Outline

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

- Sorting and Randomized Algorithms
  - QuickSelect
    - Randomized Algorithms
    - QuickSort
    - Lower Bound for Comparison-Based Sorting
    - Non-Comparison-Based Sorting
Selection Problem

- Given array $A$ of $n$ numbers, and $0 \leq k < n$, find the element that would be at position $k$ if $A$ was sorted
  - ‘select $k$’
  - $k$ elements are smaller or equal, $n - 1 - k$ elements are larger or equal

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
<td>10</td>
<td>0</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>20</td>
<td>40</td>
<td>70</td>
</tr>
</tbody>
</table>

- Special case: median finding ($k = \left\lfloor \frac{n}{2} \right\rfloor$)

- Heap-based selection can be done in $\Theta(n + k \log n)$
  - this is $\Theta(n \log n)$ for median finding
    - the same cost as our best sorting algorithms

- **Question**: can we do selection in linear time?
  - yes, with quick-select (average case analysis)
  - subroutines for quick-select also useful for sorting algorithms
Crucial Subroutines

- *quick-select* and related algorithm *quick-sort* rely on two subroutines
  - *choose-pivot*(A)
    - return an index \( p \) in \( A \)
    - use *pivot-value* \( v \leftarrow A[p] \) to rearrange the array
  - *partition*(\( A, p \)) rearranges \( A \) so that
    - all items in \( A \) [0, ..., \( i - 1 \)] are \( \leq v \)
    - pivot-value \( v \) is in \( A[i] \)
    - all items in \( A \) [\( i + 1, ..., n - 1 \)] are \( \geq v \)
    - index \( i \) is called *pivot-index* \( i \)
    - *partition*(\( A, p \)) returns *pivot-index* \( i \)
      - \( i \) is a correct location of \( v \) in sorted \( A \)
      - if we were interested in \( \text{select}(i) \), then \( v \) would be the answer

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
30 & 60 & 10 & 0 & v = 50 & 80 & 90 & 20 & 40 & 70 \\
\end{array}
\]
Choosing Pivot

- Simplest idea for *choose-pivot*
  - always select rightmost element in array

\[
\text{choose-pivot1}(A) \\
\text{return } A.\text{size}() - 1
\]

- Will consider more sophisticated ideas later
Partition Algorithm

\[ partition(A, p) \]
\[ A: \text{array of size } n, \ p: \text{integer s.t. } 0 \leq p < n \]
\[ \text{create empty lists } small, \text{equal and large} \]
\[ v \leftarrow A[p] \]
\[ \text{for each element } x \text{ in } A \]
\[ \quad \text{if } x < v \text{ then } small.\text{append}(x) \]
\[ \quad \text{else if } x > v \text{ then } large.\text{append}(x) \]
\[ \quad \text{else equal.\text{append}(x)} \]
\[ i \leftarrow small.\text{size} \]
\[ j \leftarrow equal.\text{size} \]
\[ \text{overwrite } A[0 \ldots i - 1] \text{ by elements in small} \]
\[ \text{overwrite } A[i \ldots i + j - 1] \text{ by elements in equal} \]
\[ \text{overwrite } A[i + j \ldots n - 1] \text{ by elements in large} \]
\[ \text{return } i \]

- Easy linear-time implementation using extra (auxiliary) \( \Theta(n) \) space
- More challenging: partition \textit{in-place}, i.e. \( O(1) \) auxiliary space
Efficient In-Place partition (Hoare)

almost done, just swap with pivot $v$

i = -1

30  60  10  0  50  80  90  20  40  \( v = 70 \)

i = 5

30  60  10  0  50  80  90  20  40  \( v = 70 \)

j = 8

i = 5

30  60  10  0  50  40  90  20  80  \( v = 70 \)

j = 8

i = 6

30  60  10  0  50  40  90  20  80  \( v = 70 \)

j = 7

i = 6

30  60  10  0  50  40  20  90  80  \( v = 70 \)

j = 7

j = 6

30  60  10  0  50  40  20  90  80  \( v = 70 \)

i = 7

j = 6

30  60  10  0  50  40  20  90  80  \( v = 70 \)

i = 7

30  60  10  0  50  40  20  80  0  \( v = 70 \)
Efficient In-Place partition (Hoare)

- **Idea Summary:** Keep swapping the outer-most wrongly-positioned pairs

<table>
<thead>
<tr>
<th>≤ v</th>
<th>?</th>
<th>≥ v</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>v</td>
</tr>
</tbody>
</table>

- One possible implementation

```plaintext
do i ← i + 1 while i < n and A[i] ≤ v
do j ← j − 1 while j > 0 and A[j] ≥ v
```

- More efficient (for quickselect and quicksort) when many repeating elements

```plaintext
do i ← i + 1 while i < n and A[i] < v
do j ← j − 1 while j > 0 and A[j] > v
```

- Can simplify the loop bounds

```plaintext
do i ← i + 1 while A[i] < v
do j ← j − 1 while j ≥ i and A[j] > v
```
Efficient In-Place partition (Hoare)

**partition** \((A, p)\)

\(A\): array of size \(n\)

\(p\): integer s.t. \(0 \leq p < n\)

\(\text{swap}(A[n - 1], A[p])\)

\(i \leftarrow -1, \quad j \leftarrow n - 1, \quad v \leftarrow A[n - 1]\)

**loop**

\(\text{do } i \leftarrow i + 1 \text{ while } A[i] < v\)

\(\text{do } j \leftarrow j - 1 \text{ while } j \geq i \text{ and } A[j] > v\)

if \(i \geq j\) then break

else \(\text{swap}(A[i], A[j])\)

end loop

\(\text{swap}(A[n - 1], A[i])\)

return \(i\)

- Running time is \(\Theta(n)\)
Efficient In-Place partition (Hoare)

\textit{partition} \ ((A, p))

\begin{itemize}
  \item \textbf{A}: array of size \( n \)
  \item \textbf{p}: integer s.t. \( 0 \leq p < n \)
\end{itemize}

\begin{align*}
  &\text{swap}(A[n - 1], A[p]) \\
  &i \leftarrow -1, \quad j \leftarrow n - 1, \quad v \leftarrow A[n - 1] \\
  \textbf{loop} \\
  &\textbf{do} \quad i \leftarrow i + 1 \quad \textbf{while} \quad A[i] < v \\
  &\textbf{do} \quad j \leftarrow j - 1 \quad \textbf{while} \quad j \geq i \quad \textbf{and} \quad A[j] > v \\
  &\quad \textbf{if} \quad i \geq j \quad \textbf{then} \quad \textbf{break} \\
  &\quad \textbf{else} \quad \textbf{swap}(A[i], A[j]) \\
  \textbf{end loop} \\
  &\text{swap}(A[n - 1], A[i]) \\
  &\text{return} \quad i
\end{align*}

- Running time is \( \Theta(n) \)
Quick Select Algorithm

- Find item that would be in $A[k]$ if $A$ was sorted
- Similar to quick-sort, but recurse only on one side (“quick-sort with pruning”)
- Example: $\text{select}(k = 4)$
  - [the correct answer is 40 in this case]

| 30 | 60 | 10 | 0 | 50 | 80 | 90 | 20 | 40 | $v=70$ |

- $i > k$, search recursively in the left side to select $k$
Quick Select Algorithm

- Example continued: $\text{select}(k = 4)$

| 30 | 60 | 10 | 0 | 50 | 40 | $v=20$ |

- $i < k$, search recursively on the right, select $k - (i + 1)$
  - $k = 1$ in our example
Quick Select Algorithm

- Example continued: \( \text{select}(k = 1) \)

```
30 50 40 \( v=60 \)
```

- \( i > k \), search on the left to select \( k \)
Quick Select Algorithm

- Example continued: \textbf{select}(k = 1)

\[ i = 1 \]

\[ \begin{array}{ccc}
30 & 50 & v=40 \\
\end{array} \]

\textit{partition, } v=40

\[ i=1 \]

\[ \begin{array}{ccc}
30 & 40 & 50 \\
\end{array} \]

- \( i = k, \) found our item, done!
- In our example, we got to subarray of size 3
- Often stop much sooner than that
  - running time?
QuickSelect Algorithm

**quick-select1**($A, k$)

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

$p \leftarrow \text{choose-pivot1}(A)$

$i \leftarrow \text{partition}(A, p)$

if $i = k$ then

return $A[i]$

else if $i > k$ then

return quick-select1($A[0, 1, ..., i - 1], k$)

else if $i < k$ then

return quick-select1($A[i + 1, ..., n - 1], k - (i + 1)$)

- **Best case**
  - first chosen pivot could have pivot-index $k$
  - no recursive calls, total cost $\Theta(n)$

- **Worst case**: recurrence equation $T(n) = \begin{cases} cn + T(n - 1) & n > 1 \\ c & n = 1 \end{cases}$
QuickSelect Algorithm

- **Worst case**: recurrence equation $T(n) = \begin{cases} cn + T(n-1) & n > 1 \\ c & n = 1 \end{cases}$

- Solution: repeatedly expand until we see a pattern forming

  \[ T(n) = cn + T(n-1) \]

  \[ T(n-1) = c(n-1) + T(n-2) \]

  \[ T(n) = cn + c(n-1) + T(n-2) \]  \hspace{1cm} \text{after 1 expansion}

  \[ T(n-2) = c(n-2) + T(n-3) \]

  \[ T(n) = cn + c(n-1) + c(n-2) + T(n-3) \]  \hspace{1cm} \text{after 2 expansions}

- After $i$ expansions

  \[ T(n) = cn + c(n-1) + c(n-2) + \cdots + c(n-i) + T(n-(i+1)) \]

- Stop expanding when get to base case \( T(n-(i+1)) = T(1) \)

- Happens when \( n-(i+1) = 1 \), or, rewriting, \( i = n-2 \)

- Thus \( T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + T(1) \)

  \[ = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + c \]

  \[ = c(n + (n-1) + \cdots + 2 + 1) \in \Theta(n^2) \]
Average-Case Analysis of *quick-select1*

\[ T_{avr}(n) = \frac{1}{\text{# instances of size } n} \sum_{I: \text{size}(I)=n} T(I) \]

- Need to make some assumptions
- First assumption
  - all input numbers are distinct
  - this assumption is just for simpler analysis, can prove the same thing without this assumption
Average-Case Analysis of quick-select

- **QuickSelect** is *comparison-based*
  - only cares if $A[i] < A[j]$ for $i, j$
  - does not care what the actual values of $A[i], A[j]$ are

\[
\begin{array}{cccc}
I_1 & 30 & 60 & 0 & 10 \\
I_2 & 20 & 50 & 10 & 15 \\
\end{array}
\]

- **QuickSelect** makes exactly the same sequences of steps on $I_1$ and $I_2$
  - therefore $T(I_1) = T(I_2)$

- Any comparison based algorithm has exactly the same running time for arrays that have the same relative order of elements, regardless of actual array values

- Second assumption: we are sorting integers $0, ..., n - 1$
  - now there are $n!$ possible input instances $I$
  - more formal proof uses sorting permutations
    - permutation $\pi$ for which $A[\pi(0)] \leq A[\pi(1)] \leq ... \leq A[\pi(n - 1)]$
    - for $I_1$ (and $I_2$) sorting permutation is $\pi = (2, 3, 0, 1)$
    - assume *each sorting permutation is equally likely*
    - $n!$ possible permutations
Average-Case Analysis of *quick-select1*

$$T_{avr} (n) = \frac{1}{\# \text{ instances of size } n} \sum_{I: \text{size}(I)=n} T(I)$$

- Example for $n = 3$, using all the assumptions

$$T_{avr} (3) = \frac{1}{3!} (T(\{0,1,2\}) + T(\{0,2,1\}) + T(\{1,0,2\}) + T(\{1,2,0\}) + T(\{2,0,1\}) + T(\{2,1,0\}))$$
Average-Case Analysis of quick-select1

- Recall that pivot is last array element
- Pivot index is equal to pivot value due to assuming we sort $0, \ldots, n - 1$

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
2 & 3 & 0 & v=1 \\
\end{array}
\]

for $v=1$, pivot index $i = 1$

- Partition sum over different pivot indexes

\[
T^{av}_r(n) = \frac{1}{n!} \sum_{I:\text{Size}(I)=n} T(I) = \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{I:\text{size}(I)=n, pivot is i} T(I)
\]

- Example for $n = 3$

\[
T^{av}_r(3) = \frac{1}{3!} \left( T(\{0,1,2\}) + T(\{0,2,1\}) + T(\{1,0,2\}) + T(\{1,2,0\}) + T(\{2,0,1\}) + T(\{2,1,0\}) \right)
\]

\[
T^{av}_r(3) = \frac{1}{3!} \left( T(\{1,2, 0\}) + T(\{2,1, 0\}) + \\
(T(\{0,2, 1\}) + T(\{2,0, 1\})) + \\
(T(\{0,1, 2\}) + T(\{1,0, 2\})) \right)
\]
Average-Case Analysis of quick-select1

- Partition sum over different pivots
  \[ T^{avr}(n) = \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{I:\text{size}(I)=n, \text{pivot is } i} T(I) \]

- There are \((n - 1)!\) input instances \(I\) with pivot index \(i\)

\[
\begin{array}{|c|c|c|c|}
\hline
A & \text{choice of } n - 1 \text{ items: anything but } i & \text{choice of } n - 2 \text{ items: anything but } i \text{ and } A[0] & \ldots & \text{‘choice’ of 1 items} \\
\hline
\text{no choice} \hline
\end{array}
\]

- One can show (will only hint at the proof with example for \(n = 4, i = 1\))

\[
\sum_{I:\text{size}(I)=n, \text{pivot is } i} T(I) \leq (n - 1)! \cdot cn + (n - 1)! \cdot \max\{T^{avr}(i), T^{avr}(n - i - 1)\}
\]

- Therefore

\[ T^{avr}(n) \leq cn + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{avr}(i), T^{avr}(n - i - 1)\} \]
Average-Case Analysis of quick-select1

- Let \( n = 4, i = 1 \)

\[
\sum_{\text{size}(I)=4, \ pivot \ is \ 1} T(I) = T(\{0,2,3, 1\}) + T(\{0,3,2, 1\}) + T(\{2,0,3, 1\}) + T(\{2,3,0, 1\}) + T(\{3,0,2, 1\}) + T(\{3,2,0, 1\})
\]

- Total work is proportional to comparisons, will count comparisons

<table>
<thead>
<tr>
<th>comparisons to partition:</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>Total: 3(3)!</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>{0,2,3, 1}</td>
<td>{0,3,2, 1}</td>
<td>{2,0,3, 1}</td>
<td>{2,3,0, 1}</td>
<td>{3,0,2, 1}</td>
<td>{3,2,0, 1}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| partitions               | \{0\} | \{2,3\} | \{0\} | \{3,2\} | \{2,3\} | \{0\} | \{3,2\} | \{0\} | \{3,2\} | (assume stable order)
Average-Case Analysis of quick-select

- Let $n = 4, i = 1$
  \[ \sum_{I: \text{size}(I) = 4, \text{pivot is 1}} T(I) = T(\{0,2,3, 1\}) + T(\{0,3,2, 1\}) + T(\{2,0,3, 1\}) + T(\{2,3,0, 1\}) + T(\{3,0,2, 1\}) + T(\{3,2,0, 1\}) \]

- Total work is proportional to comparisons, will count comparisons

<table>
<thead>
<tr>
<th>comparisons to partition</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>{0,2,3, 1}</td>
<td>{0,3,2, 1}</td>
<td>{2,0,3, 1}</td>
<td>{2,3,0, 1}</td>
<td>{3,0,2, 1}</td>
<td>{3,2,0, 1}</td>
<td></td>
</tr>
<tr>
<td>partitions</td>
<td>{0}</td>
<td>{2,3}</td>
<td>{0}</td>
<td>{3,2}</td>
<td>{0}</td>
<td>{2,3}</td>
<td>{0}</td>
</tr>
</tbody>
</table>

Case 1: $k > i$

\[
T(\{2,3\}) + T(\{3,2\}) + T(\{2,3\}) + T(\{2,3\}) + T(\{3,2\}) + T(\{3,2\}) = T(\{0,1\}) + T(\{1,0\}) + T(\{0,1\}) + T(\{0,1\}) + T(\{1,0\}) + T(\{1,0\})
\]

since only relative order matters

swap
Average-Case Analysis of quick-select1

- Let $n = 4, i = 1$
  \[
  \sum_{I: \text{size}(I) = 4, \ pivot \ is \ 1} T(I) = T(\{0,2,3, 1\}) + T(\{0,3,2, 1\}) + T(\{2,0,3, 1\}) + T(\{2,3,0, 1\}) + T(\{3,0,2, 1\}) + T(\{3,2,0, 1\})
  \]

- Total work is proportional to comparisons, will count comparisons

<table>
<thead>
<tr>
<th>comparisons to partition</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>{0,2,3, 1}</td>
<td>{0,3,2, 1}</td>
<td>{2,0,3, 1}</td>
<td>{2,3,0, 1}</td>
<td>{3,0,2, 1}</td>
<td>{3,2,0, 1}</td>
<td></td>
</tr>
<tr>
<td>partitions</td>
<td>{0}</td>
<td>{2,3}</td>
<td>{0}</td>
<td>{3,2}</td>
<td>{0}</td>
<td>{2,3}</td>
<td>{0}</td>
</tr>
</tbody>
</table>

Case 1: $k > i$

\[
T(\{2,3\}) + T(\{3,2\}) + T(\{2,3\}) + T(\{2,3\}) + T(\{3,2\}) + T(\{3,2\})
\]

\[
= T(\{0,1\}) + T(\{1,0\}) + T(\{0,1\}) + T(\{1,0\}) + T(\{0,1\}) + T(\{1,0\})
\]

\[
= 2! T^{avr}(2) + 2! T^{avr}(2) + 2! T^{avr}(2)
\]

Total recursive comparisons
\[
\frac{3!}{2!} \cdot 2! T^{avr}(2) = 3! T^{avr}(2)
\]
Average-Case Analysis of quick-select1

- Let $n = 4$, $i = 1$

\[
\sum_{I: \text{size}(I) = 4, \text{pivot is 1}} T(I) = 
T(\{0, 2, 3, 1\}) + T(\{0, 3, 2, 1\}) + T(\{2, 0, 3, 1\}) + T(\{2, 3, 0, 1\}) + T(\{3, 0, 2, 1\}) + T(\{3, 2, 0, 1\})
\]

- Total work is proportional to comparisons, will count comparisons

<table>
<thead>
<tr>
<th>comparisons to partition</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>Total: 3(3)!</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>{0, 2, 3, 1}</td>
<td>{0, 3, 2, 1}</td>
<td>{2, 0, 3, 1}</td>
<td>{2, 3, 0, 1}</td>
<td>{3, 0, 2, 1}</td>
<td>{3, 2, 0, 1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>partitions</td>
<td>{0}</td>
<td>{2, 3}</td>
<td>{0}</td>
<td>{3, 2}</td>
<td>{0}</td>
<td>{2, 3}</td>
<td>{0}</td>
<td>{3, 2}</td>
</tr>
</tbody>
</table>

Case 2: $k < i$

\[
T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\})
\]

\[
\begin{align*}
1! T^{avr}(1) & + 1! T^{avr}(1) + 1! T^{avr}(1) + 1! T^{avr}(1) + 1! T^{avr}(1) + 1! T^{avr}(1)
\end{align*}
\]

Total recursive comparisons

\[
\frac{3!}{1!} 1! T^{avr}(1) = 3! T^{avr}(1)
\]

\[
\frac{[\text{Case 1, total recursive comparisons:}]}{= 3! T^{avr}(2)} \leq 3! \max\{T^{avr}(1), T^{avr}(2)\}
\]

Combining both cases, total recursive comparisons:

\[
\leq 3(3)! + 3! \max\{T^{avr}(1), T^{avr}(2)\}
\]

Adding comparisons to partition:
### Average-Case Analysis of quick-select1

- Let \( n = 4, i = 1 \)
  
  \[
  
  \sum_{I: \text{size}(I)=4, \text{pivot is } 1} T(I) = T(\{0,2,3, 1\}) + T(\{0,3,2, 1\}) \\
  + T(\{2,0,3, 1\}) + T(\{2,3,0, 1\}) \\
  + T(\{3,0,2, 1\}) + T(\{3,2,0, 1\})
  
  \]

- Total work is proportional to comparisons, will count comparisons

<table>
<thead>
<tr>
<th>comparisons to partition:</th>
<th>(3)</th>
<th>(3)</th>
<th>(3)</th>
<th>(3)</th>
<th>(3)</th>
<th>(3)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>({0,2,3, 1})</td>
<td>({0,3,2, 1})</td>
<td>({2,0,3, 1})</td>
<td>({2,3,0, 1})</td>
<td>({3,0,2, 1})</td>
<td>({3,2,0, 1})</td>
<td></td>
</tr>
<tr>
<td>partitions</td>
<td>({0})</td>
<td>({2,3})</td>
<td>({0})</td>
<td>({3,2})</td>
<td>({0})</td>
<td>({2,3})</td>
<td>({0})</td>
</tr>
</tbody>
</table>

**Case 2: \( k < i \)**

\[
T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\}) + T(\{0\})
\]

\[
\frac{3!}{1!} T^{avr}(1) = 3! T^{avr}(1)
\]

Total recursive comparisons

\[
\sum_{I: \text{size}(I)=n, \text{pivot is } i} T(I) \leq (n-1)!cn + (n-1)! \max\{T^{avr}(i), T^{avr}(n-i-1)\}
\]

Adding comparisons to partition:

\[
\leq 3(3)! + 3! \max\{T^{avr}(1), T^{avr}(2)\}
\]
Theorem: \( T(n) \in O(n) \)

Proof:

- will prove \( T(n) \leq 4cn \) by induction on \( n \)
- base case, \( n = 1 \): \( T(1) = c \leq 4c \cdot 1 \)
- induction hypothesis: assume \( T(m) \leq 4cm \) for all \( m < n \)
- need to show \( T(n) \leq 4cn \)

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

\[
\leq c \cdot n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{4ci, 4c(n-i-1)\}
\]

\[
\leq c \cdot n + \frac{4c}{n} \sum_{i=0}^{n-1} \max\{i, n-i-1\}
\]
Proof: (cont.) $T(n) \leq c \cdot n + \frac{4c}{n} \sum_{i=0}^{n-1} \max\{i, n - i - 1\} \leq c \cdot n + \frac{4c}{n} \cdot \frac{3}{4} n^2 = 4cn$

$$\sum_{i=0}^{n-1} \max\{i, n - i - 1\} = \sum_{i=0}^{\frac{n-1}{2}} \max\{i, n - i - 1\} + \sum_{i=\frac{n}{2}}^{n-1} \max\{i, n - i - 1\}$$

$$= \max\{0, n - 1\} + \max\{1, n - 2\} + \max\{2, n - 3\} + \cdots + \max\left\{\frac{n}{2} - 1, \frac{n}{2}\right\}$$

$$+ \max\left\{\frac{n}{2}, \frac{n}{2} - 1\right\} + \max\left\{\frac{n}{2} + 1, \frac{n}{2} - 2\right\} + \cdots + \max\{n - 1, 0\}$$

$$= (n - 1) + (n - 2) + \cdots + \frac{n}{2} + \frac{n}{2} + \left(\frac{n}{2} + 1\right) + \cdots + (n - 1)$$

$$= \left(\frac{3n}{2} - 1\right) \frac{n}{2} + \left(\frac{3n}{2} - 1\right) \frac{n}{4} \leq \frac{3}{4} n^2$$

exactly what we need for the proof
Average-Case Analysis of quick-select

- Proved average case time $T(n)$ is $O(n)$
- Average case is also $\Omega(n)$ since have to perform $\text{partition}(A,p)$
- Therefore average case is $T(n)$ is $\Theta(n)$
Outline

- 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.
- The cost will depend on both the input and the random numbers used.

**Goal**
- Shift the dependency of run-time from what we cannot control (the input), to what we can control (random numbers).
- No more bad instances, just unlucky numbers.
  - If running time is long on some instance, it’s because we generated unlucky random numbers, not because of the instance itself.

**Side note**
- Computers cannot generate truly random numbers.
- We assume there is a pseudo-random number generator (PRNG), a deterministic program that uses an initial value or *seed* to generate a sequence of seemingly random numbers.
- Quality of randomized algorithm depends on the quality of the PRNG.
Expected Running Time

- How do we measure the running time of a randomized algorithm?
  - it depends on the input $I$ and on $R$, the sequence of random numbers an algorithm choses during execution

- Define $T(I, R)$ to be running time of randomized algorithm for instance $I$ and $R$

- The expected running time $T^{\text{exp}}(I)$ for instance $I$ is expected value for $T(I, R)$
  $$T^{\text{exp}}(I) = E[T(I, R)] = \sum_{\text{all possible sequences } R} T(I, R) \cdot \Pr[R]$$

- **Worst-case expected running time**
  $$T^{\text{exp}}(n) = \max_{\{I: \text{size}(I) = n\}} T^{\text{exp}}(I)$$

- **Average-case expected running time**
  $$T^{\text{exp}}(n) = \frac{1}{|I: \text{size}(I) = n|} \sum_{I: \text{size}(I) = n} T^{\text{exp}}(I)$$

- Usually design $A$ so that all instances of size $n$ have the same expected run time
- Thus the average and worst case expected run times are the same, and we just compute the worst case expected time
Expected Running Time

- How do we measure the running time of a randomized algorithm?
  - it depends on the input $I$ and on $R$, the sequence of random numbers an algorithm chooses during execution

- Define $T(I, R)$ to be running time of randomized algorithm for instance $I$ and $R$

- The expected running time $T_{\text{exp}}(I)$ for instance $I$ is expected value for $T(I, R)$

$$T_{\text{exp}}(I) = E[T(I, R)] = \sum_{\text{all possible sequences } R} T(I, R) \cdot \Pr[R]$$

- **Worst-case expected running time** $T_{\text{exp}}(n) = \max \{T_{\text{exp}}(I) : \text{size}(I) = n\}$

- **Average-case expected running time** $T_{\text{exp}}(n) = \frac{1}{|\{I : \text{size}(I) = n\}|} \sum_{\text{Size}(I) = n} T_{\text{exp}}(I)$

- Usually design $A$ so that all instances of size $n$ have the same expected run time

- Thus average and worst case expected run times are usually the same
  - just compute the worst case expected time

- Sometimes we also want to know the running time if we got really unlucky with the random numbers $R$ we generate during the execution, or, formally

$$\max_{R} \max_{\{I : \text{size}(I) = n\}} T(I, R)$$
Randomized QuickSelect: Shuffle

- **Goal**: create a randomized version of *QuickSelect* for which all input has the same expected run-time
- **First idea**: first randomly permute input using *shuffle* and then run selection algorithm

```plaintext
shuffle(A)
A : array of size n
for i ← 0 to n − 1 do
    swap(A[i], A[random(i + 1)])
```

- `random(n)` returns an integer uniformly sampled from {0, 1, 2, ..., n − 1}
- can show that expected running time is $\Theta(n)$, the same as average running time
Randomized QuickSelect: Shuffle

- **Goal**: create a randomized version of *QuickSelect* for which all input has the same expected run-time
- **First idea**: first randomly permute input using *shuffle* and then run selection algorithm

```
shuffle(A)
A : array of size n
for i ← 0 to n − 1 do
    swap(A[i], A[random(i + 1)])
```

- `random(n)` returns an integer uniformly sampled from `{0, 1, 2, ..., n − 1}`
- can show that expected running time is $\Theta(n)$, the same as average running time
- if we get very unlucky with random numbers, we could get a sorted or almost sorted array after shuffle, resulting in $O(n^2)$ performance for selection algorithm
  - probability of this happening is almost zero
- whereas the user is quite likely to give instance which is sorted or almost sorted to the selection algorithm
  - probability is far from zero, humans often produce almost sorted data
Randomized QuickSelect: Random Pivot

- **Second idea**: select a random pivot from \{0, 1, 2, ..., n - 1\}

```cpp
choose-pivot2(A)
return random(A.size())
```

- Simpler and more efficient than shuffling the array
- Usually fastest in practice
- Expected running time is again $\Theta(n)$
Efficiency of Randomized QuickSelect

- Assume all elements of A are distinct
- Select pivot with equal probability at each recursive call, and independently from other recursive calls
  - \( P(\text{pivot has index } i) = \frac{1}{n} \) for any instance of size \( n \)
- \( T^{\text{exp}}(I) \) depends only on the size of \( I \), not the contents of \( I \)
- Let \( T^{\text{exp}}(n) \) be expected time on an instance of size \( n \)
- Running time to partition array is \( cn \), and with probability \( 1/n \) pivot-index is \( i \)

\[
\begin{array}{|c|c|c|}
\hline
T^{\text{exp}}(i) & \nu & T^{\text{exp}}(n - i - 1) \\
\hline
\text{size } i & \text{size } n - i - 1 & \text{running time if pivot index is } i \leq c \cdot n + \max\{T^{\text{exp}}(i), T^{\text{exp}}(n - i - 1)\} \\
\hline
\end{array}
\]
Efficiency of Randomized QuickSelect

Running time if pivot-index is $i \leq c \cdot n + \max\{T^{\text{exp}}(i), T^{\text{exp}}(n - i - 1)\}$

- Taking expectation over pivot index $i$
  
  $T^{\text{exp}}(n) = \sum_{i=0}^{n-1} (\text{running time if pivot index is } i) P(\text{index of pivot is } i)$

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

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

- Same recurrence as for non-randomized average case
- Resolves to $\Theta(n)$ expected time on instance of size $n$
- Side note
  - there is selection algorithm “Median of Medians” (cs341) that has worst-case running time $O(n)$
    - uses double recursion
    - slower in practice
QuickSelect: Badly Designed Randomization

\[
\text{choose-random-pivot-badly}(A) = \\
\text{if } A.\text{size} \geq 3 \text{ return random}(3) \\
\text{else return 0}
\]

\[
T^{\text{exp}}(n) = \max_{\{I:\text{size}(I)=n\}} T^{\text{exp}}(I)
\]

- Worst instance is sorted array \( I_n = \{0, 1, ..., n-1\} \)
- \( T^{\text{exp}}(I_n) = \begin{cases} 
  cn + \frac{1}{3} T^{\text{exp}}(I_{n-1}) + \frac{1}{3} T^{\text{exp}}(I_{n-2}) + \frac{1}{3} T^{\text{exp}}(I_{n-3}) & \text{if } n \geq 3 \\
  c & \text{if } n < 3 
\end{cases} \)
- \( T^{\text{exp}}(I_n) \geq cn + T(I_{n-3}) \) if \( n \geq 3 \)
- Resolves to \( \Theta(n^2) \)
- Worst case expected time is \( \Theta(n^2) \)
Outline

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

- Hoare developed *partition* and *quick-select* in 1960
- He also used them to *sort* based on partitioning

```
quick-sort1(A)

Input: array A of size n

if n ≤ 1 then return
p ← choose-pivot1(A)
i ← partition (A, p)
quick-sort1(A[0, 1,...,i − 1])
quick-sort1(A[i + 1,...,n − 1])
```

- Let $T(n)$ to be the runtime on size $n$ array
- If we know pivot-index $i$, then $T(n) = cn + T(i) + T(n − i − 1)$
- Worst case $T(n) = T(n − 1) + cn$
  - recurrence solved in the same way as *quick-select1*, $\Theta(n^2)$
- Best case $T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + cn$
  - solved in the same way as *merge-sort*, $\Theta(n \log n)$
Average-case analysis of quick-sort1

- Make the same assumptions as for quick-select1
- Deriving recurrence equation is similar to quick-select1, but recurse on both sides

Using the same approach as for quick-select1, average running time is

\[
T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (cn + T(i) + T(n - i - 1)), \quad n \geq 2
\]

Running time is proportional to the number of comparisons

Recurrence for counting comparisons

\[
T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (n + T(i) + T(n - i - 1)), \quad n \geq 2
\]
Average-case analysis of quick-sort1

- First let us get a simpler recursive expression for $T(n)$

\[
T(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left( n + T(i) + T(n - i - 1) \right)
\]

\[
= n + \frac{1}{n} \sum_{i=0}^{n-1} T(i) + \frac{1}{n} \sum_{i=0}^{n-1} T(n - i - 1)
\]

\[
= n + \frac{2}{n} \sum_{i=0}^{n-1} T(i)
\]

- Thus \( T(n) = n + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \)
Average-case analysis of quick-sort

\[ T(n) = n + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \] is \( \Theta(n \log n) \)

Proof

Multiply by \( n \):
\[ nT(n) = n^2 + 2 \sum_{i=0}^{n-1} T(i) \]

Plug in \( n - 1 \):
\[ (n - 1)T(n - 1) = (n - 1)^2 + 2 \sum_{i=0}^{n-2} T(i) \]

Subtract:
\[ nT(n) - (n - 1)T(n - 1) = 2n - 1 + 2T(n - 1) \]

Rearrange:
\[ nT(n) = (n + 1)T(n - 1) + 2n - 1 \]

Divide by \( (n + 1)n \):
\[ \frac{T(n)}{n+1} = \frac{T(n - 1)}{n} + \frac{2n - 1}{n(n+1)} \]

Let \( A(n) = \frac{T(n)}{n+1} \):
\[ A(n) = A(n - 1) + \frac{2n - 1}{n(n+1)} = A(n - 2) + \frac{2(n - 1) - 1}{(n-1)n} + \frac{2n - 1}{n(n+1)} \]

\[ = \ldots = \sum_{i=1}^{n} \frac{2i - 1}{i(i + 1)} = \sum_{i=1}^{n} \frac{2}{i + 1} - \sum_{i=1}^{n} \frac{1}{i(i + 1)} \]

\[ \Theta(\log n) \quad \Theta(1) \]

Therefore:
\[ A(n) = c \log n \]

Finally:
\[ T(n) = (n + 1)A(n) = c(n + 1) \log n \in \Theta(n \log n) \]
Improvement ideas for QuickSort

- Randomize by using \textit{choose-pivot2}, giving $\Theta(n \log n)$ expected time for \textit{quick-sort2}

- The auxiliary space is $\Omega$(recursion depth)
  - $\Theta(n)$ in the worst-case
  - can be reduce to $\Theta(\log n)$ worst-case by
    - recurse in smaller sub-array first
    - replacing the other recursion by a while-loop (tail call elimination)

- Stop recursion when, say $n \leq 10$
  - array is not completely sorted, but almost sorted
  - at the end, run insertionSort, it sorts in just $O(n)$ time since all items are within 10 units of the required position

- Arrays with many duplicates sorted faster by changing \textit{partition} to produce three subsets $< v$ $= v$ $> v$

- Programming tricks
  - 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

\((l, r) \leftarrow S\.pop()\) // get the next subproblem

**while** \(r - l + 1 > 10\) // work on it if it’s larger than 10

\(p \leftarrow\) choose-pivot2\((A, l, r)\)

\(i \leftarrow\) partition \((A, l, r, p)\)

**if** \(i - l > r - i\) do // is left side larger than right?

\(S\.push((l, i - 1))\) // store larger problem in \(S\) for later

\(l \leftarrow i + 1\) // next work on the right side

**else**

\(S\.push((i + 1, r))\) // store larger problem in \(S\) for later

\(r \leftarrow i - 1\) // next work on the left side

**InsertionSort**\((A)\)

- This is often the most efficient sorting algorithm in practice
Outline

- 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

<table>
<thead>
<tr>
<th>Sort</th>
<th>Running Time</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$\Theta(n^2)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Theta(n^2)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Theta(n \log n)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\Theta(n \log n)$</td>
<td>worst-case</td>
</tr>
<tr>
<td>quick-sort1</td>
<td>$\Theta(n \log n)$</td>
<td>average-case</td>
</tr>
<tr>
<td>quick-sort2</td>
<td>$\Theta(n \log n)$</td>
<td>expected</td>
</tr>
</tbody>
</table>

**Question**: Can one do better than $\Theta(n \log n)$ running time?

**Answer**: *It depends on what we allow*

- No: comparison-based sorting lower bound is $\Omega(n \log n)$
  
  - no restriction on input, just must be able to compare

- Yes: non-comparison-based sorting can achieve $O(n)$
  
  - restrictions on input
The Comparison Model

- All sorting algorithms seen so far are in the comparison model.
- In the *comparison model* data can only be accessed in two ways:
  - comparing two elements
    - \( A[i] \leq A[j] \)
  - moving elements around (e.g. copying, swapping)
- This makes very few assumptions on the things we are sorting.
  - just count the number of above operations
- Under comparison model, will show that any sorting algorithm requires \( \Omega(n \log n) \) comparisons.
- This lower bound is not for an algorithm, it is for the sorting problem.
- How can we talk about problem without algorithm?
  - count number of comparisons any sorting algorithm has to perform.
Decision Tree

- Decision tree succinctly describes all the decisions that are taken during the execution of an algorithm and the resulting outcome.
- For each sorting algorithm we can construct a corresponding decision tree.
- Given decision tree, we can deduce the algorithm.
- Decision tree can be constructed for any algorithm, not just sorting.
Decision Tree Example

- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements \([x_0, x_1, x_2]\)

Set of all possible inputs

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Comparison</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2</td>
<td>(x_0 &lt; x_1 &lt; x_2)</td>
<td>([x_0, x_1, x_2])</td>
</tr>
<tr>
<td>0, 2, 1</td>
<td>(x_0 &lt; x_2 &lt; x_1)</td>
<td>([x_0, x_2, x_1])</td>
</tr>
<tr>
<td>1, 0, 2</td>
<td>(x_1 &lt; x_0 &lt; x_2)</td>
<td>([x_1, x_0, x_2])</td>
</tr>
<tr>
<td>1, 2, 0</td>
<td>(x_2 &lt; x_0 &lt; x_1)</td>
<td>([x_2, x_0, x_1])</td>
</tr>
<tr>
<td>2, 0, 1</td>
<td>(x_1 &lt; x_2 &lt; x_0)</td>
<td>([x_1, x_2, x_0])</td>
</tr>
<tr>
<td>2, 1, 0</td>
<td>(x_2 &lt; x_1 &lt; x_0)</td>
<td>([x_2, x_1, x_0])</td>
</tr>
</tbody>
</table>

- Have to determine which of the 6 inputs we are given before can give output
  - unique output for each distinct input
Decision Tree

- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements

.home
[0, 1, 2] 0, 1, 2
[0, 2, 1] 0, 2, 1
[1, 2, 0] 1, 2, 0

- Root corresponds to the set of all possible inputs
- Interior nodes are comparisons: each comparison splits the set of possible inputs into two
- Know correct sorting order only when the set of possible inputs shrinks to size one
  - nodes where possible input shrunk to size one are leaves, when reach them, can output sorting result
- Sorting algorithm will traverse a path starting at root and ending at a leaf
  - length of the path is the number of comparisons to be made
- Tree height is the number of comparisons required for sorting in the worst case
- Decision tree for a concrete comparison based sorting algorithm, with 3 non-repeating elements

- Algorithm could do more comparisons than necessary
- Thus can have more leafs than possible inputs
- But the number of leaves must be \textit{at least} the number of possible inputs
Decision Tree

- **Decision tree** for any comparison-based sorting algorithm, \( n \) non-repeating elements

\[
S = A \cup B
\]

- Tree must have at least \( n! \) leaves
- Binary tree with height \( h \) has at most \( 2^h \) leaves
- Height \( h \) must be at least such that \( 2^h \geq n! \)
- Tree height is the number of comparisons required in the worst case
Theorem: Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparisons.

Proof:

- There exists a set of $n!$ possible inputs such that each leads to a different output.
- Decision tree must have at least $n!$ leaves.
- Binary tree with height $h$ has at most $2^h$ leaves.
- Height $h$ must be at least such that $2^h \geq n!$.
- Taking logs of both sides:

$$h \geq \log(n!) = \log(n(n-1) \cdots 1) = \log n + \cdots + \log\left(\frac{n}{2} + 1\right) + \log\frac{n}{2} + \cdots + \log 1$$

$$\geq \log\frac{n}{2} + \cdots + \log\frac{n}{2} = \frac{n}{2} \log\frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2} \in \Omega(n \log n)$$
Outline

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

- Sort without comparing items to each other
- Non-comparison based sorting is less general than comparison based
- In particular, we need to make assumptions about items we sort
  - unlike in comparison based sorting, which sorts any data, as long as it can be compared
- Will assume we are sorting non-negative integers
  - can adapt to negative integers
  - also to some other data types, such as strings
  - but cannot sort arbitrary data
Non-Comparison-Based Sorting

- Simplest example
  - suppose all keys in $A$ are integers in range $[0, ..., L - 1]$
- For non-comparison sorting, running time depends on both
  - array size $n$
  - $L$
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, ..., L - 1]$
- Use an axillary bucket array $B[0, ..., L - 1]$ to sort
  - i.e. array of initially empty linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L - 1]$
- Use an auxiliary *bucket array* $B[0, \ldots, L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

$k = 0$

<table>
<thead>
<tr>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L - 1]$
- Use an auxiliary \textit{bucket array} $B[0, \ldots, L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

\[
\begin{array}{c|ccccccccccc}
 k = 1 & 12 & 14 & 7 & 6 & 7 & 0 & 10 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
 A & & & & & & & & & \textit{B} & & & & & \\
 & 12 & & & & & & & & 12 & & & & & \\
 & 14 & & & & & & & & 14 & & & & & \\
 & 7 & & & & & & & & \downarrow & & & & & \\
 & 6 & & & & & & & & \downarrow & & & & & \\
 & 7 & & & & & & & & \downarrow & & & & & \\
 & 0 & & & & & & & & \downarrow & & & & & \\
 & 10 & & & & & & & & \downarrow & & & & & \\
\end{array}
\]
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L - 1]$
- Use an auxiliary *bucket array* $B[0, \ldots, L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

![Diagram showing Bucket Sort example](image)
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, ..., L - 1]$
- Use an axillary \textit{bucket array} $B[0, ..., L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

\begin{align*}
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
A & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
12 & 14 & 7 & 6 & 0 & 10 & 6 & 7 & 12 & 14 & & & & & \\
\hline
\end{array}
\end{align*}
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, ..., L - 1]$
- Use an auxiliary *bucket array* $B[0, ..., L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

![Diagram of bucket sort example]

$A$

| 12 | 14 | 7 | 6 | 0 | 10 |

$k = 4$

$B$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

| 6 | 7 | 12 | 14 |
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L - 1]$
- Use an auxiliary bucket array $B[0, \ldots, L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

\[A\]

\[
\begin{array}{c}
12 \\
14 \\
7 \\
6 \\
7 \\
0 \\
10
\end{array}
\]

\[B\]

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
0 & 6 & 7 & 12 & 14
\end{array}
\]
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, \ldots, L - 1]$
- Use an axillary *bucket array* $B[0, \ldots, L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$

\[
\begin{array}{l}
A \hspace{1cm} B \\
\begin{array}{c}
12 \\
14 \\
7 \\
6 \\
7 \\
0 \\
k = 6 \\
10
\end{array} \\
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
& & & & & & & & & & & & & & \\
& & & & & 6 & 7 & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
\hline
0 & & & & & 6 & 7 & 10 & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
\hline
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
& & & & & & & & & & & & & & \\
\hline
\end{array}
\end{array}
\]
Bucket Sort

- Suppose all keys in $A$ are integers in range $[0, ..., L - 1]$
- Use an auxiliary bucket array $B[0, ..., L - 1]$ to sort
  - i.e. array of linked lists, initialization is $\Theta(L)$
- Example with $L = 15$
- Now iterate through $B$ and copy non-empty buckets to $A$

- Time complexity is $\Theta(L + n)$
  - $n$ is size of $A$
Digit Based Non-Comparison-Based Sorting

- Running time of bucket sort is $\Theta(L + n)$
  - $n$ is size of $A$
  - $L$ is range $[0, L)$ of integers in $A$

- What if $L$ is much larger than $n$?
  - i.e. $A$ has size 100, range of integers in $A$ is $[0, \ldots, 99999]$

- Assume at most $m$ digits in any key
  - pad with leading 0s

| 123  | 230  | 021  | 320  | 210  | 232  | 101  |

- Can sort ‘digit by digit’, can go
  - forward, from digit $1 \rightarrow m$ (more obvious)
  - backward, from digit $m \rightarrow 1$ (less obvious)
  - bucketsort is perfect for sorting ‘by digit’

- Example: $A$ has size 100, range of integers in $A$ is $[0, \ldots, 99999]$
  - integers have at most 5 digits, need only 5 iterations of bucketsort
Bucket Sort on Last Digit

- Equivalent to normal bucket sort if we redefine comparison
  - $a \leq b$ if the last digit of $a$ is smaller than (or equal) to the last digit of $b$

- Bucket sort is stable: equal items stay in original order
  - Crucial for developing LSD radix sort later
Base $R$ number representation

- Number of distinct digits gives the number of buckets $R$
- Useful to control number of buckets
  - larger $R$ means less digits (less iterations), but more work per iteration (larger bucket array)
  - may want exactly 2, or 4, or even 128 buckets
- Can do so with base $R$ representation
  - digits go from 0 to $R - 1$
  - $R$ buckets
  - numbers are in the range $\{0, 1, ..., R^m - 1\}$
- From now on, assume keys are numbers in base $R$ ($R$: radix)
  - $R = 2, 10, 128, 256$ are common
- Example ($R = 4$)

| 123 | 230 | 21 | 320 | 210 | 232 | 101 |
Single Digit Bucket Sort

**Bucket-sort**\((A, d)\)

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

initialize array \(B[0, ..., R - 1]\) of empty lists (buckets)

for \(i \leftarrow 0\) to \(n - 1\) do

\(next \leftarrow A[i]\)

append \(next\) at end of \(B[d]\)th digit of \(next\)

\(i \leftarrow 0\)

for \(j \leftarrow 0\) to \(R - 1\) do

while \(B[j]\) is non-empty do

move first element of \(B[j]\) to \(A[i + 1]\)

- Sorting is stable: equal items stay in original order
- Run-time \(\Theta(n + R)\)
- Auxiliary space \(\Theta(n + R)\)
  - \(\Theta(R)\) for array \(B\), and linked lists are \(\Theta(n)\)
Single Digit Bucket Sort

**Bucket-sorting**: 

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

1. Initialize array \(B[0] \ldots B[R-1]\) of empty lists (buckets).
2. For \(i \leftarrow 0\) to \(n-1\) do:
   - Next \(\leftarrow A[i]\)
   - Append `Next` at end of \(B[d] \text{th digit of } Next\)
3. For \(j \leftarrow 0\) to \(R-1\) do:
   - While \(B[j]\) is non-empty do:
     - Move first element of \(B[j]\) to \(A[i]+\)

- Sorting is stable:
- Run-time \(\Theta(n + R)\)
- Auxiliary space \(\Theta(n + R)\)
  - \(\Theta(R)\) for array \(B\), and linked lists are \(\Theta(n)\)
- Can replace lists by two auxiliary arrays of size \(R\) and \(n\), resulting in **count-sort**
  - no details
MSD-Radix-Sort

- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit

<table>
<thead>
<tr>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
</tr>
<tr>
<td>021</td>
</tr>
<tr>
<td>320</td>
</tr>
<tr>
<td>210</td>
</tr>
<tr>
<td>230</td>
</tr>
<tr>
<td>230</td>
</tr>
<tr>
<td>101</td>
</tr>
</tbody>
</table>
MSD-Radix-Sort

- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit

123
232
021
320
210
230
101
MSD-Radix-Sort

- Sorts multi-digit numbers from the most significant to the least significant
- Start by sorting the whole array by the first digit

- Cannot sort the whole array by the second digit, will mess up the order
- Have to break down in groups by the first digit
  - each group can be safely sorted by the second digit
  - call sort recursively on each group, with appropriate array bounds
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

```
<table>
<thead>
<tr>
<th>group 1</th>
<th>group 2</th>
<th>group 3</th>
<th>group 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>021</td>
<td>123</td>
<td>101</td>
<td>232</td>
</tr>
<tr>
<td>232</td>
<td>210</td>
<td>230</td>
<td>320</td>
</tr>
</tbody>
</table>

recursion depth 0
recursion depth 1
```
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

```
group 1
    021

group 2
    123
    101

<table>
<thead>
<tr>
<th>group 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
</tr>
<tr>
<td>210</td>
</tr>
<tr>
<td>230</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>group 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>320</td>
</tr>
</tbody>
</table>
```

recursion depth 0
recursion depth 1
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

```
  group 1
    021
    123
    101
    232
    210
    230

  group 2
    123
    101

  group 3

  group 4
    320
```
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group
**MSD-Radix-Sort**

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

- **Group 1**
  - 021
  - 123
  - 101
  - 232

- **Group 2**
  - 101
  - 123

- **Group 3**
  - 210
  - 230

- **Group 4**
  - 320

- **Recursion depth**
  - 0
  - 1
  - 2
### MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

<table>
<thead>
<tr>
<th>group 1</th>
<th>group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>021</td>
<td>123</td>
</tr>
<tr>
<td>101</td>
<td>232</td>
</tr>
<tr>
<td>210</td>
<td>230</td>
</tr>
<tr>
<td>230</td>
<td>320</td>
</tr>
</tbody>
</table>

- Grouping by leading digit:
  - 0: 021
  - 1: 123, 101
  - 2: 232, 210, 230, 320

- Recursion depth:
  - 0: 021
  - 1: 101, 123
  - 2: 123
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

```
021
123
101
232
210
230
320
```

```
021
101
123
```

```
021
101
123
```

```
021
101
123
```

recursion

depth 0

recursion

depth 1

recursion

depth 2
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

```
<table>
<thead>
<tr>
<th>Group 1</th>
<th>Group 2</th>
<th>Group 3</th>
<th>Group 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>021</td>
<td>123</td>
<td>101</td>
<td>230</td>
</tr>
<tr>
<td>123</td>
<td>101</td>
<td>232</td>
<td>230</td>
</tr>
<tr>
<td>232</td>
<td>101</td>
<td>210</td>
<td>320</td>
</tr>
<tr>
<td>230</td>
<td>123</td>
<td>210</td>
<td>230</td>
</tr>
</tbody>
</table>
```

Recursion depth: 0, 1, 2
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group
**MSD-Radix-Sort**

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

![Diagram of MSD-Radix-Sort algorithm]

- group 1
  - 021
  - 123
  - 101
  - 232
  - 210
  - 230
  - 320

- group 2
  - 021
  - 123
  - 101
  - 101
  - 123

- group 3
  - 210
  - 232
  - 230

- group 4
  - 210
  - 230

- recursion depth 0
- recursion depth 1
- recursion depth 2
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

![Diagram of MSD-Radix-Sort algorithm]

**group 1**
- 021
- 123
- 101
- 232
- 210
- 230
- 320

**group 2**
- 101
- 123
- 232
- 210
- 230

**group 3**
- 210
- 232
- 230

**group 4**
- 101
- 123
- 232

**Recursion depth:**
- 0
- 1
- 2
- 3
- 4
- 5
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group
**MSD-Radix-Sort**

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

![Diagram](image_url)
MSD-Radix-Sort

- Recursively sorts multi-digit numbers
  - sort by leading digit, group by next digit, then call sort recursively on each group

![Diagram of MSD-Radix-Sort process with recursion depths and groups]

- Group 1: 021, 123, 101, 232, 210, 230, 320
- Group 2: 101, 123
- Group 3: 210, 232, 230
- Group 4: 210, 230, 232

Recursion depth:
- 0: 021
- 1: 101, 123
- 2: 210, 230, 232
- 3: 320
- 4: 320
- 5: 320
- 6: 320
MSD-Radix-Sort Space Analysis

- Bucket-sort
  - auxiliary space $\Theta(n + R)$
- Recursion depth is $m - 1$
  - auxiliary space $\Theta(m)$
- Total auxiliary space $\Theta(n + R + m)$
MSD-Radix-Sort Time Analysis

- Time spent for each recursion depth
  - Depth 0
    - one bucket sort on \( n \) items
    - \( \Theta(n + R) \)
  - All other depths
    - lets \( k \) be the number of bucket sorts at each depth
      - \( k \leq n \)
        - cannot have more bucket sorts than the array size
    - each bucket sort is on \( n_i \) items
      - \( \sum_{i=0}^{k} n_i = n \)
    - each bucket sort is \( n_i + R \)
      - \( \sum_{i=0}^{k} (n_i+R) = n + \sum_{i=0}^{k} R \leq n + nR \)
      - total time at any depth is \( O(nR) \)
- Number of depths is at most \( m - 1 \)
- Total time \( O(mnR) \)
MSD-Radix-Sort Time Analysis

- Total time $O(mnR)$

- This is $O(n)$ if sort items in limited range
  - suppose $R = 2$, and we sort are $n$ integers in the range $[0, 2^{10})$
  - then $m = 10$, $R = 2$, and sorting is $O(n)$
    - note that $n$, the number of items to sort, can be arbitrarily large
MSD-Radix-Sort Time Analysis

- Total time $O(mnR)$
- This is $O(n)$ if sort items in limited range
  - suppose $R = 2$, and we sort are $n$ integers in the range $[0, 2^{10})$
  - then $m = 10$, $R = 2$, and sorting is $O(n)$
    - note that $n$, the number of items to sort, can be arbitrarily large
- This does not contradict $\Omega(n \log n)$ bound on the sorting problem, since the bound applies to comparison-based sorting
MSD-Radix-Sort Pseudocode

- Sorts array of \( m \)-digit radix-\( R \) numbers recursively
- Sort by leading digit, then each group by next digit, etc.

\[
\text{MSD-Radix-sort}(A, \ l \leftarrow 0, \ r \leftarrow n - 1, \ d \leftarrow \text{leading digit index}) \\
l, r: \text{indexes between which to sort, } 0 \leq l, r \leq n - 1 \\
\quad \text{if } l < r \\
\quad \quad \text{bucket-sort}(A[l \ldots r], \ d) \\
\quad \quad \text{if there are digits left} \\
\quad \quad \quad \ l' \leftarrow l \\
\quad \quad \quad \text{while } (l' < r) \ \text{do} \\
\quad \quad \quad \quad \text{let } r' \geq l' \text{ be the maximal s.t } A[l' \ldots r'] \text{ have the same } d \text{th digit} \\
\quad \quad \quad \quad \text{MSD-Radix-sort}(A, l', r', d + 1) \\
\quad \quad \quad \ l' \leftarrow r' + 1 \\
\]

- Run-time \( O(mnR) \)
- Auxiliary space is \( \Theta(m + n + R) \) for bucket sort and recursion stack
- Drawback of \textit{MSD-Radix-sort} is many recursions
LSD-Radix-Sort

- **Idea:** apply single digit bucket sort from least significant digit to the most significant digit
- Observe that digit bucket sort is stable
  - equal elements stay in the original order
  - therefore, we can apply single digit bucket sort to the **whole array**, and the output will be sorted after iterations over all digits
### LSD-Radix-Sort

<table>
<thead>
<tr>
<th></th>
<th>123</th>
<th>230</th>
<th>230</th>
<th>101</th>
<th>101</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>232</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **prepare to sort by last digit**
- **last digit sorted**
- **prepare to sort by middle digit**
- **last two digits sorted**
- **prepare to sort by first digit**
- **last three digits sorted**

- $m$ bucket sorts, on $n$ items each, one bucket sort is $\Theta(n + R)$
- Total time cost $\Theta(m(n + R))$
LSD-Radix-Sort

\[ \text{LSD-radix-sort}(A) \]
\[ A: \text{array of size } n, \text{ contains } m\text{-digit radix-}R \text{ numbers} \]

\[ \text{for } d \gets \text{least significant down to most significant digit do} \]
\[ \text{bucket-sort}(A, d) \]

- Loop invariant: after iteration \( i \), \( A \) is sorted w.r.t. the last \( i \) digits of each entry
- Time cost \( \Theta(m(n + R)) \)
- Auxiliary space \( \Theta(n + R) \)
Summary

- Sorting is an important and very well-studied problem.
- Can be done in $\Theta(n \log n)$ time.
  - Faster is not possible for general input.
- HeapSort is the only $\Theta(n \log n)$ time algorithm we have seen with $O(1)$ auxiliary space.
- MergeSort is also $\Theta(n \log n)$ time.
- Selection and insertion sorts are $\Theta(n^2)$.
- QuickSort is worst-case $\Theta(n^2)$, but often the fastest in practice.
- BucketSort and RadixSort can achieve $o(n \log n)$ if the input is special.
- Best-case, worst-case, average-case can all differ.
- Randomized algorithms can eliminate “bad cases”, resulting in the same expected time for all cases.