Unit 2: Sorting and Order Statistics

- **Course contents:**
  - Heapsort
  - Quicksort
  - Sorting in linear time
  - Order statistics

- **Readings:**
  - Chapters 6, 7, 8, 9

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best case</td>
<td>Average case</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Counting</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
</tr>
<tr>
<td>Radix</td>
<td>$O(d(n + k'))$</td>
<td>$O(d(n + k'))$</td>
</tr>
<tr>
<td>Bucket</td>
<td>–</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Types of Sorting Algorithms

• A sorter is **in-place** if only a constant # of elements of the input are ever stored outside the array.

• A sorter is **comparison-based** if the only operation on keys is to **compare two keys**.
  - Insertion sort, merge sort, heapsort, quicksort

• The non-comparison-based sorters sort keys by looking at the values of **individual** elements.
  - **Counting sort**: Assumes keys are in [1..k] and uses array indexing to count the # of elements of each value.
  - **Radix sort**: Assumes each integer contains $d$ digits, and each digit is in [1..$k'$].
  - **Bucket sort**: Sort data into buckets and then merge across buckets. Requires information for input **distribution**.
# Sorting Algorithm Comparisons

## Comparison-based sorters

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best case</td>
<td>Average case</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg n)$</td>
</tr>
</tbody>
</table>

## Non-comparison-based sorters

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Runtime</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best case</td>
<td>Average case</td>
</tr>
<tr>
<td>Counting</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
</tr>
<tr>
<td>Radix</td>
<td>$O(d(n + k'))$</td>
<td>$O(d(n + k'))$</td>
</tr>
<tr>
<td>Bucket</td>
<td>—</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* Unstable, in-place radix sort can be implemented
Binary Heap

- Binary heap data structure: represented by an array $A$
  - Complete binary tree, except that some rightmost leaves on the bottom level may be missing.
  - **Max-Heap property:** A node's key $\geq$ its children's keys.
  - **Min-Heap property:** A node's key $\leq$ its children's keys.

- Implementation
  - $A$.heap-size (# of elements in the heap stored within $A$) $\leq A$.length (# of elements in $A$).
MAX-HEAPIFY: Maintaining the Heap Property

- Assume that subtrees indexed $\text{RIGHT}(i)$ and $\text{LEFT}(i)$ are heaps, but $A[i]$ may be smaller than its children.
- MAX-HEAPIFY($A$, $i$) will “float down” the value at $A[i]$ so that the subtree rooted at $A[i]$ becomes a heap.
MAX-HEAPIFY: Complexity

MAX-HEAPIFY(A, i)
1. \( l = \text{LEFT}(i) \)
2. \( r = \text{RIGHT}(i) \)
3. \( \text{if } l \leq A.\text{heap-size} \text{ and } A[l] > A[i] \)
4. \( \text{largest} = l \)
5. \( \text{else } \text{largest} = i \)
6. \( \text{if } r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}] \)
7. \( \text{largest} = r \)
8. \( \text{if } \text{largest} \neq i \)
9. \( \text{exchange } A[i] \text{ with } A[\text{largest}] \)
10. \( \text{MAX-HEAPIFY}(A, \text{largest}) \)

- Worst case: last row of binary tree is half empty \( \Rightarrow \) children's subtrees have size \( \leq 2n/3 \).
- Recurrence: \( T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log n) \)
BUILD-MAX-HEAP: Building a Max-Heap

- Intuition: Use MAX-HEAPIFY in a bottom-up manner to convert $A$ into a heap.
  - Leaves are already heaps, start at parents of leaves, and work upward till the root.
**BUILD-MAX-HEAP: Complexity**

**BUILD-MAX-HEAP(A)**
1. A.heap-size = A.length
2. for \(i = \lceil A.length/2 \rceil\) downto 1
3. MAX-HEAPIFY(A, i)

- Naive analysis: \(O(n \lg n)\) time in total.
  - About \(n/2\) calls to HEAPIFY.
  - Each takes \(O(\lg n)\) time.

- Careful analysis: \(O(n)\) time in total.
  - Each MAX-HEAPIFY takes \(O(h)\) time (\(h\): tree height).
  - At most \(\lceil n/2^{h+1} \rceil\) nodes of height \(h\) in an \(n\)-element array.
  - \(T(n) = \sum_{h=0}^{\lceil \lg n \rceil} (#\text{nodes in height } h)O(h) = \sum_{h=0}^{\lceil \lg n \rceil} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^n}) = O(n)\)
  - Note: (1) cf. height & depth, (2) Won't improve the overall complexity of the heap sort.
Tree Height and Depth

- **Height** of a node: # of edges on the longest simple downward path from the node to a leaf
- **Depth**: Length of the path from the root to a node

<table>
<thead>
<tr>
<th>Node</th>
<th>Height</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

# of nodes with height 0 = 5
HEAPSORT

HEAPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for $i = A.\text{length}$ downto 2
4. $A.\text{heap-size} = A.\text{heap-size} - 1$
5. MAX-HEAPIFY(A,1)

- Time complexity: $O(n \log n)$.
- Space complexity: $O(n)$ for array, in-place (stable?).
Priority Queues

- A priority queue is a data structure on sets of keys; a max-priority queue supports the following operations:
  - INSERT(S, x): insert x into set S.
  - MAXIMUM(S): return the largest key in S.
  - EXTRACT-MAX(S): return and remove the largest key in S.
  - INCREASE-KEY(S, x, k): increase the value of element x’s key to the new value k.

- These operations can be easily supported using a heap.
  - INSERT: Insert the node at the end and fix heap in $O(\lg n)$ time.
  - MAXIMUM: read the first element in $O(1)$ time.
  - INCREASE-KEY: traverse a path from the target node toward the root to find a proper place for the new key in $O(\lg n)$ time.
  - EXTRACT-MAX: delete the 1st element, replace it with the last, decrement the element counter, then heapify in $O(\lg n)$ time.

- Compare with an array?
**Heap: EXTRACT-MAX and INSERT**

---

### HEAP-EXTRACT-MAX(A)
1. **if** `A.heap-size < 1`
2. **error** “heap underflow”
3. `max = A[1]`
5. `A.heap-size = A.heap-size - 1`
6. MAX-HEAPIFY(A, 1)
7. **return** `max`

---

### MAX-HEAP-INSERT(A, key)
1. `A.heap-size = A.heap-size + 1`
2. `i = A.heap-size`
3. **while** `i > 1` and `A[PARENT(i)] < key`
5. `i = PARENT(i)`
6. `A[i] = key`
Quicksort

• A divide-and-conquer algorithm
  - **Conquer**: Recursively sort two subarrays.
  - **Combine**: Do nothing; quicksort is an in-place algorithm.

```plaintext
QUICKSORT(A, p, r)
// Call QUICKSORT(A, 1, A.length) to sort an entire array
1. if p < r
2.   q = PARTITION(A, p, r)
3.   QUICKSORT(A, p, q)
4.   QUICKSORT(A, q+1, r)
```
Quicksort: Hoare Partition

PARTITION(A, p, r)
1. \( x = A[p] \) // break up \( A \) wrt \( x \)
2. \( i = p - 1 \)
3. \( j = r + 1 \)
4. **while** TRUE
5. \( \text{repeat } j = j - 1 \)
6. \( \text{until } A[j] \leq x \)
7. \( \text{repeat } i = i + 1 \)
8. \( \text{until } A[i] \geq x \)
9. **if** \( i < j \)
11. **else** return \( j \)

- Partition \( A \) into two subarrays \( A[j] \leq x \) and \( A[i] \geq x \).
- PARTITION runs in \( \Theta(n) \) time, where \( n = r - p + 1 \).
- Ways to pick \( x \): always pick \( A[p] \), pick a key at random, pick the median of several keys, etc.
- There are several partitioning variants
Quick Sort Example

Divide (Partition) & Conquer (Sort)

Combine (do nothing)
Quicksort Runtime Analysis: Best Case

• A divide-and-conquer algorithm
  \[ T(n) = T(q - p + 1) + T(r - q) + \theta(n) \]
  – Depends on the position of \( q \) in \( A[p..r] \), but ???

• Best-, worst-, average-case analyses?

• **Best case**: Perfectly balanced splits---each partition gives an \( n/2 : n/2 \) split.
  \[ T(n) = T(n/2)+T(n/2) + \theta(n) \]
  \[ = 2T(n/2) + \theta(n) \]

• Time complexity: \( \theta(n \lg n) \)
  – Master method? Iteration? Substitution?
Quicksort Runtime Analysis: Worst Case

- **Worst case:** Each partition gives a 1 : n - 1 split.

\[ T(n) = T(1) + T(n-1) + \theta(n) \]
\[ = T(1) + (T(1) + T(n-2) + \theta(n-1)) + \theta(n) \]
\[ = \ldots \]
\[ = nT(1) + \Theta \left( \sum_{k=1}^{n} k \right) \]
\[ = \theta(n^2) \]
More on Worst-Case Analysis

• The real upperbound:
  \[ T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q) + \theta(n)) \]

• Guess \( T(n) \leq cn^2 \) and verify it inductively:
  \[ T(n) \leq \max_{1 \leq q \leq n-1} (cq^2 + c(n-q)^2 + \theta(n)) \]
  \[ = c \max_{1 \leq q \leq n-1} (q^2 + (n-q)^2) + \theta(n) \]
  \[ = c n^2 \]

• \( q^2 + (n-q)^2 \) is maximum at its endpoints:
  \[ T(n) \leq c1^2 + c(n-1)^2 + \theta(n) \]
  \[ = cn^2 - 2c(n-1) + \theta(n) \]
  \[ \leq cn^2 \]
QuickSort: Average-Case Analysis

- **Intuition**: Some splits will be close to balanced and others imbalanced; good and bad splits will be randomly distributed in the recursion tree.

- **Observation**: Asymptotically bad run time occurs only when we have many bad splits in a row.
  - A bad split followed by a good split results in a good partitioning after one extra step!
  - Thus, we will still get $O(n \lg n)$ run time.

![Diagram](a) ![Diagram](b)
Randomized Quicksort

- How to modify quicksort to get good average-case behavior on all inputs?
- **Randomization!**
  - Randomly permute inputs, or
  - Choose the partitioning element $x$ randomly at each iteration.

```plaintext
RANDOMIZED-PARTITION(A, p, r)
1. $i = \text{RANDOM}(p, r)$
3. return PARTITION(A, p, r)
```

```plaintext
RANDOMIZED-QUICKSORT(A, p, r)
1. if $p < r$
2. $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
3. RANDOMIZED-QUICKSORT(A, p, q)
4. RANDOMIZED-QUICKSORT(A, q+1, r)
```
Average-Case Recurrence

- Assume that all keys are distinct.
- Partition into lower side : upper side = 1 : n - 1 with probability 2/n; others with probability 1/n. **Why?**
- Partition at an index q:

\[
T(n) = \frac{1}{n} \left( 2T(1) + T(n-1) + \sum_{q=2}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n)
\]

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

\[
= \frac{1}{n} \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n) \quad \text{// why?}
\]

\[
= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \quad \text{// why?}
\]
Average-Case Recurrence (cont'd)

- Guess \( T(n) \leq cn \log n \) and verify it inductively:

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

\[
\leq \frac{2c}{n} \sum_{k=1}^{n-1} k \log k + \Theta(n)
\]

- Need to show that

\[
\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{[n/2]-1} k \log k + \sum_{k=[n/2]}^{n-1} k \log k
\]

\[
\leq (\log n - 1) \sum_{k=1}^{[n/2]-1} k + \log n \sum_{k=[n/2]}^{n-1} k \quad // \text{why?}
\]

\[
\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2.
\]

- Substituting \( \sum_{k=1}^{n-1} k \log k \), we have \( T(n) \leq cn \log n \).

- Practically, quicksort is often 2-3 times faster than merge sort or heap sort.
Decision-Tree Model for Comparison-Based Sorter

- Consider only the comparisons in the sorter.
- Correspond to each internal node in the tree to a comparison.
- Start at root and do the first comparison: $\leq$ ⇒ go to the left branch; $>$ ⇒ go to the right branch.
- Represent each leaf an ordering of the input ($n!$ leaves!)

![Decision Tree Diagram]

decision tree
\( \Omega(n \log n) \) Lower Bound for Comparison-Based Sorters

- There must be \( n! \) leaves in the decision tree.
- Worst-case \# of comparisons = \#edges of the longest path in the tree (tree height).
- **Theorem**: Any decision tree that sorts \( n \) elements has height \( \Omega(n \log n) \).
  
  Let \( h \) be the height of the tree \( T \).
  
  - \( T \) has \( \geq n! \) leaves.
  
  - \( T \) is binary, so has \( \leq 2^h \) leaves.

\[
\begin{align*}
2^h & \geq n! \\
\therefore h & \geq \log n
\end{align*}
\]

\[
\geq \Omega(n \log n) \quad // \text{Stirling's approximation } n! > \left( \frac{n}{e} \right)^n
\]

- Thus, any comparison-based sorter takes \( \Omega(n \log n) \) time in the **worst case**.
- Merge sort and heapsort are **asymptotically optimal** comparison sorts.
Counting Sort: A Non-comparison-Based Sorter

- **Requirement:** Input integers are in known range [1..k].
- **Idea:** For each $x$, find # of elements $\leq x$ (say $m$, excluding $x$) and put $x$ in the $(m + 1)st$ slot.
- Runs in $O(n+k)$ time, but needs extra $O(n+k)$ space.
- Example: A: input; B: output; C: working array.

![Diagram](image)
Counting Sort

COUNTING-SORT(A, B, k)
1. for $i = 1$ to $k$
2. $C[i] = 0$
3. for $j = 1$ to $A.length$
5. // $C[i]$ now contains the number of elements equal to $i$.
6. for $i = 2$ to $k$
7. $C[i] = C[i] + C[i-1]$
8. // $C[i]$ now contains the number of elements $\leq i$.
9. for $j = A.length$ downto 1

• Linear time if $k = O(n)$.
• **Stable** sorters: counting sort, insertion sort, merge sort.
• **Unstable** sorters: heap sort, quicksort.
Radix Sort

RADI$X$-SORT$(A, \ d)$
1. for $i = 1$ to $d$
2. Use a stable sorter to sort array $A$ on digit $i$

| 928 | 520 | 101 | 101 |
| 101 | 101 | 401 | 228 |
| 401 | 401 | 308 | 308 |
| 228 | 928 | 520 | 329 |
| 329 | 228 | 928 | 401 |
| 308 | 308 | 228 | 520 |
| 520 | 329 | 329 | 928 |

• Time complexity: $\Theta(d(n+k'))$ for $n$ $d$-digit numbers in which each digit has $k'$ possible values.
  - Which sorter?
Bucket Sort

Step 1: distribute

Step 2: sort

Step 3: combine
Bucket Sort Example

1. Original list
   - 623, 192, 144, 253, 152, 752, 552, 231

2. Bucket based on 1st digit, then sort bucket
   - 192, 144, 152 \Rightarrow 144, 152, 192
   - 253, 231 \Rightarrow 231, 253
   - 552 \Rightarrow 552
   - 623 \Rightarrow 623
   - 752 \Rightarrow 752

3. Concatenate buckets
   - 144, 152, 192 \hspace{1cm} 231, 253 \hspace{1cm} 552 \hspace{1cm} 623 \hspace{1cm} 752
Note on Sorting in Linear Time

- Counting sort: Linear time if \( k = O(n) \); pseudo-linear time, otherwise.
- Radix sort: Linear time if \( d \) is a constant and \( k' = O(n) \); pseudo-polynomial time, otherwise.
  - Unstable, in-place radix sort can be implemented.
- Bucket sort: Expected linear time if the sum of the squares of the bucket sizes is linear in the # of elements (uniform distribution)

<table>
<thead>
<tr>
<th>Non-comparison-based sorters</th>
<th>Algorithm</th>
<th>Best case</th>
<th>Average case</th>
<th>Worst case</th>
<th>Stable?</th>
<th>In-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting</td>
<td>( O(n + k) )</td>
<td>( O(n + k) )</td>
<td>( O(n + k) )</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Radix</td>
<td>( O(d(n + k')) )</td>
<td>( O(d(n + k')) )</td>
<td>( O(d(n + k')) )</td>
<td>Yes*</td>
<td>No*</td>
<td></td>
</tr>
<tr>
<td>Bucket</td>
<td>–</td>
<td>( O(n) )</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>
Order Statistics

• **Def:** Let $A$ be an ordered set containing $n$ elements. The *$i$-th order statistic* is the $i$-th smallest element.
  
  – Minimum: 1st order statistic
  
  – Maximum: $n$-th order statistic
  
  – Median: $\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{n+1}{2} \right\rceil$-th order statistic

• **The Selection Problem:** Find the $i$-th order statistic for a given $i$.
  
  – **Input:** A set $A$ of $n$ (distinct) numbers and a number $i$, $1 \leq i \leq n$.
  
  – **Output:** The element $x \in A$ that is larger than exactly $(i-1)$ elements of $A$.

  
  – Time complexity: $O(n\log n)$.
  
  – Can we do better??
Finding Minimum (Maximum)

Minimum(A)
1. \( min = A[1] \)
2. for \( i = 2 \) to \( A.length \)
3. if \( min > A[i] \)
4. \( min = A[i] \)
5. return \( min \)

- cf. The hiring problem in Chapter 5
- **Exactly** \( n-1 \) comparisons.
  - Best possible?
  - Expected # of times executed for line 4: \( O(lg \ n) \).
- Naive simultaneous minimum and maximum: \( 2n-3 \) comparisons.
  - Best possible? (See Exercise 9.1-2; Hint: divide & conquer.)
Selection in Linear Expected Time

Randomized-Select($A, p, r, i$)
1. if $p == r$
2. return $A[p]$
3. $q =$ Randomized-Partition($A, p, r$)
4. $k = q - p + 1$
5. if $i \leq k$
6. return Randomized-Select($A, p, q, i$)
7. else return Randomized-Select($A, q+1, r, i-k$)

- Randomized-Partition first swaps $A[p]$ with a random element of $A$ and then proceeds as in regular PARTITION.
- Randomized-Select is like Randomized-Quicksort, except that we only need to make one recursive call.
- Time complexity
  - Worst case: $1:n-1$ partitions.
  - $T(n) = T(n-1) + \theta(n) = \theta(n^2)$
  - Best case: $T(n) = \theta(n)$
  - Average case? Like quicksort, asymptotically close to best case.
Selection in Linear Expected Time: Average Case

\[ T(n) \leq \frac{1}{n} \left( T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k)) \right) + O(n) \]

\[ \leq \frac{1}{n} \left( T(n-1) + 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) \right) + O(n) \]

\[ = \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n) \]

- Assume \( T(n) \leq cn. \)

\[ T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) \]

\[ \leq \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \right) + O(n) \]

\[ \leq cn - c \left( \frac{n}{4} + \frac{1}{2} \right) + O(n) \]

\[ \leq cn. \]

- Thus, on average, Randomized-Select runs in linear time.
Selection in Worst-Case Linear Time

- **Key:** Guarantee a good split when array is partitioned.
- **Select**\((A, p, r, i)\)
  
  1. Divide input array \(A\) into \([n/5]\) groups of size 5 (possibly with a leftover group of size < 5).
  2. Find the median of each of the \([n/5]\) groups.
  3. Call Select recursively to find the median \(x\) of the \([n/5]\) medians.
  4. Partition array around \(x\), splitting it into two arrays of \(A[p, q]\) (with \(k\) elements) and \(A[q+1, r]\) (with \(n-k\) elements).
  5. \textbf{if} \((i \leq k)\ \textbf{then} \) Select\((A, p, q, i)\) \textbf{else} Select\((A, q + 1, r, i - k)\).
Runtime Analysis

- Main idea: Select guarantees that $x$ causes a good partition; at least

$$3 \left( \left\lfloor \frac{1}{2} \left\lfloor \frac{n}{5} \right\rfloor \right\rfloor - 2 \right) \geq \frac{3n}{10} - 6$$

elements $> x$ (or $< x$) $\rightarrow$ worst-case split has $7n/10 + 6$ elements in the bigger subproblem.

- Run time: $T(n) = T(\left\lceil n/5 \right\rceil) + T(7n/10 + 6) + O(n)$.
  1. $O(n)$: break into groups.
  2. $O(n)$: finding medians (constant time for 5 elements).
  3. $T(\left\lceil n/5 \right\rceil)$: recursive call to find median of the medians.
  4. $O(n)$: partition.
  5. $T(7n/10 + 6)$: searching in the bigger partition.

- Apply the substitution method to prove that $T(n) = O(n)$. 