Sorting
Almost half of all CPU cycles are spent on sorting!

• **Input**: array X[1..n] of integers
• **Output**: sorted array (permutation of input)

In: 5,2,9,1,7,3,4,8,6
Out: 1,2,3,4,5,6,7,8,9

• Assume WLOG all input numbers are unique
• Decision tree model ⇒ count comparisons “<”
Lower Bound for Sorting

**Theorem**: Sorting requires $\Omega(n \log n)$ time.

**Proof**: Assume WLOG unique numbers

$\Rightarrow n!$ different permutations

$\Rightarrow$ comparison decision tree has $n!$ leaves

$\Rightarrow$ tree height $\geq \log(n!) > \log\left(\left(\frac{n}{e}\right)^n\right) = n \cdot \log\left(\frac{n}{e}\right) = \Omega(n \log n)$

$\Rightarrow \Omega(n \log n)$ decisions / time necessary to sort
Sorting Algorithms (Sorted!)

1. AKS sort 17. Franceschini's sort 33. Selection sort
2. Bead sort 18. Gnome sort 34. Shaker sort
5. Block sort 21. Insertion sort 37. Sleep sort
11. Cocktail sort 27. Pigeonhole sort 43. Stooge sort
12. Comb sort 28. Postman sort 44. Strand sort
13. Counting sort 29. Quantum sort 45. Timsort
15. Cycle sort 31. Radix Sort 47. Tournament sort
16. Flashsort 32. Sample sort 48. UnShuffle Sort
Q: Why so many sorting algorithms?
A: There is no “best” sorting algorithm!

Some considerations:

- **Worst** case?
- **Average** case?
- In practice?
- Input **distribution**?
- Near-sorted data?
- **Stability**?
- In-situ?
- Randomized?
- Stack depth?
- Internal vs. **external**?
- Pipeline compatible?
- **Parallelizable**?
- Locality?
- **Online**
Problem: Given $n$ pairs of integers $(x_i, y_i)$, where $0 \leq x_i \leq n$ and $1 \leq y_i \leq n$ for $1 \leq i \leq n$, find an algorithm that sorts all $n$ ratios $x_i / y_i$ in linear time $O(n)$.

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations
Problem: Given $n$ integers, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations
Problem: Given $n$ objects, find in $O(n)$ time the majority element (i.e., occurring $\geq n/2$ times, if any), using only equality comparisons ($=$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations
Problem: Given $n$ integers, find both the \textbf{maximum} and the \textbf{next-to-maximum} using the least number of \textbf{comparisons} (\textbf{exact} comparison count, not just $O(n)$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations
Bubble Sort

Input: array X[1..n] of integers
Output: sorted array (monotonic permutation)

Idea: keep swapping adjacent pairs

until array X is sorted do
  for i=1 to n-1
    if X[i+1]<X[i]
      then swap(X,i,i+1)

• O(n^2) time worst-case,
  but sometimes faster
• Adaptive, stable, in-situ, slow
Odd-Even Sort

Input: array $X[1..n]$ of integers
Output: sorted array (monotonic)

Idea: swap even and odd pairs

until array $X$ is sorted do
  for even $i=1$ to $n-1$
    if $X[i+1]<X[i]$ swap($X,i,i+1$)
  for odd $i=1$ to $n-1$
    if $X[i+1]<X[i]$ swap($X,i,i+1$)

• $O(n^2)$ time worst-case, but faster on near-sorted data
• Adaptive, stable, in-situ, parallel
Selection Sort

Input: array \( X[1..n] \) of integers

Output: sorted array (monotonic permutation)

Idea: move the largest to current pos

for \( i=1 \) to \( n-1 \)
   let \( X[j] \) be largest among \( X[i..n] \)
   swap(\( X, i, j \))

- \( \Theta(n^2) \) time worst-case
- Stable, in-situ, simple, not adaptive
- Relatively fast (among quadratic sorts)
**Insertion Sort**

- **Input**: array $X[1..n]$ of integers
- **Output**: sorted array (monotonic permutation)

**Idea**: insert each item into list

```plaintext
for i=2 to n
    insert $X[i]$ into the sorted list $X[1..(i-1)]$
```

- $O(n^2)$ time worst-case
- $O(nk)$ where $k$ is max dist of any item from final sorted pos
- **Adaptive, stable, in-situ, online**
Heap Sort

**Input:** array $X[1..n]$ of integers  
**Output:** sorted array (monotonic)

**Idea:** exploit a heap to sort

```
InitializeHeap
For i=1 to n HeapInsert(X[i])
For i=1 to n do
    M=HeapMax; Print(M)
    HeapDelete(M)
```

- $\Theta(n \log n)$ optimal time
- **Not** stable, not adaptive, in-situ
SmoothSort

**Input:** array X[1..n] of integers

**Output:** sorted array (monotone)

**Idea:** adaptive heapsort

InitializeHeaps
for i=1 to n HeapsInsert(X[i])
for i=1 to n do
    M=HeapsMax; Print(M)
    HeapsDelete(M)

- Uses multiple (Leonardo) heaps
- O(n log n)
- O(n) if list is mostly sorted
- Not stable, adaptive, in-situ
Historical Perspectives

Edsger W. Dijkstra (1930-2002)

• Pioneered software engineering, OS design
• Invented concurrent programming, mutual exclusion / semaphores
• Invented shortest paths algorithm
• Advocated structured (GOTO-less) code
• Stressed elegance & simplicity in design
• Won Turing Award in 1972
Quotes by Edsger W. Dijkstra (1930-2002)

• “Computer science is no more about computers than astronomy is about telescopes.”

• “If debugging is the process of removing software bugs, then programming must be the process of putting them in.”

• “Testing shows the presence, not the absence of bugs.”

• “Simplicity is prerequisite for reliability.”

• “The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.”

• “Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

• “Elegance has the disadvantage, if that's what it is, that hard work is needed to achieve it and a good education to appreciate it.”
Generalizing Heap Sort

**Input:** array \(X[1..n]\) of integers

**Output:** sorted array

```
InitializeTree
For i=1 to n
    TreeInsert(X[i])
For i=1 to n do
    M=TreeMax; Print(M)
    TreeDelete(M)
```

- **Observation:** other data structures can work here!
- **Ex:** replace heap with any height-balanced tree
- **Retains** \(O(n \log n)\) worst-case time!
**Tree Sort**

**Input:** array $X[1..n]$ of integers  
**Output:** sorted array (monotonic)

**Idea:** populate a tree & traverse  

```
InitializeTree
for i=1 to n TreeInsert(X[i])
traverse tree in-order  
    to produce sorted list
```

- Use balanced tree (AVL, B, 2-3, splay)
- $O(n \log n)$ time worst-case
- Faster for near-sorted inputs
- **Stable, adaptive, simple**
B-Tree Sort

- Multi-rotations occur infrequently
- Rotations don’t propagate far
- Larger tree $\Rightarrow$ fewer rotations
- Same for other height-balanced trees
- Non-balanced search trees average $O(\log n)$ height
AVL-Tree Sort

- Multi-rotations occur infrequently
- Rotations don’t propagate far
- Larger tree $\Rightarrow$ fewer rotations
- Same for other height-balanced trees
- Non-balanced trees average $O(\log n)$ height
Merge Sort

Input: array $X[1..n]$ of integers
Output: sorted array (monotonic)

Idea: sort sublists & merge them

$$\text{MergeSort}(X,i,j)$$

if $i < j$ then $m = \lfloor (i+j)/2 \rfloor$
$$\text{MergeSort}(X,i..m)$$
$$\text{MergeSort}(X,m+1..j)$$
$$\text{Merge}(X,i..m,m+1..j)$$

• $T(n) = 2T(n/2) + n = \Theta(n \log n)$ optimal!
• Stable, parallelizes, not in-situ
• Can be made in-situ & stable
Theorem: MergeSort runs within time \( \Theta(n \log n) \) which is optimal.

Proof: Even-split divide & conquer:

\[
T(n) = 2 \cdot T(n/2) + n
\]

Total time is \( O(n \log n) \); \( \Omega(n \log n) \) \( \Rightarrow \) \( \Theta(n \log n) \)
**Quicksort**

**Input:** array $X[1..n]$ of integers  
**Output:** sorted array (monotonic)  

**Idea:** sort two sublists around pivot  

```
QuickSort(X,i,j)
    If i<j Then p=Partition(X,i,j)
    QuickSort(X,i,p)
    QuickSort(X,p+1,j)
```

- $\Theta(n \log n)$ time average-case  
- $\Theta(n^2)$ worst-case time (rare)  
- **Unstable**, parallelizes, $O(\log n)$ space  
- Ave: only beats $\Theta(n^2)$ sorts for $n>40$
Shell Sort

**Input:** array X[1..n] of integers

**Output:** sorted array (monotonic)

**Idea:** generalize insertion sort

- for each \( h_i \) in sequence \( h_k, \ldots, h_1 = 1 \)
- Insertion-sort all items \( h_i \) apart

- Array is sorted after last pass (\( h_i = 1 \))
- Long swaps quickly reduce disorder
- \( O(n^2), O(n^{3/2}), O(n^{4/3}), \ldots \) ?
- Complexity still open problem!
- LB is \( \Omega(N(\log/\log \log n)^2) \)
- Not stable, adaptive, in-situ
Counting Sort

**Input:** array \( X[1..n] \) of integers in small range \( 1..k \)

**Output:** sorted array (monotonic)

**Idea:** use values as array indices

```
for i=1 to k do C[i] = 0
for i=1 to n do C[X[i]]++
for i=1 to k do if C[i] ≠ 0 then print(i) C[i] times
```

- \( \Theta(n) \) time, \( \Theta(k) \) space
- Not comparison-based
- For specialized data only
- Stable, parallel, not in-situ
Q: Why not use counting sort for arbitrary 32-bit integers? (i.e., range k is “fixed”)

A: Range is fixed ($2^{32}$) but very large (4,294,967,296). Space/time: the counts array will be huge (4 GB)

Much worse for 64-bit integers ($2^{64} > 10^{19}$):

Time: 5 GHz PC will take over $2^{64} / (5 \cdot 10^9) / (60 \cdot 60 \cdot 24 \cdot 365)$ sec > 116 years to initialize array!

Memory: $2^{64} > 10^{19} > 18$ Exabytes > 2.3 million TB RAM chips! > total amount of Google’s data!

Q: What’s an Exabyte? ($10^{18}$)
What does an Exabyte look like?
What does an Exabyte look like?
What does an Exabyte look like?
What does an Exabyte look like?
What does an Exabyte look like?
What does an Exabyte look like?

- All content of Library of Congress: ~ 0.001 Exabytes
- Total words ever spoken by humans: ~ 5 Exabytes
- Total data stored by Google: ~ 15 Exabytes
- Total monthly world internet traffic: ~ 110 Exabytes
- Storage capacity of 1 gram of DNA: ~ 455 Exabytes
### Orders-of-Magnitude

**Standard International (SI) quantities:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deca</td>
<td>$10^1$</td>
</tr>
<tr>
<td>Hecto</td>
<td>$10^2$</td>
</tr>
<tr>
<td>Kilo</td>
<td>$10^3$</td>
</tr>
<tr>
<td>Mega</td>
<td>$10^6$</td>
</tr>
<tr>
<td>Giga</td>
<td>$10^9$</td>
</tr>
<tr>
<td>Tera</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>Peta</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>Exa</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>Zetta</td>
<td>$10^{21}$</td>
</tr>
<tr>
<td>Yotta</td>
<td>$10^{24}$</td>
</tr>
<tr>
<td>Deci</td>
<td>$10^{-1}$</td>
</tr>
<tr>
<td>Centi</td>
<td>$10^{-2}$</td>
</tr>
<tr>
<td>Milli</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Micro</td>
<td>$10^{-6}$</td>
</tr>
<tr>
<td>Nano</td>
<td>$10^{-9}$</td>
</tr>
<tr>
<td>Pico</td>
<td>$10^{-12}$</td>
</tr>
<tr>
<td>Femto</td>
<td>$10^{-15}$</td>
</tr>
<tr>
<td>Atto</td>
<td>$10^{-18}$</td>
</tr>
<tr>
<td>Zepto</td>
<td>$10^{-21}$</td>
</tr>
<tr>
<td>Yocto</td>
<td>$10^{-24}$</td>
</tr>
</tbody>
</table>
Orders-of-Magnitude

• “Powers of Ten”, Charles and Ray Eames, 1977
Orders-of-Magnitude

- “Scale of the Universe”, Cary and Michael Huang, 2012

• $10^{-24}$ to $10^{26}$ meters $\Rightarrow$ 50 orders of magnitude!
Bucket Sort

**Input:** array $X[1..n]$ of real numbers in $[0,1]$

**Output:** sorted array (monotonic)

**Idea:** spread data among buckets

```plaintext
for i=1 to n do
    insert $X[i]$ into bucket $\lfloor n \cdot X[i] \rfloor$
for i=1 to n do Sort bucket i
concatenate all the buckets
```

- $O(n+k)$ time expected, $O(n)$ space
- $O($Sort$)$ time worst-case
- Assumes substantial data uniformity
- Stable, parallel, not in-situ
- Generalizes counting sort / quicksort
Bucket Sort

Q: How does bucket sort generalize counting sort? Quicksort?
**Radix Sort**

**Input**: array $X[1..n]$ of integers each with $d$ digits in range $1..k$

**Output**: sorted array (monotonic)

**Idea**: sort each digit in turn

For $i=1$ to $d$ do

StableSort($X$ on digit $i$)

- Makes $d$ calls to bucket sort
- $\Theta(d \cdot n)$ time, $\Theta(k+n)$ space
- Not comparison-based
- Stable
- Parallel
- Not in-situ
Radix Sort

| 6428 | 4754 | 9650 | 5650 | 9843 | 7118 | 8804 | 3871 | 6592 | 1163 | 2899 | 9602 |

Q: is Radix Sort faster than Merge Sort? $\Theta(d \cdot n)$ vs. $\Theta(n \log n)$
• $O(n \log n)$ sorts tend to beat the $O(n^2)$ sorts ($n>50$)
• Some sorts work faster on random data vs. near-sorted data
• For more details see http://www.sorting-algorithms.com
Q: how can we easily modify quicksort to have $O(n \log n)$ worst-case time?

Idea: combine two algorithms to leverage the best behaviors of each one.

```
MetaSort(X,i,j):
  parallel-run:
    • QuickSort(X,i,j)
    • MergeSort(X,i,j)
  when either stops, abort the other
```

- Ave-case time is $\text{Min}$ of both: $O(n \log n)$
- Worst-case time is $\text{Min}$ of both: $O(n \log n)$
- Meta-algorithms / meta-heuristics generalize!
“The Sound of Sorting” (15 algorithms)

- Sound **pitch** is proportional to **value** of current sort element sorted!

https://www.youtube.com/watch?v=kPRA0W1kECg
Finding the Minimum
Finding the Minimum

Input: array X[1..n] of integers
Output: minimum element

Theorem: \( \Omega(n) \) time is necessary to find Min.

Proof 1: each element must be examined at least once, otherwise we may miss the true minimum. Therefore \( \Omega(n) \) work is required.

Proof 2: Assume a correct min-finding algorithm didn’t examine element \( X_i \) for some array \( X \). Then the same algorithm will be wrong on \( X \) with \( X_i \) replaced with say -10^{100}.
Finding the Minimum

Every input must be examined linear time lower bound.
Finding the Minimum

Input: array $X[1..n]$ of integers
Output: minimum element

Idea: keep track of the best-so-far

$$\text{Min} = X[1]$$

for $i = 2$ to $n$

if $X[i] < \text{min}$ then $	ext{min} = X[i]$

• Exact comparison count: $n-1$

Theorem: $n-1$ comparisons are sufficient for finding the minimum.

Corollary: This $\Theta(n)$-time algorithm is optimal.

Q: What about finding the maximum?
Finding the Minimum

Q: Can we do better than \( n-1 \) comparisons?

Theorem: \( n/2 \) comparisons are necessary for finding the minimum.

Idea: must examine all \( n \) inputs!

Proof: each element must participate in at least 1 comparison (otherwise we may miss e.g. \(-10^{100}\)).

- Each comparison involves 2 elements
- At least \( n/2 \) comparisons are necessary

Q: Can we improve lower bound up to \( n-1 \)?
Theorem: \( n-1 \) comparisons are necessary for finding the minimum (or maximum).

Idea: keep track of “knowledge” gained!

Proof: consider two classes of elements:

- At each comparison, at most 1 element moves from “unknown” to “won (Min)”.
- At least \( n-1 \) moves/comparisons are necessary to convert the initial state into the final state.

Corollary: The \((n-1)\)-comparison algorithm is optimal.

Every non-Min element must win at least once!
Finding the Min and Max

**Input:** array \( X[1..n] \) of integers

**Output:** minimum and maximum elements

**Idea:** find Min independently from Max

\[
\text{FindMin}(X) \\
\text{FindMax}(X) \equiv \text{FindMin}(-X)
\]

- \( n-1 \) comparisons to find Min
- \( n-1 \) comparisons to find Max
- Total \( 2n-2 \) comparisons needed

**Observation:** much information is discarded!

**Q:** Can we do better than \( 2n-2 \) comparisons?
Finding the Min and Max

**Input:** array $X[1..n]$ of integers  
**Output:** minimum and maximum elements  
**Idea:** pairwise compare to reduce work

**Theorem:** $3n/2 - 2$ comparisons are sufficient for finding the minimum and maximum.
Finding the Min and Max

Theorem: $3n/2 - 2$ comparisons are necessary for finding the minimum and maximum.

Idea: keep track of “knowledge” gained!

Proof: consider four classes of elements:

- not tested
- only won
- only lost
- won & lost

Initial state:
- not tested $n$
- only won $0$
- only lost $0$
- won & lost $0$

Final state:
- not tested $0$
- only won $1$
- only lost $1$
- won & lost $n-2$

Min & Max
## Finding the Min and Max

### Minimum guaranteed knowledge gained i.e. “moves” towards final state

<table>
<thead>
<tr>
<th>Condition</th>
<th>Next State</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>N &lt; N ⇒ L &amp; W</td>
<td>N &gt; N ⇒ W &amp; L</td>
<td>2</td>
</tr>
<tr>
<td>N &lt; W ⇒ L &amp; W</td>
<td>N &gt; W ⇒ W &amp; L</td>
<td>1</td>
</tr>
<tr>
<td>N &lt; L ⇒ L &amp; B</td>
<td>N &gt; L ⇒ W &amp; L</td>
<td>1</td>
</tr>
<tr>
<td>N &lt; B ⇒ L &amp; B</td>
<td>N &gt; B ⇒ W &amp; L</td>
<td>1</td>
</tr>
<tr>
<td>W &lt; W ⇒ B &amp; W</td>
<td>W &gt; W ⇒ W &amp; B</td>
<td>1</td>
</tr>
<tr>
<td>W &lt; L ⇒ B &amp; B</td>
<td>W &gt; L ⇒ W &amp; L</td>
<td>0</td>
</tr>
<tr>
<td>W &lt; B ⇒ B &amp; B</td>
<td>W &gt; B ⇒ W &amp; B</td>
<td>0</td>
</tr>
<tr>
<td>L &lt; L ⇒ L &amp; B</td>
<td>L &gt; L ⇒ B &amp; L</td>
<td>1</td>
</tr>
<tr>
<td>L &lt; B ⇒ L &amp; B</td>
<td>L &gt; B ⇒ B &amp; B</td>
<td>0</td>
</tr>
<tr>
<td>B &lt; B ⇒ B &amp; B</td>
<td>B &gt; B ⇒ B &amp; B</td>
<td>0</td>
</tr>
</tbody>
</table>
Finding the Min and Max

• Moving from N to B forces passing through W or L
• Emptying N into W & L takes n/2 comparisons
• Emptying most of W takes n/2-1 comparisons
• Emptying most of L takes n/2-1 comparisons
• Other moves will not reach the “final state” any faster
• Total comparisons required: 3n/2 - 2
  \( \Rightarrow 3n/2 - 2 \) comparisons are necessary for finding the minimum and maximum.

Theorem: Our Min&Max algorithm is optimal.
Problem: Given $n$ integers, find both the maximum and the next-to-maximum using the least number of comparisons (exact comparison count, not just $O(n)$).

- What approaches fail?
- What techniques work and why?
- Lessons and generalizations
Finding the Max and Next-to-Max

Theorem: \((n-2) + \log n\) comparisons are sufficient for finding the maximum and next-to-maximum.

Proof: consider elimination tournament:

Theorem: \((n-2) + \log n\) comparisons are necessary for finding the maximum and next-to-maximum.
Selection (Order Statistics)

**Input:** array \(X[1..n]\) of integers and \(i\)

**Output:** \(i^{th}\) largest integer

**Obvious:** \(i^{th}\)-largest subroutine can find median since median is the special case \((n/2)^{th}\)-largest

**Not obvious:** repeat medians can find \(i^{th}\) largest:

**Two cases:**

- \(i < n/2 \Rightarrow \text{find } i^{th}\) largest
- \(i > n/2 \Rightarrow \text{find } (i-n/2)^{th}\) largest
Selection (Order Statistics)

Run time for $i^{\text{th}}$ largest: $T(n) = T(n/2) + M(n)$

where $M(n)$ is time to find median

• Finding median in $O(n \log n)$ time is easy (why?)
• Assume $M(n) = c \cdot n = O(n)$
  \[ \Rightarrow T(n) < c \cdot (n + n/2 + n/4 + n/8 + \ldots) \]
  \[ < c \cdot (2n) = O(n) \]

Conclusion: linear-time median algorithm automatically yields linear-time $i^{\text{th}}$ selection!

New goal: find the median in $O(n)$ time!

Recurse!

\[ i < n/2 \Rightarrow \text{find } i^{\text{th}} \text{ largest} \quad \text{or} \quad i > n/2 \Rightarrow \text{find } (i-n/2)^{\text{th}} \text{ largest} \]
**QuickSelect** (i\textsuperscript{th}-Largest)

Idea: partition around pivot and recurse

\[
X: \begin{array}{cccccccc}
p & p+1 & \ldots & q & q+1 & \ldots & r-1 & r \\
\end{array}
\]

- \(k = q - p + 1\) elements
- \(r - q\) elements

\[i < k \Rightarrow \text{QuickSelect } i\text{th largest} \quad \text{or} \quad i > k \Rightarrow \text{QuickSelect } (i-k)\text{th largest}\]

**QuickSelect**\((X,p,r,i)\)

- if \(p == r\) then return\((X[p])\)
- \(q = \text{RandomPartition}(X,p,r)\)
- \(k = q - p + 1\)
- If \(i \leq k\) then return\((\text{QuickSelect}(X,p,q,i))\)
- else return\((\text{QuickSelect}(X,q+1,r,i-k))\)

- \(O(n)\) time average-case (analysis like QuickSort’s)
- \(\Theta(n^2)\) worst-case time (very rare)
**Median in Linear Time**

**Idea:** quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]

- Partition into $n/5$ groups of 5 each
- Sort each group (high to low)
- Compute median of medians (recursively)
- Move columns with larger medians to right
- Move columns with smaller medians to left
Median in Linear Time

Idea: quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]

- > 3/10 of elements larger than median of medians
- > 3/10 of elements smaller than median of medians
- Partition all elements around median of medians
- Each partition contains at most $7n/10$ elements
- Recurse on the proper partition (like in QuickSelect)
Median in Linear Time

Idea: quickly eliminate a constant fraction & repeat

[Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]

\[ T(n) = T(n/5) + T(7n/10) + O(n) \]
\[ = T(2n/10) + T(7n/10) + O(n) \]
\[ \leq T(2n/10 + 7n/10) + O(n) \text{ since } T(n) = \Omega(n) \]
\[ = T(9n/10) + O(n) \Rightarrow T(n) = O(n) \]

• Median is found in \( \Theta(n) \) time worst-case!
Median selection in $O(n)$ time worst-case.

Exact upper bounds: $< 24n, 5.4n, 3n, 2.95n, \ldots + o(n)$

Exact lower bounds: $> 1.5n, 1.75n, 1.8n, 1.837n, 2n, \ldots + O(1)$

Closing this comparisons gap further is still an open problem!