Readings

- [Lecture Notes Chapter 14: Binary Heaps and Heapsort]

Review: Heaps

A heap is a tree-like data structure that implements the priority queue ADT, which allows us to maintain a set of elements, each with an associated key, and select the element with highest/lowest priority. Heaps satisfy the two following properties:

**Heap Property:** In a max-heap, for each node $i$, we have $A[\text{Parent}(i)] \geq A[i]$, so the maximum value is stored at the root. In a min-heap, for each node $i$, we have $A[\text{Parent}(i)] \leq A[i]$, so the minimum value is stored at the root.

**Shape Property:** A heap is an almost complete binary tree, meaning that every level of the tree is completely filled except for the last, which must be filled from left to right.

Implementation Details

Because a heap is an almost complete binary tree, we are able implement it using an array as shown below:

Observe that we can populate the array from left to right by doing a level-order traversal of the tree, where we start from the root and go through each level of the tree from left to right. Additionally, because of the shape property, if the root is stored at index 1 of the array, given a node at index $i$, its left child can be found at index $2i$, its right child can be found at index $2i + 1$, and its parent can be found at index $\lfloor i/2 \rfloor$. 
Operations (Max-Heaps)

MAX-HEAPIFY allows us to maintain the max-heap property at the node it is called on. More specifically, given a node whose children are both max-heaps, we can call MAX-HEAPIFY on the node so the entire subtree rooted at the node will now be a max-heap. It works by allowing the node to “float-down” through the tree; at each level, we swap it with its largest child, or if it is larger than both of its children, we terminate since the max-heap property holds. It runs in $O(\log n)$ time or $O(h)$ time, where $h$ is the height of the heap, since in the worst-case, the node must “float-down” through the entire tree.

BUILD-MAX-HEAP constructs a max-heap from an unsorted array by repeatedly calling MAX-HEAPIFY on each node from the “bottom-up”, starting at the nodes right above the leaves (which by definition are max-heaps!). It runs in $O(n)$ time. We know that an $n$-element heap has height $\lfloor \lg n \rfloor$, calling MAX-HEAPIFY at any height $h$ takes $O(h)$ time, and there are at most $\lceil n/2^h+1 \rceil$ nodes at any height $h$ of an $n$-element heap. So, we can express the running time as

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^h+1} \right\rfloor \cdot O(h) \leq O \left( \frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n \cdot 2) \quad (\text{arithmetic-geometric series})$$

$$= O(n)$$

EXTRACT-MAX allows us to remove and return the element with the maximum key. It works by removing the root, replacing it with the right-most element in the bottom level/last element in the array, and then calling MAX-HEAPIFY on the “new” root to maintain the max-heap property. Removing the root and replacing it with the “last” element in the heap takes $O(1)$ time, and calling MAX-HEAPIFY takes $O(\log n)$ time, so it runs in $O(\log n)$ time.

INSERT allows us to add an element to our max-heap. It works by adding the element to the end of the array/max-heap, and then allowing it to “float-up” to its correct position in the max-heap by repeatedly swapping it with its parent as necessary to maintain the max-heap property. It runs in $O(\log n)$ time, since the path it takes while it “floats-up” has length $O(\log n)$.

PEEK returns the maximum element in the heap, which is stored at the root. Since we implement a heap with an array, this runs in $O(1)$ time because we just have to index into the array.

Problems

Problem 1

Given a data stream of $n$ test scores, design an $O(n \log k)$ time algorithm to find the $k$-th highest test score. Since PEFS provides minimal monetary resources, CIS 121 staff have limited access to storage space and can only afford you $O(k)$ space, where $k \ll n$.

Solution

**Algorithm:** Construct a min-heap from the first $k$ tests, where tests are ordered by their score, by calling BUILD-MIN-HEAP. For each remaining test in the data stream: if its score is greater than the score at the root of the heap, remove the root by calling EXTRACT-MIN and then INSERT the current test; otherwise, the score of the current test is less than or equal to the score at the root, so do nothing. After processing all tests in the data stream, return the score at the root of the heap by calling PEEK.
Proof of Correctness: We will prove the correctness of our algorithm by showing that our min-heap contains the $k$-highest scores at the termination of our algorithm. Let $s$ be the $k$-th highest score. Assume for the sake of contradiction that a test $a_{bad}$ with a score lower than $s$ remains in the heap when our algorithm terminates. Since we maintain a heap of size $k$, this implies that some test $a_{good}$ with a score higher than $s$ is excluded from the heap. However, by design, it is impossible for $a_{good}$ to have been excluded, since its score is higher than $a_{bad}$, so it would have been included by transitivity. This is a contradiction, so by returning the score at the root, our algorithm is correct.

Runtime Analysis: Constructing a min-heap from the first $k$ tests by calling `Build-Min-Heap` takes $O(k)$ time. For each remaining test, we either do nothing, or we maintain the heap at size $k$ by calling `Extract-Min` and then `Insert` on the current score, which takes $O(\log k)$ time. Each test is inserted into the heap at most once, and since the data stream has $n$ scores, our overall running time is $O(k + n \log k)$. Since $k \ll n$, our final running time is $O(n \log k)$.

Space Analysis: As a “pre-processing” step, we first construct a min-heap with the first $k$ tests. We maintain the heap at size $k$ because for each remaining test that we insert, we remove the root. Therefore, the space complexity of our algorithm is $O(k)$.

Midterm 1 Review Problems

Problem 2

Provide a running time analysis of the following loop. That is, find both Big-O and Big-$\Omega$:

```java
for (int i = 0; i < n; i++)
    for (int j = i; j <= n; j++)
        for (int k = i; k <= j; k++)
            sum++;
```

Solution

We can try to analyze the running time using an exact summation while also leveraging some Big-O notation. We know that the innermost loop runs in at most $(j - i + 1)$ time for fixed some $i, j$. Hence, the body of the middle loop runs at most $c(j - i + 1)$ time. Since $0 \leq i < n$ and $i \leq j \leq n$, we can express the running time of the code snippet as

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n} c(j - i + 1)$$

Note that this summation is difficult to compute by hand. However, since the conditions in the nested for loops just deal with incrementing variables, we can find a Big-O and Big-$\Omega$ bound for the code snippet by finding the running time on a superset and subset of values, respectively.

First, we find Big-O. Since Big-O upper bounds the running time of an algorithm, we consider supersets of values for $i, j,$ and $k$ and upper bound the running time of the code on these supersets. We are able to do this because an upper bound on the running time of these supersets is also going to an upper bound on the running time of the code snippet! Specifically, we choose supersets where $0 \leq i \leq n$, $0 \leq j \leq n$, $0 \leq k \leq n$. Since we have three loops and each loop runs at most $n + 1$ times, we can say that the entire code snippet runs in $O((n + 1)^3) = O(n^3)$ time.

Next, we find Big-$\Omega$. Since Big-$\Omega$ lower bounds the running time of an algorithm, we consider subsets of values for $i, j,$ and $k$ and lower bound the running time of the code on these subsets. We are able to do this because a lower bound on the running time of these subsets is also going to be a lower bound on the running time of the code snippet! Specifically, we choose subsets where $0 \leq i \leq n/4$ and $3n/4 \leq j \leq n$. Since the inner loop runs from when $k = i$ until when $k \leq j$, we know that for each $n^2/16$ possible combinations of
these values for $i$ and $j$, the innermost loop runs at least (or a minimum of) $3n/4 - n/4 = n/2$ times, since $3n/4$ is the minimum value of $j$ we bound on and $n/4$ is the maximum value of $i$ we bound on. Hence, the running time is $\Omega \left( \frac{n^2}{16} \cdot \frac{n}{2} \right)$, or equivalently, $\Omega(n^3)$.

**Note:** There are usually many different supersets/subsets that you can take to get achieve a Big-$O$/Big-$\Omega$ bound. However, it is important to double check that your subsets and supersets are valid. In this example, since the value of $j$ depends on $i$, the values of $j$ we take as a “subset” should actually be a subset for each value of $0 \leq i \leq n/4$ we bound on for us to have a “valid” subset of the original values $j$ can take on. That is, when $i = 0$, we see that $0 \leq j \leq n$; when $i = 1$, we see that $1 \leq j \leq n$; and then when $i = n/4$, $n/4 \leq j \leq n$. In each case, we can see that $3n/4 \leq j \leq n$ will always be a subset of the actual values $j$ takes on given a value of $i$ in our subset. Additionally, observe that some code snippets may not have a matching Big-$O$/Big-$\Omega$ bound; while this code snippet does, depending on what subsets/supersets we took, we may not have found a $\Theta$-bound.

**Problem 3**

Assume $n$ is a power of 3.

$$T(n) = \begin{cases} 2T(\frac{n}{3}) + n & n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solve this recurrence by expansion (do not use Simplified Master Theorem).

**Solution**

Since we can assume that $n$ is some power of 3, observe that $n = 3^k$ implies that $k = \log_3 n$. Using the method of expansion, we expand $T(n)$ as follows:

$$T(n) = 2T\left( \frac{n}{3} \right) + n$$
$$= 2 \left( 2T\left( \frac{n}{3^2} \right) + \frac{n}{3} \right) + n$$
$$= 2 \left( 2 \left( 2T\left( \frac{n}{3^3} \right) + \frac{n}{3^2} \right) + \frac{n}{3} \right) + n$$
$$\vdots$$
$$= 2^k T\left( \frac{n}{3^k} \right) + n \sum_{i=0}^{k-1} \left( \frac{2}{3} \right)^i$$

The recursion bottoms out when $n/3^k = 1$, so when $k = \log_3 n$. Thus, we get

$$T(n) = 2^{\log_3 n} \cdot T(1) + n \sum_{i=0}^{\log_3 n - 1} \left( \frac{2}{3} \right)^i$$

$$= n^{\log_3 2} + n \left( \frac{1 - \left( \frac{2}{3} \right)^{\log_3 n}}{1 - \frac{2}{3}} \right)$$

**geometric series**

$$= n^{\log_3 2} + 3n - 3n \left( \frac{2^{\log_3 n}}{n} \right)$$

**log properties**

$$= 3n - 3n^{\log_3 2}$$

$$= \Theta(n)$$