\Omega(n)$ time
  - If we're unlucky in the random numbers then the run-time is still $\Omega(n^2)$
- So the expected run-time of *shuffle-quick-select* is also $\Theta(n)$
- So the average-case run-time of *quick-select* is $\Theta(n)$
- *randomized-quick-select* is generally the fastest solution to SELECTION
- There exists a variation that solves SELECTION with worst-case run-time $\Theta(n)$, but it uses double recursion and is slower in practice ($\rightarrow$ *cs*341)

Outline

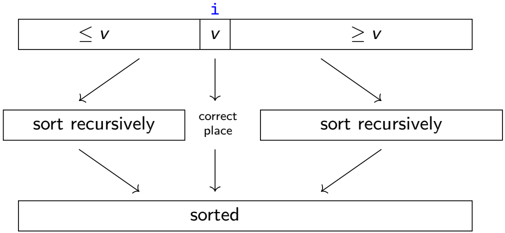
3. Sorting, Average-case and Randomization
   - Analyzing average-case run-time
   - Randomized Algorithms
   - SELECTION and *quick-select*
   - SORTING and *quick-sort*
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Algorithm *quick-sort*

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

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

> *quick-sort*(A, n)
> A: array of size n
> **if** $n \leq 1$ **then return**
> $p \leftarrow$ *choose-pivot*(A)
> $i \leftarrow$ *partition*(A, n, p)
> *quick-sort*$(A[0, 1, \ldots, i{-}1], i)$
> *quick-sort*$(A[i{+}1, \ldots, n{-}1], n{-}i{-}1)$



Analysis of *quick-sort*

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*

Analysis of *quick-sort*

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: $\Theta(n \log n)$

- If pivot-rank 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.

Analysis of *quick-sort*

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: $\Theta(n \log n)$

- If pivot-rank 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: by randomization

Randomizing quick-sort

randomized-quick-sort(A, n)

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

$i \leftarrow$ *partition*$(A, n,$ *random*$(n))$

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

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

Randomizing quick-sort

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

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

Randomizing quick-sort

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

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

This implies

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

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

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

(since $T(0) = 0$)

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

Proof:

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 $\Theta(n \log n)$.

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 $\Theta(n \log n)$.

- 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 $\Theta(\log n)$)

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

Outline

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

Lower bounds for sorting

We have seen many sorting algorithms:

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

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

Lower bounds for sorting

We have seen many sorting algorithms:

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

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

- 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)

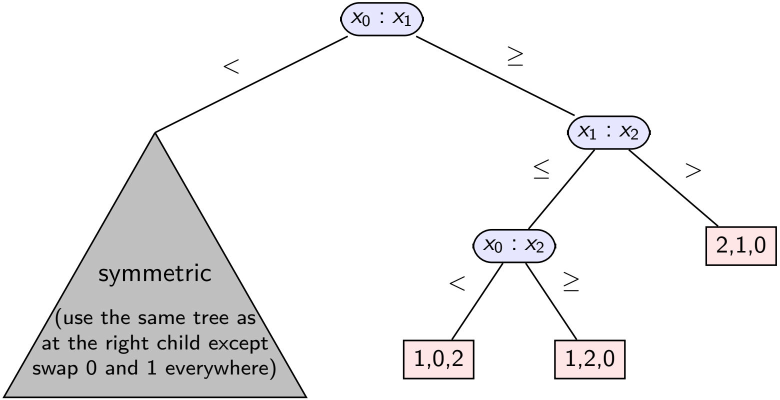
Theorem. Any *comparison-based* sorting algorithm requires in the worst case $\Omega(n \log n)$ comparisons to sort n distinct items.

Proof.

Decision trees

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

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

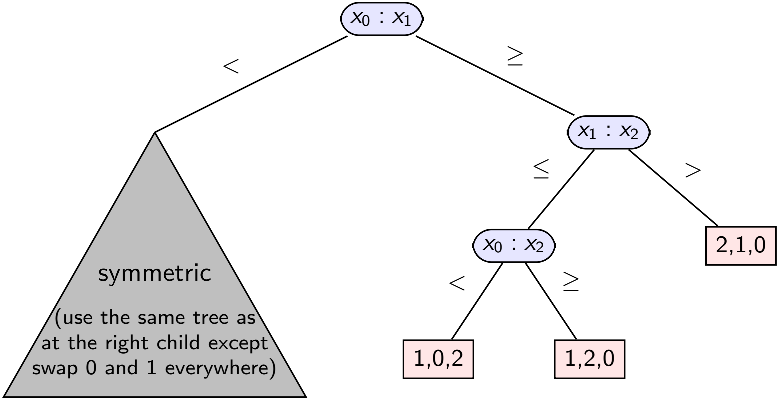


Decision trees

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

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

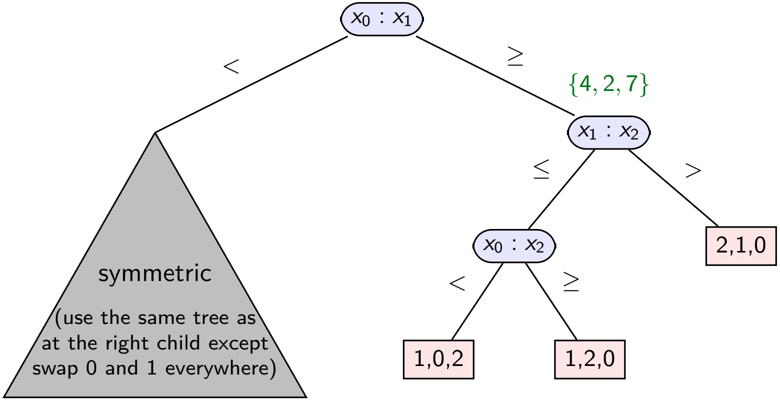
Example: $\{x_0{=}4, x_1{=}2, x_2{=}7\}$



Decision trees

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

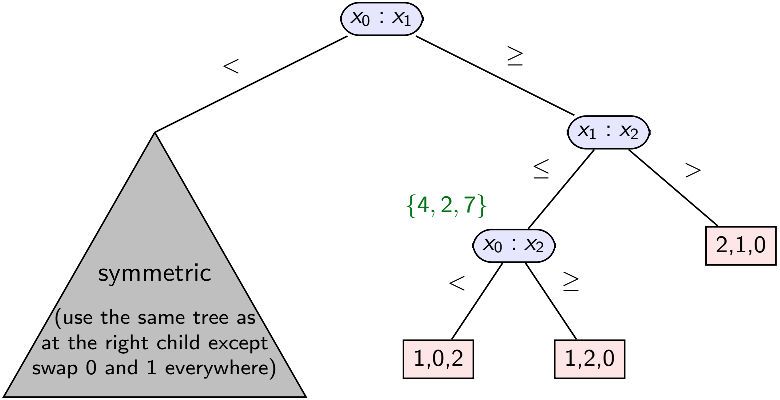
To sort $\{x_0, x_1, x_2\}$:



Decision trees

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

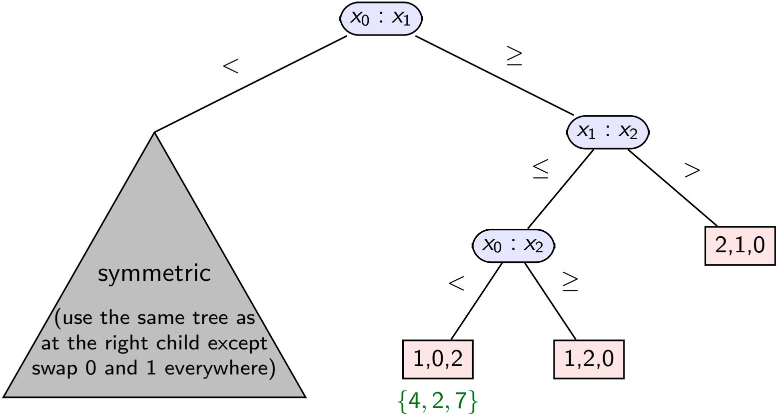
To sort $\{x_0, x_1, x_2\}$:



Decision trees

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

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



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

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

Outline

Non-comparison-based sorting

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

Example ($R = 4$):

123	230	21	320	210	232	101

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

Example ($R = 4$):

123	230	021	320	210	232	101

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

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A

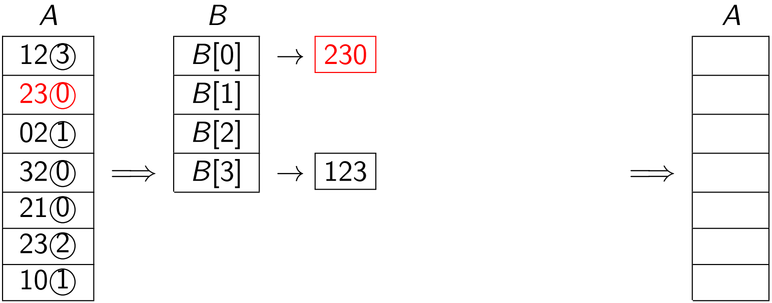
(Single-digit) *bucket-sort*

Sort array A by last digit:

A		B				A
123		$B[0]$				
230		$B[1]$				
021		$B[2]$				
320	$\Longrightarrow$	$B[3]$	$\rightarrow$	$\boxed{123}$	$\Longrightarrow$	
210						
232						
101						

(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

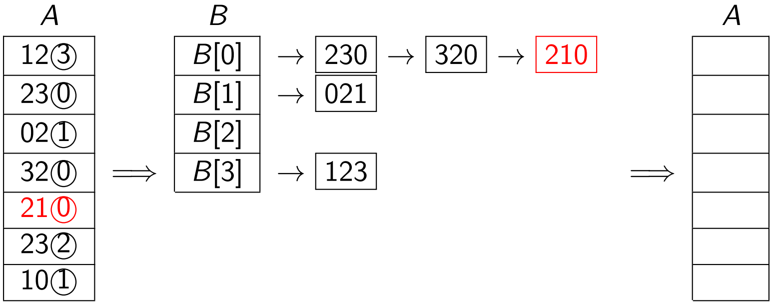
B				
$B[0]$	$\rightarrow$	230	$\rightarrow$	320
$B[1]$	$\rightarrow$	021		
$B[2]$				
$B[3]$	$\rightarrow$	123		

$\Longrightarrow$

A

(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

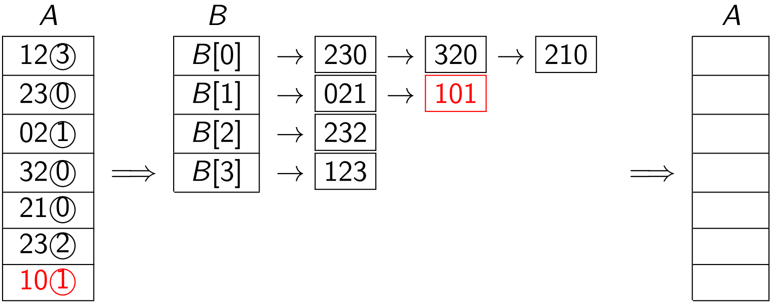
B	
$B[0]$	$\rightarrow$ 230 $\rightarrow$ 320 $\rightarrow$ 210
$B[1]$	$\rightarrow$ 021
$B[2]$	$\rightarrow$ 232
$B[3]$	$\rightarrow$ 123

$\Longrightarrow$

A

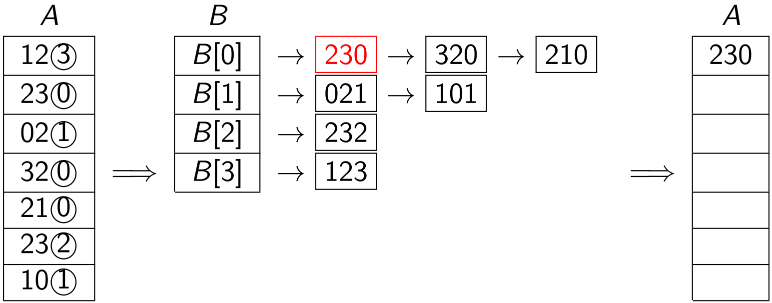
(Single-digit) *bucket-sort*

Sort array A by last digit:



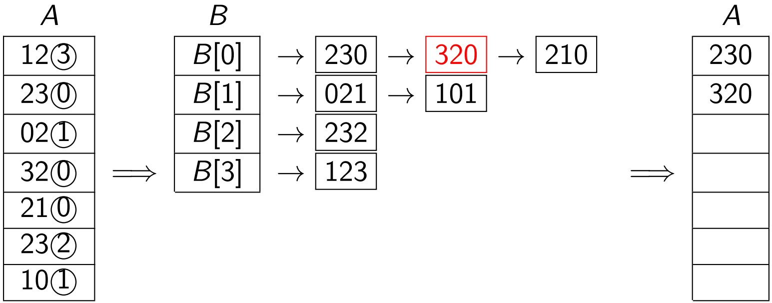
(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

Sort array A by last digit:



(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A
230
320
210

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A
230
320
210
021

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A
230
320
210
021
101

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A
230
320
210
021
101
232

(Single-digit) *bucket-sort*

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

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

$\Longrightarrow$

A
230
320
210
021
101
232
123

(Single-digit) *bucket-sort*

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

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

- In our example *sort-key*($A[i]$) returns the last digit of $A[i]$
- *bucket-sort* is **stable**: equal items stay in original order.
- Run-time $\Theta(n + R)$, auxiliary space $\Theta(n + R)$
- It is possible to replace the lists by arrays $\rightsquigarrow$ *count-sort* (no details).

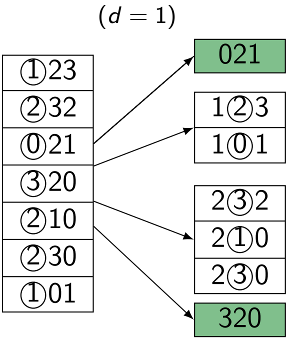
Most-significant-digit(MSD)-radix-sort

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

123
232
021
320
210
230
101

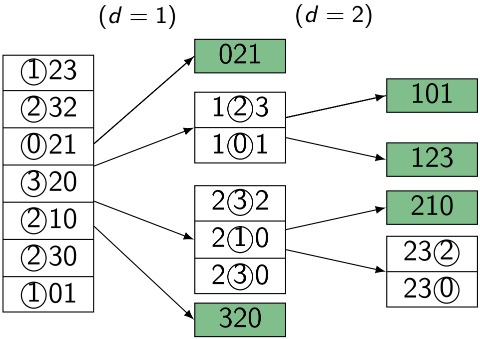
Most-significant-digit(MSD)-radix-sort

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



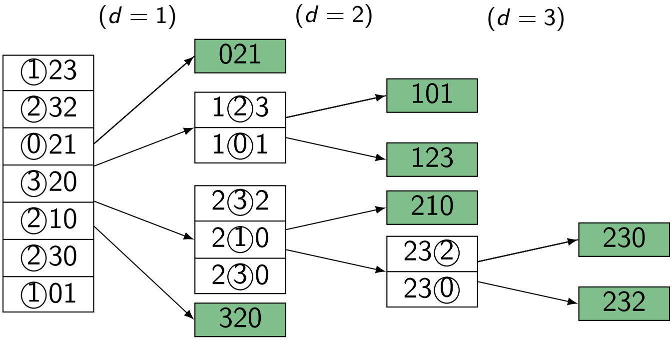
Most-significant-digit(MSD)-radix-sort

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



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

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



MSD-radix-sort

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

Analysis:

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

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

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

LSD-radix-sort(A, n)
A: array of size n, contains m-digit radix-R numbers

```
for d ← least significant to most significant digit do
        bucket-sort(A, n, 'return dth digit of A[i]')
```

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

- Loop-invariant: A is sorted w.r.t. digits $d, \ldots, w$ of each entry.
- **Time cost**: $\Theta(w(n+R))$ **Auxiliary space**: $\Theta(n+R)$

Summary

- SORTING is an important and *very* well-studied problem
- Can be done in $\Theta(n \log n)$ time; faster is not possible for general input
- *heap-sort* is the only $\Theta(n \log n)$-time algorithm we have seen with $O(1)$ auxiliary space.
- *merge-sort* is also $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.
- *quick-sort* is worst-case $\Theta(n^2)$, but often the fastest in practice
- *bucket-sort* and *radix-sort* achieve $o(n \log n)$ if the input is special
- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case can all differ.
- Often it is easier to analyze the run-time on randomly chosen input rather than the average-case run-time.