DSC 190 - Discussion 02

Problem 1.

In lecture, we saw that inserting an element into an existing heap takes $\Theta(\log n)$ time in the worst case, where n is the number of elements currently in the heap. This means that if we start with an empty heap and insert n elements, the time taken in the worst case is $\Theta(n \log n)$. In this problem, we'll see that we can actually build a heap in $\Theta(n)$ time if we already have all of the elements to be inserted stored in an array.

a) Now suppose we have an array with n elements that we wish to turn into a heap. We will do this by calling `._push_down(i)` on each heap node, but in a particular order. We don't need to call it on the leaf nodes, as they are already as low as they can go. Instead, we'll start by calling `.push_down(i)` on the nodes at height 1, then nodes at height 2, and so on, going from right to left.

Implement this strategy in code.

Solution:

```
def parent(ix):
    return (ix - 1)//2


def build_heap(arr):
    n = len(arr)
    heap = MaxHeap(arr)
    # find the index of the rightmost non-leaf node
    # this will be the parent of the last node
    ix = parent(n-1)
    while ix >= 0:
        heap._push_down(ix)
```

b) Show that building a heap in this way takes $\Theta(n)$ time, where n is the length of the array.

Hint: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

Solution: The cost of a `._push_down` is $O(h)$ in the worst case, where h is the height of the node being pushed down.

At first, we push down nodes at height one, where $h = 1$. How many such nodes are there? A quick check shows that in a full binary tree, there are exactly $(n+1)/4$.

Next, we push down nodes at height two, where $h = 2$. There are $(n+1)/8$ such nodes in a full binary tree.

And so forth. To push down a node at height h, it takes work ch, but there are $(n+1)/2^{h+1}$ such nodes.

In total, the work is:

$$\sum_{k=1}^{h} \frac{n+1}{2^{k+1}} k = \frac{n+1}{2} \sum_{k=1}^{h} \frac{k}{2^k}$$

Using the hint with $x = 1/2$, we see that:

$$\sum_{k=1}^{h} \frac{k}{2^k} \leq \sum_{k=0}^{\infty} k(1/2)^k = \frac{(1/2)}{(1/2)^2} = 1/2$$

So the sum is $\Theta(1)$. Not forgetting the $(n+1)/2$ out in front, we're left with $\Theta(n)$.

c) (Extra) Let's check that starting from an empty heap and inserting n elements one by one actually does take $\Theta(n \log n)$ time overall. This is a little trickier than it might seem, since n is changing as we insert elements. The first insert takes time roughly $c \log 1$ (for some constant c), the second takes time $c \log 2$, and so forth, until the last takes time $c \log n$. So the total time is:

$$c\,(\log 1 + \log 2 + \log 3 + \ldots + \log n)$$

Show that this is $\Theta(n \log n)$.

Hint: the upper bound is easier than the lower bound. For the lower bound, try splitting the sum in half and working with just the larger half.

Solution: For the upper bound:

$$c\,(\log 1 + \log 2 + \log 3 + \ldots + \log n) \leq c\,(\log n + \log n + \log n + \ldots + \log n)$$

Since there are n terms in the sum:

$$= cn \log n$$

For the lower bound, we apply the trick of splitting the sum in half, keeping everything from the $n/2$ term on and throwing out the rest. We can assume that n is even and thus divisible by 2 to allow us to avoid writing a floor or ceiling:

$$c\,(\log 1 + \log 2 + \log 3 + \ldots + \log n) \geq c\,[\log(n/2) + \log(n/2+1) + \log(n/2+2) + \ldots \log n]$$

To get another lower bound, simply replace every term by the smallest term, $\log(n/2)$:

$$\geq c\,[\log(n/2) + \log(n/2) + \log(n/2) + \ldots \log(n/2)]$$

There are $n/2$ terms remaining, so:

$$\begin{aligned} &= c(n/2)\log(n/2) \\ &= \Theta(n \log n) \end{aligned}$$

Since the sum is bounded below by something which is $\Theta(n \log n)$, the sum is also $\Omega(n \log n)$.

Problem 2.

Describe a simple algorithm which takes in an array of size n and an integer parameter k and returns the k *most frequent* elements of the array. State the time complexity of your approach.

Example: given `[1, 9, 2, 4, 5, 2, 3, 4, 1, 1, 5]`, and $k = 3$, return 1, 2, and 5 (in no particular order).

Solution: Create a `dict` (hash map) of counts. Loop through the array of n elements, incrementing its count by one in the dictionary. Next, insert all of the $O(n)$ elements in the dictionary into a priority queue, where the priority is the count of the element. Pop k elements from the priority queue and return.

This takes $\Theta(n)$ average case time to do the insertions into the hash map, $\Theta(n)$ time to build a heap from an existing collection, and $k \log n$ time to pop k elements from the heap, for a total of $\Theta(n+k \log n)$ (average case) time.