Learning Goals
During this lab, you will:

- Recap quicksort and quickselect
- Review min/max heaps and heapsort
- Apply these ideas to solve problems

Summary of Quicksort and Quickselect
Recall the (rough) sketches of the quicksort and quickselect algorithms:

Quicksort (sorting any array)
- Within the array pick an arbitrary value and call that the pivot.
- Rearrange the array such that all values less than the pivot come before the pivot, and all values greater than the pivot come after the pivot.
- Apply the above steps in a recursive manner to the both sub-arrays (elements of smaller values, and elements of greater values). For any sub-arrays that contain 1 element, return that sub-array.
- Running time: worst case $O(n^2)$, best case $O(n \log n)$

Quickselect (finding the k-th largest element in an array)
- Within the array pick an arbitrary value and call that the pivot.
- Rearrange the array such that all values less than the pivot come before the pivot, and all values greater than the pivot come after the pivot.
- Apply the above steps in a recursive manner to one of the subarrays based on $k$.
- Running time: worst case $O(n^2)$, best case $O(n)$

Problems

Problem 1: Running time analysis

*Explain the worst and best case run times for both Quicksort and Quickselect*

- Which particular inputs cause worst and best cases?
- Use recurrences were applicable.

Explain the worst and best case run times for both quicksort and quickselect.
Problem 2: Checking value and rank

Given an unsorted array of distinct integers, determine whether there is an element in the array that has a
value equal to its rank in the sorted array.

• Should you do this in an iterative or recursive manner and why?
• How might you utilize quicksort and or quickselect?
• What is the running time of the approach? Is it optimal?

Problem 3: Finding the popular element

How can you determine whether or not there is an element in a given array $A$ of integers (not necessarily
distinct) that occurs at least $\lceil \frac{n}{2} \rceil$ times?

• Does this particular element have any special properties?
• What is the running time of the optimal approach?

Introduction: Heaps

A heap is a tree-like data structure that satisfies the heap-order property.

**Definition** (Heap-Order Property). A tree has the heap-order property if for any parent node $P$ with a
child $C$, the key of $P$ is ordered with respect to the child $C$.

Common examples of orderings on a heap would be $\geq$ (max-heap) or $\leq$ (min-heap). For $\geq$, the key in
each node in the heap $T$ is greater than or equal to the keys of all nodes in its subtree.

![An example binary max-heap. Note that the root contains the maximum key.](image)

Notice that this definition immediately implies that the root must contain either the “maximum” or the
“minimum” of the ordering relationship that we define, since the root is the parent/ancestor of every other
node. Specializing this definition to keys that act like natural numbers, or keys that implement `Comparable`,
we have our classic min-heap and max-heap. This basic idea is really powerful, as the heap data structure
maintains the “maximum” or “minimum” element whenever we add to it or remove from it. This means
that we can retrieve the max/min element quickly!

**Binary Heaps**

A binary heap is a binary tree, but with the heap-order property. A binary heap is most commonly imple-
mented by flattening a tree in level order into an array. It satisfies the following property:
**Definition** (Shape Property). A tree has the heap-shape property if the tree is a complete binary tree. That is, all levels of the tree are fully filled, except for possibly the last, where all nodes are as far left as possible.

With the shape property, we can easily index into a binary heap, since we will not have to worry about “gaps.”

![A max-heap visualized as both a tree and an array.](image)

For an element at index \( i \) of \( A \), its left and right children can be found at indices \( 2i \) and \( 2i+1 \) respectively. Conversely, an element at index \( i \) has its parent at index \( \lfloor i/2 \rfloor \).

This property holds true only if the heap begins at index 1 of the array (or if the array is one-indexed).

**Running time of Operations**

Running times are given with respect to \( n \), where \( n \) is the number of elements in the binary heap.

- **INSERT**(*x, k*): An element \( x \) with key \( x \) may be inserted in \( O(\log n) \) time.
- **FIND-MIN/MAX():** Finding the min/max of a binary heap takes \( O(1) \) time.
- **EXTRACT-MIN/MAX():** Removing the root and restoring the min/max heap property takes \( O(\log n) \) time.
- **DECREASE/INCREASE-KEY**(*x, k*): Changing the key of an element can be done in \( O(\log n) \) time. Note that the **Java** implementation of a priority queue does not support this operation.

**Partial Ordering**

We say that the heap-order property induces a partial order over its elements. Intuitively, a partial order means that not every pair of elements are related. Even though we know that 17 is less than 23, when we insert these numbers into the heap, we cannot determine which number is “greater” solely by its position in the heap. Compare this to inserting both elements in a binary search tree, where we can determine the order by examining their relative positions. We say that the binary search tree establishes a total order.

For some problems, it is enough to have just a partial ordering. For example, if you want to get the \( k \)-largest elements of a list relatively fast, you can use a heap to achieve this. As you’ve seen with **MERGESORT** and some implementations of **QUICKSORT**, you can get a stronger, total ordering at the cost of a larger running time \([\Omega(n \log n)]\). However, building a heap only takes time linear in the number of elements. Therefore, we can get the maximum/minimum in linear time and the partial ordering!
Building a (Max) Heap

In order to build a heap, we define the following subroutine: \texttt{MAX-HEAPIFY}. Under the assumption that the

\begin{verbatim}
function \texttt{MAX-HEAPIFY}(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if \(l \leq A.heapsize \text{ and } A[l] > A[i]\) then
  largest ← l
else
  largest ← i
if \(r \leq A.heapsize \text{ and } A[r] > A[largest]\) then
  largest ← r
if \(largest \neq i\) then
  \texttt{SWAP}(A[i], A[largest])
  \texttt{MAX-HEAPIFY}(A, largest)
\end{verbatim}

left and right subtrees of the \(i\)'th vertex are valid max heaps, \texttt{MAX-HEAPIFY} ensures that the subtree rooted
at \(i\) is also a valid max heap. The running time analysis of \texttt{MAX-HEAPIFY} is left as a discussion topic. We
can then write:

\begin{verbatim}
function \texttt{BUILD-MAX-HEAP}(A)
A.heapsize ← A.length
for \(i ← \lfloor A.length/2 \rfloor\) downto 1 do
  \texttt{MAX-HEAPIFY}(A, i)
\end{verbatim}

The \texttt{BUILD-MAX-HEAP} algorithm starts from the last internal node of the binary tree representation of
\(A\) and converts each subtree to a max-heap, recursing upwards. As above, the running time analysis of \texttt{BUILD-MAX-HEAP} is left as a discussion topic.

Heapsort

\begin{verbatim}
function \texttt{HEAPSORT}(A)
  \texttt{BUILD-MAX-HEAP}(A)
  for \(i ← A.length\) downto 2 do
    \texttt{SWAP}(A[1], A[i])
    A.heapsize ← A.heapsize − 1
    \texttt{MAX-HEAPIFY}(A, 1)
\end{verbatim}

The \texttt{HEAPSORT} algorithm works by first converting the input array \(A\) to a max-heap. It grows the
sorted subarray from right to left by swapping out the root (largest element at \(A[1]\)) to its proper place in
the sorted subarray and restoring the max-heap property on the unsorted subarray. (Does this notion of
dividing the input into an unsorted/sorted region remind you of another sorting algorithm...?) The running
time analysis of \texttt{HEAPSORT} is also left as a discussion topic.

Testing Your Understanding

Answer the following questions regarding implementations of binary heaps.

Problem 1. Consider the following array:
Problem 2. You have been hired to write an application for 121-CIS’s new network router! 121-CIS plans to sell their routers to businesses with large corporate networks that need swift detection of network attacks. A network attack is characterized by a large amount of traffic from a single IP address. For the application, you are parsing a stream of packets containing an IP-address and their frequency. Routers have limited memory, and you can only maintain $O(k)$ space for your application, where $k \ll n$.

Design an $O(n \log k)$ time algorithm to find the $k$-th most frequent IP-address, where $n$ is the total number of IP addresses in the stream.

Problem 3. Consider an indefinitely long stream of unsorted integers. We are interested in knowing the median (in sorted order) at any given time. How would we do this in an efficient manner?