CS 240 – Data Structures and Data Management

Module 3: Sorting and Randomized Algorithms

Mark Petrick Olga Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2020

References: Sedgewick 6.10, 7.1, 7.2, 7.8, 10.3, 10.5
Goodrich & Tamassia 8.3

version 2020-01-21 10:06

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Selection vs. Sorting

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

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

select(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?

Selection vs. Sorting

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

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

select(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?
The *quick-select* algorithm answers this question in the affirmative.

Selection vs. Sorting

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

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

select(3) should return 30.

Special case: **median finding** = selection with $k = \lfloor \frac{n}{2} \rfloor$.

Selection can be done with heaps in time $\Theta(n + k \log n)$.
Median-finding with this takes time $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?
The *quick-select* algorithm answers this question in the affirmative.

The encountered sub-routines will also be useful otherwise.

Crucial Subroutines

quick-select and the related algorithm *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 algorithm *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.

  Simplest idea: Always select rightmost element in array

  choose-pivot1(A)
  1. **return** $A.size - 1$

  We will consider more sophisticated ideas later on.

Crucial Subroutines

quick-select and the related algorithm *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.

  Simplest idea: Always select rightmost element in array

  ```
  choose-pivot1(A)
  1.   return A.size − 1
  ```


  We will consider more sophisticated ideas later on.

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

  - the pivot-value v is in $A[i]$,
  - all items in $A[0, \ldots, i-1]$ are $\leq v$, and
  - all items in $A[i+1, \ldots, n-1]$ are $\geq v$.

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

Partition Algorithm

Conceptually easy linear-time implementation:

partition(A, p)
A: array of size n, p: integer s.t. $0 \le p < n$

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

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

Efficient In-Place partition (Hoare)

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

Efficient In-Place partition (Hoare)

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

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

Efficient In-Place partition (Hoare)

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

0	1	2	3	4	i=5	6	7	j=[illegible]	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)

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)

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)

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)

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)

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

A	$\leq v$	?	$\geq v$	v
		i	j	

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

Running time: $\Theta(n)$.

QuickSelect Algorithm

quick-select1(A, k)
A: array of size n, k: integer s.t. $0 \leq k < n$

$p \leftarrow$ *choose-pivot1*(A)
$i \leftarrow$ *partition*(A, p)
if $i = k$ **then**
    return $A[i]$
else if $i > k$ **then**
    return *quick-select1*$(A[0, 1, \ldots, i-1], k)$
else if $i < k$ **then**
    return *quick-select1*$(A[i+1, i+2, \ldots, n-1], k-i-1)$

Analysis of *quick-select1*

Define $T(n)$ to be the run-time for selecting from size-n array, presuming we use *choose-pivot1*(A).

- $T(n)$ depends on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + \max\{T(i), T(n-i-1)\}$.

Analysis of *quick-select1*

Define $T(n)$ to be the run-time for selecting from size-n array, presuming we use *choose-pivot1*(A).

- $T(n)$ depends on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + \max\{T(i), T(n-i-1)\}$.
- **Worst-case analysis**: $i = 0$ or $n-1$ always. Then

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

for some constant $c > 0$. This resolves to $\Theta(n^2)$.

Analysis of *quick-select1*

Define $T(n)$ to be the run-time for selecting from size-n array, presuming we use *choose-pivot1*(A).

- $T(n)$ depends on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + \max\{T(i), T(n-i-1)\}$.
- **Worst-case analysis**: $i = 0$ or $n-1$ always. Then

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

  for some constant $c > 0$. This resolves to $\Theta(n^2)$.

- **Best-case analysis**: First pivot-value could be the desired element No recursive calls; total cost is $\Theta(n)$.

Analysis of *quick-select1*

Define $T(n)$ to be the run-time for selecting from size-n array, presuming we use *choose-pivot1*(A).

- $T(n)$ depends on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + \max\{T(i), T(n-i-1)\}$.
- **Worst-case analysis**: $i = 0$ or $n-1$ always. Then

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

  for some constant $c > 0$. This resolves to $\Theta(n^2)$.

- **Best-case analysis**: First pivot-value could be the desired element No recursive calls; total cost is $\Theta(n)$.
- Average-case?

Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size n?
  (There are infinitely many sets of n numbers.)

Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size n?
  (There are infinitely many sets of n numbers.)
- **Simplifying assumption:** All input numbers are *distinct*.
- Observe: quick-select1 would act the same on inputs
  14, 2, 3, 6, 1, 11, 7 and
  14, 2, 4, 6, 1, 12, 8
- The actual numbers do not matter, only their *relative order*.

Sorting Permutations

- Need to take average running time over all inputs.
- How to characterize input of size n?
  (There are infinitely many sets of n numbers.)

- **Simplifying assumption:** All input numbers are *distinct*.
- Observe: quick-select1 would act the same on inputs
  14, 2, 3, 6, 1, 11, 7 and
  14, 2, 4, 6, 1, 12, 8

- The actual numbers do not matter, only their *relative order*.
- Characterize input via **sorting permutation**: the permutation that would put the input in order.

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

$\rightsquigarrow$ Average cost is sum of costs for all permutations, divided by $n!$

Average-Case Analysis of *quick-select1*

- Define $T(n)$ to be the average cost for selecting from size-n array, presuming we use *choose-pivot1*(A).
- Fix one $0 \le i \le n-1$. There are $(n-1)!$ permutations for which the pivot-index is i.

$$
\begin{aligned}
T(n) &= \frac{1}{n!} \sum_{I:\text{size}(I)=n} \text{running time for instance } I \\
&= \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\substack{I:\textit{size}(I)=n \\ I \text{ has pivot-index } i}} \text{running time for instance } I \\
&\le \frac{1}{n!} \sum_{i=0}^{n-1} (n-1)! \quad (c \cdot n + \max\{T(i), T(n-i-1)\}) \\
&= c \cdot n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n-i-1)\}
\end{aligned}
$$

Average-Case Analysis of *quick-select1*

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

Theorem: $T(n) \in \Theta(n)$.
Proof:

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - 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!)

- The run-time will depend on the input and the random numbers used.
- **Goal:** Shift the dependency of run-time from what we can't control (the input) to what we *can* control (the random numbers).

No more bad instances, just unlucky numbers.

Expected running time

Define $T(I, R)$ to be the running time of the randomized algorithm for an instance I and the sequence of random numbers R.

The *expected running time* $T^{(\exp)}(I)$ for instance I is the expected value for $T(I, R)$:

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

Expected running time

Define $T(I, R)$ to be the running time of the randomized algorithm for an instance I and the sequence of random numbers R.

The *expected running time* $T^{(\exp)}(I)$ for instance I is the expected value for $T(I, R)$:

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

The *worst-case expected running time* is

$$T^{(\exp)}_{\text{worst}}(n) = \max_{\{I : size(I) = n\}} T^{(\exp)}(I).$$

Expected running time

Define $T(I, R)$ to be the running time of the randomized algorithm for an instance I and the sequence of random numbers R.

The *expected running time* $T^{(\exp)}(I)$ for instance I is the expected value for $T(I, R)$:

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

The *worst-case expected running time* is

$$T^{(\exp)}_{\text{worst}}(n) = \max_{\{I : size(I) = n\}} T^{(\exp)}(I).$$

The *average-case expected running time* is

$$T^{(\exp)}_{\text{avg}}(n) = \frac{1}{|\{I : size(I) = n\}|} \sum_{\{I : size(I) = n\}} T^{(\exp)}(I).$$

Randomized QuickSelect: Shuffle

Goal: Create a randomized version of *QuickSelect* for which all input has the same expected run-time. (Recall that we assume that all elements in A are distinct.)

First idea: Randomly permute the input first using *shuffle*:


```
shuffle(A)
A: array of size n
    for i ← 0 to n − 2 do
        swap( A[i], A[i + random(n − i)] )
```


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

Expected cost becomes the same as the average-case cost of *quick-select1*: $\Theta(n)$.

Randomized QuickSelect: Random Pivot

Second idea: Change the pivot selection.

```
choose-pivot2(A)
    return random(n)
```

```
quick-select2(A, k)
    p ← choose-pivot2(A)
    ...
```


With probablity $\frac{1}{n}$ the random pivot has index i, so the analysis is just like that for the average-case of *quick-select1*. The expected run-time is again $\Theta(n)$.

Randomized QuickSelect: Random Pivot

Second idea: Change the pivot selection.

```
choose-pivot2(A)
    return random(n)
```

```
quick-select2(A, k)
    p ← choose-pivot2(A)
    ...
```

With probablity $\frac{1}{n}$ the random pivot has index i, so the analysis is just like that for the average-case of *quick-select1*. The expected run-time is again $\Theta(n)$.

This is generally the fastest quick-select implementation.

There exists a variation that has worst-case running time $O(n)$, but it uses double recursion and is slower in practice. ($\rightsquigarrow$ *cs*341)

Outline

QuickSort

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

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

quick-sort1(A)
A: array of size n
if $n \leq 1$ **then return**
  $p \leftarrow$ *choose-pivot1*(A)
  $i \leftarrow$ *partition*(A, p)
  quick-sort1$(A[0, 1, \ldots, i-1])$
  quick-sort1$(A[i+1, \ldots, n-1])$

QuickSort analysis

Define $T(n)$ to be the run-time for *quick-sort1* in a size-n array.

- $T(n)$ depends again on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + T(i) + T(n-i-1)$.

QuickSort analysis

Define $T(n)$ to be the run-time for *quick-sort1* in a size-n array.

- $T(n)$ depends again on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + T(i) + T(n-i-1)$.

- **Worst-case analysis**: $i = 0$ or $n-1$ always. Then as for *quick-select*

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

for some constant $c > 0$. This resolves to $\Theta(n^2)$.

QuickSort analysis

Define $T(n)$ to be the run-time for *quick-sort1* in a size-n array.

- $T(n)$ depends again on the pivot-index i.
- If we know i: $T(n) = \Theta(n) + T(i) + T(n-i-1)$.

- **Worst-case analysis**: $i = 0$ or $n-1$ always. Then as for *quick-select*

$$T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ c, & n = 1 \end{cases}$$

  for some constant $c > 0$. This resolves to $\Theta(n^2)$.

- **Best-case analysis**: $i = \lfloor \frac{n}{2} \rfloor$ or $\lceil \frac{n}{2} \rceil$ always. Then

$$T(n) = \begin{cases} T(\lfloor \frac{n-1}{2} \rfloor) + T(\lceil \frac{n-1}{2} \rceil) + cn & n \geq 2 \\ c, & n = 1 \end{cases}$$

  Similar to *merge-sort*: This resolves to $\Theta(n \log n)$.

Average-case analysis of *quick-sort1*

Now let $T(n)$ to be the *average-case* run-time for *quick-sort1* in a size-n array.

- As before, $(n-1)!$ permutations have pivot-index i.
- So average running time is

$$
\begin{aligned}
T(n) &= \frac{1}{n!}\sum_{i=0}^{n-1} \sum_{\substack{I:size(I)=n \\ I \text{ has pivot-index } i}} \text{running time for instance } I \\
&\leq \frac{1}{n!}\sum_{i=0}^{n-1} (n-1)! \quad \Big(c\cdot n + T(i) + T(n-i-1)\Big) \\
&= c\cdot n + \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i)+T(n-i-1)\big)
\end{aligned}
$$

Average-case analysis of *quick-sort1*

Now let $T(n)$ to be the *average-case* run-time for *quick-sort1* in a size-n array.

- As before, $(n-1)!$ permutations have pivot-index i.
- So average running time is

$$T(n) = \frac{1}{n!}\sum_{i=0}^{n-1} \sum_{\substack{I:size(I)=n \\ I \text{ has pivot-index } i}} \text{running time for instance } I$$

$$\leq \frac{1}{n!}\sum_{i=0}^{n-1} (n-1)! \Big(c \cdot n + T(i) + T(n-i-1)\Big)$$

$$= c \cdot n + \frac{1}{n}\sum_{i=0}^{n-1} \big(T(i) + T(n-i-1)\big)$$

Theorem: $T(n) \in \Theta(n \log n)$.

Proof:

Improvement ideas for QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.

Improvement ideas for QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.
- The auxiliary space is Ω(recursion depth).
  - This is $\Theta(n)$ in the worst-case.
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.

Improvement ideas for QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.
- The auxiliary space is Ω(recursion depth).
  - This is $\Theta(n)$ in the worst-case.
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.
- One should stop recursing when $n \leq 10$.
  One run of InsertionSort at the end then sorts everything in $O(n)$ time since all items are within 10 units of their required position.

Improvement ideas for QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.
- The auxiliary space is Ω(recursion depth).
  - This is $\Theta(n)$ in the worst-case.
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.
- One should stop recursing when $n \leq 10$.
  One run of InsertionSort at the end then sorts everything in $O(n)$ time since all items are within 10 units of their required position.
- Arrays with many duplicates can be sorted faster by changing *partition* to produce three subsets

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

Improvement ideas for QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ *expected time* for *quick-sort2*.
- The auxiliary space is Ω(recursion depth).
  - This is $\Theta(n)$ in the worst-case.
  - It can be reduced to $\Theta(\log n)$ worst-case by recursing in smaller sub-array first and replacing the other recursion by a while-loop.
- One should stop recursing when $n \leq 10$.
  One run of InsertionSort at the end then sorts everything in $O(n)$ time since all items are within 10 units of their required position.
- Arrays with many duplicates can be sorted faster by changing *partition* to produce three subsets

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

- Two programming tricks that apply in many situations:
  - Instead of passing full arrays, pass only the range of indices.
  - Avoid recursion altogether by keeping an explicit stack.

QuickSort with tricks

quick-sort3(A, n)

Initialize a stack S of index-pairs with $\{(0, n-1)\}$
while S is not empty
    $(\ell, r) \leftarrow S.pop()$
    while $(r-\ell+1 > 10)$ **do**
        $p \leftarrow$ *choose-pivot2*(A, ℓ, r)
        $i \leftarrow$ *partition*(A, ℓ, r, p)
        if $(i-\ell > r-i)$ **do**
            $S.push((\ell, i-1))$
            $\ell \leftarrow i+1$
        else
            $S.push((i+1, r))$
            $r \leftarrow i-1$
InsertionSort(A)

This is often the most efficient sorting algorithm in practice.

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Lower bounds for sorting

We have seen many sorting algorithms:

Sort	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
quick-sort1	$\Theta(n \log n)$	average-case
quick-sort2	$\Theta(n \log n)$	expected, all cases

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

- No: Comparison-based sorting lower bound is $\Omega(n \log n)$.
- Yes: Non-comparison-based sorting can achieve $O(n)$ (under restrictions!). $\rightarrow$ see below

The Comparison Model

In the **comparison model** data can only be accessed in two ways:

- comparing two elements
- moving elements around (e.g. copying, swapping)

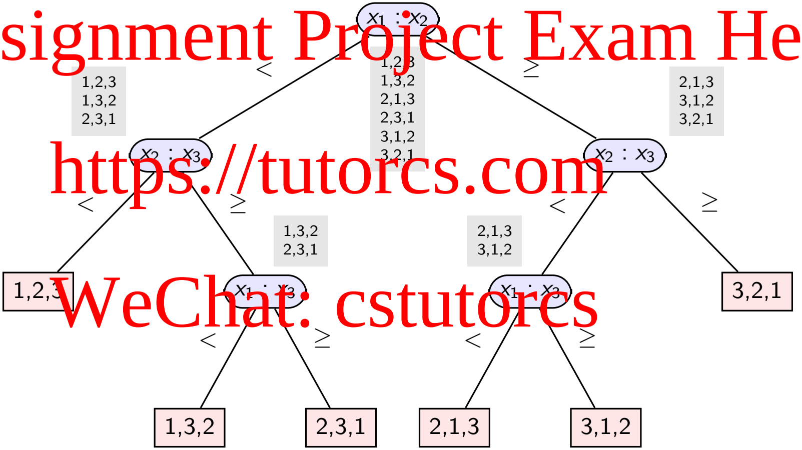
This makes very few assumptions on the kind of things we are sorting.
We count the number of above operations.

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

Decision trees

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

To sort $\{x_1, x_2, x_3\}$:



(i.e., $x_3 \le x_1 \le x_2$)

Lower bound for sorting in the comparison model

Theorem. Any correct *comparison-based* sorting algorithm requires at least $\Omega(n \log n)$ comparison operations to sort n distinct items.

Proof.

Outline

1. Sorting and Randomized Algorithms
   - QuickSelect
   - Randomized Algorithms
   - QuickSort
   - Lower Bound for Comparison-Based Sorting
   - Non-Comparison-Based Sorting

Non-Comparison-Based Sorting

- Assume keys are numbers in base R (R: **radix**)
  - $R = 2, 10, 128, 256$ are the most common.

Example ($R = 4$):

123	230	21	320	210	232	101

- Assume all keys have the same number m of digits.
  - Can achieve after padding with leading 0s.

Example ($R = 4$):

123	230	021	320	210	232	101

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

(Single-digit) Bucket Sort

Sort array A by last digit:

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
123
230
021
320
210
232
101

$\Longrightarrow$

B	
$B[0]$	
$B[1]$	
$B[2]$	
$B[3]$	$\rightarrow$ $\boxed{123}$

$\Longrightarrow$

A

(Single-digit) Bucket Sort

Sort array A by last digit:

A

123
230
021
320
210
232
101

$\Longrightarrow$

B

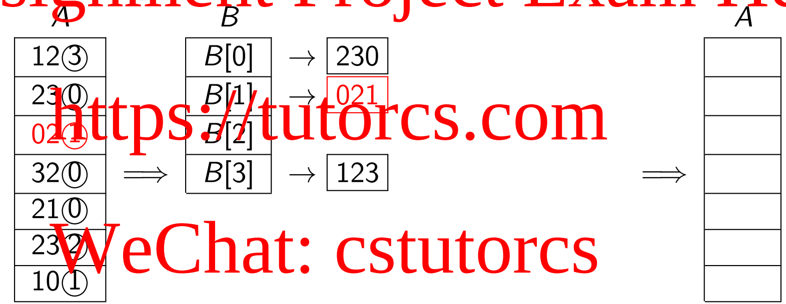
$B[0]$	$\rightarrow$ 230
$B[1]$	
$B[2]$	
$B[3]$	$\rightarrow$ 123

$\Longrightarrow$

A

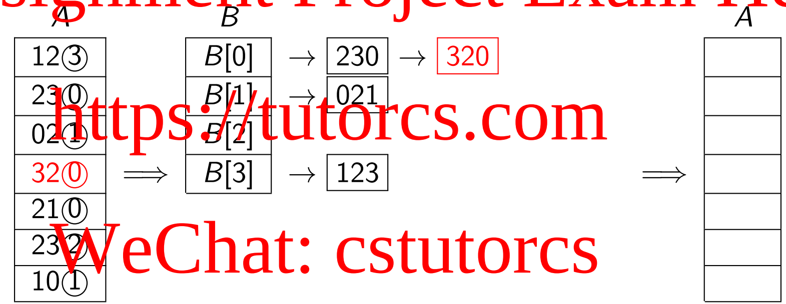
(Single-digit) Bucket Sort

Sort array A by last digit:



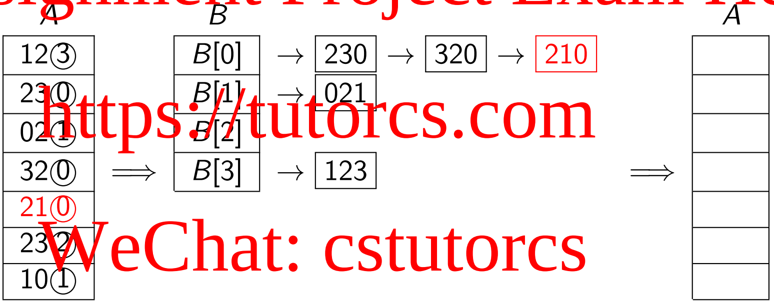
(Single-digit) Bucket Sort

Sort array A by last digit:



(Single-digit) Bucket Sort

Sort array A by last digit:



(Single-digit) Bucket Sort

Sort array A by last digit:

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

(Single-digit) Bucket Sort

Sort array A by last digit:

A:

123
230
021
320
210
232
101

$\Longrightarrow$

B:

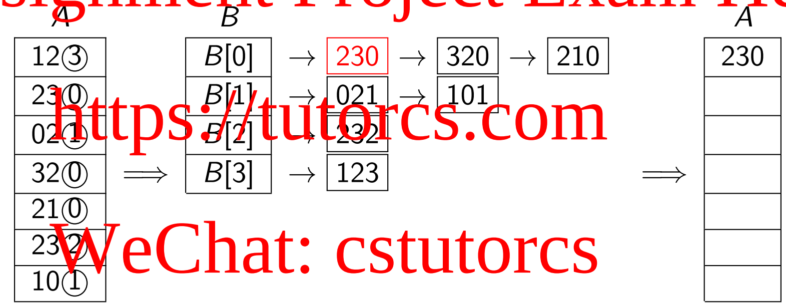
- $B[0]$ → 230 → 320 → 210
- $B[1]$ → 021 → 101
- $B[2]$ → 232
- $B[3]$ → 123

$\Longrightarrow$

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	$\rightarrow$ 101	
$B[2]$	$\rightarrow$ 232		
$B[3]$	$\rightarrow$ 123		

$\Longrightarrow$

A
230
320

(Single-digit) Bucket Sort

Sort array A by last digit:

A
12(3)
23(0)
02(1)
32(0)
21(0)
23(2)
10(1)

$\Longrightarrow$

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			
$B[0]$	$\rightarrow$ 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

(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			
$B[0]$	$\to$ 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
101
232
123

(Single-digit) Bucket Sort

- Sorts numbers by a single digit.
- Create a "bucket" for each possible digit: Array $B[0..R-1]$ of lists
- Copy item with digit i into bucket $B[i]$
- At the end copy buckets in order into A.

Bucket-sort(A, d)
A: array of size n, contains numbers with digits in $\{0, \ldots, R-1\}$
d: index of digit by which we wish to sort

Initialize an array $B[0\ldots R-1]$ of empty lists
for $i \leftarrow 0$ to $n-1$ **do**
  Append $A[i]$ at end of $B[d^{\text{th}}$ digit of $A[i]]$
$i \leftarrow 0$
for $j \leftarrow 0$ to $R-1$ **do**
  while $B[j]$ is non-empty **do**
    move first element of $B[j]$ to $A[i{+}{+}]$

- This is **stable**: equal items stay in original order.
- Run-time $\Theta(n+R)$, auxiliary space $\Theta(n+R)$

Count Sort

- Bucket sort wastes space for linked lists.
- Observe: We know exactly where numbers in $B[j]$ go:
  - The first of them is at index $|B[0]| + |B[1]| + \cdots + |B[j-1]|$
  - The others follow.

- So we don't need the lists; it's enough to count how many there would be in it.

Count Sort Pseudocode

key-indexed-count-sort(A, d)
A: array of size n, contains numbers with digits in $\{0, \ldots, R-1\}$
d: index of digit by which we wish to sort

// count how many of each kind there are
    count $\leftarrow$ array of size R, filled with zeros
    for $i \leftarrow 0$ to $n-1$ **do**
        increment *count*[d^{th} digit of $A[i]$]

// find left boundary for each kind
    idx $\leftarrow$ array of size R, *idx*$[0] = 0$
    for $i \leftarrow 1$ to $R-1$ **do**
        idx$[i] \leftarrow$ *idx*$[i-1]$ + *count*$[i-1]$

// move to new array in sorted order, then copy back
    aux $\leftarrow$ array of size n
    for $i \leftarrow 0$ to $n-1$ **do**
        aux[*idx*[d^{th} digit of $A[i]$]] $\leftarrow A[i]$
        increment *idx*[d^{th} digit of $A[i]$]
    $A \leftarrow$ *copy*(*aux*)

Example: Count Sort

//count how many of each kind there are
1. *count* ← array of size R, filled with zeros
2. **for** $i \leftarrow 0$ to $n-1$ **do**
3. increment *count*[d^{th} digit of $A[i]$]

. . .

A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	0	
1	0	
2	0	
3	0	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
    count ← array of size R, filled with zeros
    for i ← 0 to n − 1 do
        increment count[d^th digit of A[i]]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	0	
1	0	
2	0	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
1.      count ← array of size R, filled with zeros
2.      for i ← 0 to n − 1 do
3.           increment count[d^th digit of A[i]]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	1	
1	0	
2	0	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
    count ← array of size R, filled with zeros
    for i ← 0 to n − 1 do
        increment count[d^th digit of A[i]]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	1	
1	1	
2	0	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

//count how many of each kind there are
1. *count* ← array of size R, filled with zeros
2. **for** $i \leftarrow 0$ to $n-1$ **do**
3. increment *count*[d^{th} digit of $A[i]$]

...

A
123
230
021
320
210
232
101

⟹

	count	*idx*
0	2	
1	1	
2	0	
3	1	

⟹

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
     count ← array of size R, filled with zeros
     for i ← 0 to n − 1 do
          increment count[d^th digit of A[i]]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	3	
1	1	
2	0	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
1.    count ← array of size R, filled with zeros
2.    for i ← 0 to n − 1 do
3.        increment count[d^th digit of A[i]]
...
```


A
12(3)
23(0)
02(1)
32(0)
21(0)
23(2)
10(1)

$\Longrightarrow$

	count	*idx*
0	3	
1	1	
2	1	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
//count how many of each kind there are
1.      count ← array of size R, filled with zeros
2.      for i ← 0 to n − 1 do
3.              increment count[d^th digit of A[i]]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	3	
1	2	
2	1	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
...
// find left boundary for each kind
    idx ← array of size R, idx[0] ← 0
    for i ← 1 to R − 1 do
        idx[i] ← idx[i − 1] + count[i − 1]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	3	0
1	2	
2	1	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
...
// find left boundary for each kind
    idx ← array of size R, idx[0] ← 0
    for i ← 1 to R − 1 do
        idx[i] ← idx[i − 1] + count[i − 1]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	0
1	2	3
2	1	
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
...
// find left boundary for each kind
    idx ← array of size R, idx[0] ← 0
    for i ← 1 to R − 1 do
        idx[i] ← idx[i − 1] + count[i − 1]
...
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	*idx*
0	3	0
1	2	3
2	1	5
3	1	

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
    aux ← array of size n
    for i ← 0 to n − 1 do
        aux[idx[d^th digit of A[i]]] ← A[i]
        increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	0
1	2	3
2	1	5
3	1	6

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	

Example: Count Sort

...
// move to new array in sorted order, then copy back
$aux \leftarrow$ array of size n
for $i \leftarrow 0$ to $n-1$ **do**
  $aux[idx[d^{\text{th}}$ digit of $A[i]]] \leftarrow A[i]$
  increment $idx[d^{\text{th}}$ digit of $A[i]]$

	A
	123
	230
	021
	320
	210
	232
	101

$\Longrightarrow$

	$count$	idx
0	3	0
1	2	3
2	1	5
3	1	7

$\Longrightarrow$

	aux
0	
1	
2	
3	
4	
5	
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
    aux ← array of size n
    for i ← 0 to n − 1 do
          aux[idx[d^th digit of A[i]]] ← A[i]
          increment idx[d^th digit of A[i]]
```

A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	1
1	2	3
2	1	5
3	1	7

$\Longrightarrow$

	aux
0	230
1	
2	
3	
4	
5	
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
      aux ← array of size n
      for i ← 0 to n − 1 do
            aux[idx[d^th digit of A[i]]] ← A[i]
            increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	1
1	2	4
2	1	5
3	1	7

$\Longrightarrow$

	aux
0	230
1	
2	
3	021
4	
5	
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
      aux ← array of size n
      for i ← 0 to n − 1 do
            aux[idx[d^th digit of A[i]]] ← A[i]
            increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	2
1	2	4
2	1	5
3	1	7

$\Longrightarrow$

	aux
0	230
1	320
2	
3	021
4	
5	
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
      aux ← array of size n
      for i ← 0 to n − 1 do
          aux[idx[d^th digit of A[i]]] ← A[i]
          increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	3
1	2	4
2	1	5
3	1	7

$\Longrightarrow$

	aux
0	230
1	320
2	210
3	021
4	
5	
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
    aux ← array of size n
    for i ← 0 to n − 1 do
        aux[idx[d^th digit of A[i]]] ← A[i]
        increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	3
1	2	4
2	1	6
3	1	7

$\Longrightarrow$

	aux
0	230
1	320
2	210
3	021
4	
5	232
6	123

Example: Count Sort

```
...
  // move to new array in sorted order, then copy back
    aux ← array of size n
    for i ← 0 to n − 1 do
        aux[idx[d^th digit of A[i]]] ← A[i]
        increment idx[d^th digit of A[i]]
```


A
123
230
021
320
210
232
101

$\Longrightarrow$

	count	idx
0	3	3
1	2	5
2	1	6
3	1	7

$\Longrightarrow$

	aux
0	230
1	320
2	210
3	021
4	101
5	232
6	123

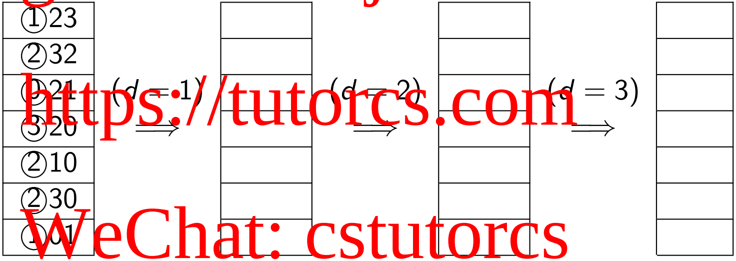
MSD-Radix-Sort

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


```
MSD-Radix-sort(A, ℓ ← 0, r ← n−1, d ← 1)
    if ℓ < r
        key-indexed-count-sort(A[ℓ..r], d)
        if d < m
            for i ← 0 to R − 1 do
                let ℓ_i and r_i be boundaries of ith bin
                    (i.e., A[ℓ_i..r_i] all have dth digit i)
                MSD-Radix-sort(A, ℓ_i, r_i, d+1)
```


- ℓ_i and r_i are automatically computed with *count-sort*
- Drawback of *MSD-Radix-Sort*: many recursions
- **Auxiliary space**: $\Theta(n + R + m)$ (for *count-sort* and recursion stack)

MSD-Radix-Sort Example



MSD-Radix-Sort Example

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

MSD-Radix-Sort Example

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

MSD-Radix-Sort Example

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

MSD-Radix-Sort Example

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

MSD-Radix-Sort Example

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

LSD-Radix-Sort

LSD-radix-sort(A)
A: array of size n, contains m-digit radix-R numbers
for $d \leftarrow m$ down to 1 **do**
  key-indexed-count-sort(A, d)

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 $d, \ldots, m$ of each entry.
- **Time cost**: $\Theta(m(n + R))$ **Auxiliary space**: $\Theta(n + R)$

Radix-Sort: Final Comments

- Bucket Sort and Count Sort can be extended to any applications where the keys being sorted come from a known ordered set of cardinality R.

- MSD-Radix-Sort and LSD-Radix-Sort can work with other auxilliary digit-sorting algorithms, not just Count Sort.
- The auxilliary digit sorting algorithm for LSD-Radix-Sort must be stable.

Summary

- Sorting is an important and *very* well-studied problem
- Can be done in $\Theta(n \log n)$ time; faster is not possible for general input

- HeapSort is the only $O(n \log n)$-time algorithm we have seen with $O(1)$ auxiliary space.
- MergeSort is also $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.

- QuickSort is worst-case $\Theta(n^2)$, but often the fastest in practice
- CountSort, RadixSort can achieve $o(n \log n)$ if the input is special

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, and average-case run-times can all differ. Randomization may result in all cases having the same expected run-time.