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.
Solution

Quicksort: If we choose a pivot at index $k$, then we divide our subarrays into one part of size $k$ and another part of size $n - k$. The total time for rearranging the arrays (relative to the pivot), takes some $cn$ time for a constant $c$. Then we have the following recurrence:

$$T(n) = T(k) + T(n - k) + cn$$

In the worst case, the pivot chosen is either the largest or smallest element in the array considered. For example, take $k = 1$, which yields a recurrence of

$$T(n) = T(1) + T(n - 1) + cn$$

It is quite intuitive to see that this expands to $O(n^2)$ as desired, alternatively we can expand the terms to yield $O(n^2)$.

For the best case, the pivot is somewhere "close” to the true median of the array. Consider the pivot being chosen at index $n/2$ such that the recurrence becomes

$$T(n) = T(n/2) + T(n/2) + cn = 2T(n/2) + cn$$

Expanding out terms, we get a general pattern of $2^kT(n/2^k) + kcn$, which continues until $n = 2^k$. Thus we can set $k = \log n$ to get a running time of $T(n) = nT(1) + cn \log n = O(n \log n)$ as the optimal running time. Alternatively, we can note that the original recurrence has the same form as that of Mergesort and thus must be $O(n \log n)$.

Quickselect: The analysis is similar to that of Quicksort, except the recurrences here are different to reflect that the recursive calls occur on one subarray rather than both.

In the worst case, we may always partition the array into uneven sizes such that $k$ is on the larger side. Take for example, the pivot being the largest element and $k$ being a small number. Then our recurrence becomes

$$T(n) = T(n - 1) + cn$$

since rearranging takes $cn$ as before and the subarray we recursively look at is always one element smaller than the original. Since $T(1) = O(1)$, this simplifies to the same worst-case analysis with Quicksort, which yields $O(n^2)$.

In the best case, we can choose, once again, a pivot that is close to the true median, such that our recurrence becomes $T(n) = T(n/2) + cn$. Expanding out terms we get a general form of

$$T(n/2^k) + (1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^k})cn$$

which continues until $n = 2^k$. The

$$(1 + \frac{1}{2} + \frac{1}{4} + ...)$$

is simply a geometric series that sums to a constant (2 in this case), thus the overall running time is

$$T(1) + 2cn = O(n)$$

in the best case.

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?

Solution
Consider an algorithm with parameters $l$ and $r$ denoting the left and right ends of the array. Now we can choose the median element, $m$ of rank $(r - l)/2$ and partition accordingly. If the value of $m$ is equal to $l + (r - l)/2$ (its rank), then return true, otherwise we can recursively call the algorithm on the left half of the array if $m > l + (r - l)/2$ and on the right half of the array if $m < l + (r - l)/2$. Notice that when we partition the array, we can eliminate exactly half of the remaining elements as candidates if the median is too high or low - if it is $> l + (r - l)/2$ for example, then we know any element above $m$ must have a value greater than its rank because the integers are distinct. Thus our recurrence relation is $T(n) = T(n/2) + O(n)$, which results in a running time of $O(n)$ (see the Quickselect best-case analysis in the problem before).

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 $\lceil n/2 \rceil$ times?

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

Solution
The key here is to note that if an element were to indeed occur at least $\lceil n/2 \rceil$ times, then it must be a median of $A$. Thus, we can use Quickselect to find the median in $O(n)$ time. However, if an element is a median, it does not need to necessarily occur at least $\lceil n/2 \rceil$ times (why?), so once we have the median we can just linearly scan the array and count how many times it appears. If it appears more than $\lceil n/2 \rceil$ times, we return true, otherwise we return false. Since the linear scan takes $O(n)$ time, the overall algorithm takes $O(n)$ time. Notice we didn’t need any extra space!

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.

![Example binary max-heap](image)

An example binary max-heap. Note that the root contains the maximum key.
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 implemented 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.”

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>16</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

A max-heap visualized as both a tree and an array.

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: `MAX-HEAPIFY`. Under the assumption that the left and right subtrees of the $i$’th vertex are valid max heaps, `MAX-HEAPIFY` ensures that the subtree rooted at $i$ is also a valid max heap. The running time analysis of `MAX-HEAPIFY` is left as a discussion topic. We can then write:

```plaintext
function MAX-HEAPIFY(A, i)
    l ← LEFT(i)
    r ← RIGHT(i)
    if $l \leq A.heapsize$ and $A[l] > A[i]$ then
        largest ← l
    else
        largest ← i
    if $r \leq A.heapsize$ and $A[r] > A[largest]$ then
        largest ← r
    if $largest \neq i$ then
        SWAP(A[i], A[largest])
        MAX-HEAPIFY(A, largest)
```

The `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 `BUILD-MAX-HEAP` is left as a discussion topic.

### Heapsort

The `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 `HEAPSORT` is also left as a discussion topic.
function HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← A.length downto 2 do
        swap(A[1], A[i])
        A.heapsize ← A.heapsize − 1
        MAX-HEAPIFY(A, 1)

Testing Your Understanding

Answer the following questions regarding implementations of binary heaps.

Problem 1. Consider the following array:

null 6 7 9 15 13 17 14 20 16 23 18 19 37 42 ···

Let this array be the underlying storage for a binary heap. Is this a max-heap or a min-heap? What is the parent of the key 17? What is the left child of 17? The right child?

Solution. We traverse the array from right-to-left, and fill the levels of a binary tree. We get the following heap:

This is a min-heap. Notice that 17 has index 6 in the array (zero-indexed).

For a heap that starts on index 1 of an array (and has index 0 unused), for an index \(k\), the following are true:

1. The parent has index \(\lfloor \frac{k}{2} \rfloor\).
2. The left child has index \(2k\).
3. The right child has index \(2k + 1\).

The parent of 17 is 9. The parent, 9, has index 3.
The left child of 17 is 19. The left child, 19, has index 12.
The right child of 17 is 37. The right child, 37, has index 13.

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.
Solution. Take the first $k$ packets of the input stream, and construct a min-heap of size $k$, where IP addresses are inserted into the min-heap and ordered by their frequency. For each IP address in the input, if the frequency is greater than or equal to the frequency of the address at the root of the heap, remove the root, insert the new address as the new root, and perform \textsc{Min-Heapify}. Else, if the frequency of the new address is less than or equal to the frequency of the root, do nothing. After processing all input, return the address at the root of the heap.

\textit{Proof of correctness.} We want to show that the algorithm, as described above, returns the IP address of the input that is the $k$-th most frequent. Consider any address that enters the heap. By construction, any address that enters the heap must have a frequency that is greater than or equal to some other address. Assume for the sake of contradiction that an address $a_{\text{bad}}$ with a frequency greater than order $k$ (i.e., less frequent than the $k$-th most frequent) remains on the heap at the termination of the algorithm. But because we always maintain a heap that has a maximum size of $k$, this implies that some address $a_{\text{good}}$ with frequency order less than or equal to $k$ (more frequent than the $k$-th most-frequent) is excluded from the heap. But by construction, it is impossible for $a_{\text{good}}$ to have been excluded, since it would have compared more frequent than $a_{\text{bad}}$, which, in turn, would also be more frequent than the root element by transitivity! This is a contradiction, so our algorithm must be correct.

\textit{Running time analysis.} Constructing a min-heap from the first $k$ elements (unsorted) takes time $O(k)$. We maintain the heap at size $k$ by removing the root in constant time and inserting a new address at the root and percolating downwards. Since each address can be inserted in the heap as the root at most once, each address is percolated downwards to its final position in time $O(\log k)$. Since the input has $n$ addresses, our overall running time is $O(k + n \log k)$. But since $n \gg k$, we have a final complexity of $O(n \log k)$. \hfill \Box

\textbf{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?

\textit{Solution.} We can keep a min-heap and a max-heap simultaneously. The max-heap contains the smaller half of numbers and the min-heap contains the larger half of numbers. Maintain the following two invariants:

1. The difference in size of the max-heap and the size of the min-heap is at most 1.

2. The root of the max-heap is always less than or equal to the root of the min-heap.

For the first two elements of the stream, put the smaller element into the max-heap, and the larger element into the min-heap. Whenever a new element of the stream is encountered, insert the element into the max-heap. If the element is smaller than the root of the max-heap, insert it into the max-heap, otherwise insert it into the min-heap. If invariant \textit{1} is violated, remove the root from the larger heap and insert that newly-removed element into the smaller heap. To retrieve the median at any given time, if the number of total elements is odd, take the root of the max-heap; otherwise, take the average of the roots of both heaps.

\textit{Proof of correctness.} The correctness of the computation of the median from the invariants is immediate. We want to show that our algorithm maintains these invariants. We leave justification of these facts to the reader.

\textit{Running time analysis.} Let $n$ be the number of elements seen in the stream. In this algorithm, we perform at most two insertions into heaps of size at most $\lceil n/2 \rceil$, which is a running time that is $O(\log n)$. We can access the roots of the heaps for the median computation in constant time. Hence, for every element of the stream, we maintain our data structures in $O(\log n)$ time. Since every element is stored internally, we use $O(n)$ space. \hfill \Box