Algorithms - Assignment 4

Due Thursday, March 28th by 11:59 pm

Starting with this assignment, we will be using jupyter notebook to submit assignments. To install jupyter notebooks, open a terminal and type pip install jupyter. Then to open a browser based environment where you can create jupyter notebooks, type in the terminal python -m notebook. (VScode also has an extension for jupyter notebooks, if you prefer that.) To write explanations, change the box you are working in to markdown. I've uploaded an example notebook to canvas. This way, we won't need to split assignments into coding and written portions anymore, and everything can appear in one place. I don't yet know what this means in terms of the extra credit per assignment! We will have to discuss that in class.

1 Theory

(1) A **d-ary heap** is a generalization of the binary heap we went over in class in which the tree breaks d ways at each node as opposed to 2. It stands to reason that, by assigning more children to a single parent node, we will get an overall lower height tree and therefore faster runtimes for our heap operations. This exercise will have you investigating how true that is.

   (a) Let $child(i, k)$ be a function which returns the index of the k^{th} child of i where i is a parent (going from left to right), for $1 \leq k \leq d$, and let $parent(i)$ be a function which returns the index of the parent of the node for i. Write down pseudocode 'algorithms' (really just formulas) for these two functions.

   (b) Recall that our $Max - Heapify(i)$ function from class was the one which allowed the number at index i to 'float down', restoring the max-heap property under the assumption that both child subtrees were themselves max-heaps. Rewrite this algorithm (in pseudocode) for d-ary heaps. What is the runtime of this $d - ary$ version of the problem? You don't need to do the rigorous proof on the last page of the written notes, you can say what is obvious.

   (c) How does it compare with the $d = 2$ case? State a precise relationship between the two. What can you conclude about d-ary heaps versus binary heaps?

(2) Suppose that we have an open-address hash table (i.e. a hash table with probing) of size m which at a particular moment contains n non-deleted items, and we decide to search for a key with no corresponding item in the table (i.e. our search is going to

be unsuccessful). Assume that all existing keys are equally likely to appear anywhere in the table. We claim that the expected number of probes required to carry out this search is at most $\frac{1}{1-\alpha}$, where $\alpha = \frac{n}{m}$ is the load factor. This exercise will walk you through a proof of this. Part (e) is an analysis of the fact and not part of the proof walkthrough.

(a) First, we need a lemma. Let X be a discrete random variable which can only take on nonnegative integer values (i.e. it's support is some subset of $\mathbb{N}$ including 0). Prove that

$$E(X) = \sum_{i=1}^{\infty} P(X \geq i) \tag{1}$$

(Hint: Start from the definition of expected value and use the fact that $P(X = i) = P(X \geq i) - P(X \geq i + 1)$.)

(b) Let X be the random variable representing the number of probes made during the search. For each i, let A_i be the event that the i^{th} probe occurs and that the index probed turns out to be occupied. Find a way to express $P(X \geq i)$ in terms of A_1 through A_{i-1}.

(c) Use the multiplication rule for probabilities to find an expression for this probability involving m, n, and i. Then find an upper bound for that expression which puts it in terms of the load factor, α. (Hint: How does probing process relate to random sampling?)

(d) Using the lemma you proved from part a and the bounds you found in part c, prove that $E(X) \leq \frac{1}{1-\alpha}$.

(e) Suppose we wanted to make the expected number of probes never greater than 2. What would we need to keep the load factor below in order to ensure this? What does this mean in terms of the table? Is it possible to make the expected number of probes less than 1?

2 Coding

(3) This problem will have you build your own heap methods, and from those make your own heapsort algorithm.

(a) Create functions parent(A,i), left(A,i), right(A,i), max_heapify(A,i) and build_max_heap(A,i) based on the pseudocode we wrote in class. Finally, build a function called is_max_heap(A), which checks to see if a given input list satisfies the max-heap property, and use it to verify that your functions work as expected.

(b) Use these to implement a heapsort(A) function. This will require you to slightly modify your functions from part (a). In particular, you will need to modify your max_heapify function to take a third argument, heap_size, and then check indices against that number rather than the actual length of the array. This is necessary in order to exclude the already sorted elements as your heapsort does it's work.

(c) Run a few test of your heapsort against the mergesort and quicksort algorithms and see how it compares. Attach a plot of runtimes to your write-up. I'm not picky about the details of the test, and you should be used to doing this by now.

(d) Import the heapq module and make a second heapsort function which sorts using those built-in methods rather (the code is available for copy-pasting in the lecture slides). Test this new heapsort against the one you made yourself. It should be much faster. Attach a plot in your write-up.

(e) Can you explain why the built-in heap methods are so much faster? Is there anything you could change about your own code which would make your heapsort comparable?

(4) This problem will have you implementing a priority queue using the heap methods you made in the previous problem. This will come in handy for other algorithms later in the class. It will also get you started working with classes and objects in python, if you've yet to be acquainted. I've uploaded the lecture slides on object oriented programming for my CS1 class on canvas if you want a quick overview of the syntax, and I'll give a bit on here as well. The basic syntax for creating a class in python looks like this:

```
class NameOfClass:
    # This is your constructor
    def __init__(self, args):
        # Assign attributes, do other constructor stuff
        self.attribute_1 = 'stuff'
        self.attribute_2 = 'more stuff'

    # Methods should always take 'self' as the first argument.
    def method_1(self, args):

    def method_2(self, args):

    # and so on
```

The __init__ method here is called a **dunder method**. These are special 'magic' methods which carry special meaning within the python language. In the case of __init__, this special property is that of being a constructor. See the OOP lecture slides if you want to know more about these (some of them are quite cool).

(a) Create a class called PriorityQueue. Our constructor doesn't need to do very much. Just create an empty list representing the heap array as the sole attribute (for now). 'self' is basically Python's equivalent of 'this' in Java. If you want a class to reference something about itself, be it it's attributes or methods, you always need to preface them with self. Our list attribute will need to contain 2-tuples as it's entries, instead of just numbers - i.e. array entries will be of the form (task, priority) where the first entry is whatever object is in the queue, and priority is the number attached as a priority. Try creating an instance of your

class and adding some things to the array. (You can do it directly; 'getter' and 'setter' methods are not necessary in Python.)

(b) Create additional methods __left(self, i), __right(self, i), and __max_heapify(self, i) which do what they're supposed to do to the array attribute. You should be able to copy-paste the code from the previous problem and make minor changes to accomplish this.

A short explanation of the strange naming: Python doesn't enforce public or private method distinctions, but the initial double underscore prefixes here are the industry standard way of saying that a particular method or attribute is *intended* to be private. These should be private methods because they are only meant to work in the service of our actual queue operations. A user of our priority queue shouldn't be messing around with these heap operations. They should instead be using the front-end methods relating to the queue: enqueue(), dequeue() and so forth. Therefore, we make declare the internal heap methods private.

(c) Let's think about implementing our first actual priority queue operation: enqueue(self, task, priority). Immediately we are faced with a problem: where do we put a new item? We can't just put it at the top, because this would disrupt our interpretation of the array as a binary tree. Our only choice is to attach it as a leaf using python's append() list method. But this means we need a new heap function, because the __max_heapify operation only let's nodes *fall down* from above. This item we add as a leaf is already at the bottom, so this won't do. What we need is a function which does the opposite. Write a new private method called __bubble_up(self, i), which performs swaps of a node with it's parent, moving it up until it reaches a position satisfying the max-heap property. Your function should look quite similar to max_heapify and have the same recursive structure. For the sake of aesthetics, you might also consider renaming __max_heapify to __sift_down. What is the runtime of this method?

(d) Create public methods enqueue(self, task, priority) and dequeue(self, task, priority) which implement the priority queue operations. Enqueue should make use of the bubble_up method from part (c). It's only fitting then that Dequeue should use sift_down. Also make a very simple one line method called get_max(self) which returns the first item in the array without deleting it.

(e) The last method our priority queue needs is a method which lets us increase or decrease the priority of existing tasks in the queue. This method is unfortunately the most difficult to implement. The issue is *finding* a particular task in the heap which might not be the first or the last one. Without any extra machinery, this task would take worst-case time $O(n)$, which quite slow. Fortunately, we can make use of hash tables (i.e. python dictionaries) in order to do better!

Add a second attribute, called data_locations (you're free to come up with a better name), to be initialized as an empty dictionary within the constructor. This dictionary should contain tasks as keys, and their associated values should be indices of the array where the tasks are currently located. Then modify *all* of the other necessary methods so that they fill in, edit, and delete items from the

dictionary as needed to keep it current. Since python dictionaries are implemented using open-address hash tables, all of the basic methods are $O(1)$, and therefore nothing you add to your methods should be effect their big-O runtimes on average, *if* you make the correct changes.

(f) Now make the final method, increase_priority(self, task, k), which uses the dictionary attribute to find the task in the queue, and then adds k to it's priority (k can be negative if the priority is to be decreased), and then either sifts the item down or bubbles it up - whatever is necessary to make sure it ends up in a new spot which satisfies the max-heap property.

Congratulations! You have created a working and very fast priority queue. This will come in very handy later. Don't lose it!

You might be wondering if it was really necessary to build our priority queue from scratch rather than using the heapq module. I tried that route first when making this problem, and found their implementation to be kind of esoteric and strangely clunky. You're welcome to have a look their documentation and see for yourself if you agree with my decision. That said, please do the problem as instructed.

For the rest of this assignment (as well as this class) you are free to use the heapq module as much or as little as you want.

(5) (Leetcode 215) Consider the problem of returning the k^{th} largest element of an array.

(a) Use your favorite $n\log(n)$ sorting algorithm to solve this problem in time $O(n\log(n))$. This solution is clear immediately, but isn't sorting the entire list more work than is actually necessary just for finding the k^{th} largest number?

(b) Find an incrementally faster solution by making use of a max-heap. By incrementally, I mean that it should operate in time $O(k\log(n))$. This is still $O(n\log(n))$ in the worst case, but if $k < \frac{1}{2}n$ it's saving quite a bit of time, and only doing necessary operations.

(c) Find a different solution by making use of a min-heap instead of a max-heap, and which operates in time $O(n\log(k))$. I believe this could be considered a greedy solution to the problem, but I haven't looked very closely at that material yet.

(d) Which solution is better, between (b) and (c)? Argue your case theoretically, and then show it empirically by plotting test runs of all three against each other using matplotlib. Include a plot in your write-up.

(Not part of the assignment, but there is also a divide-and-conquer solution to this problem which makes use of the partition function for quicksort, and which operates in worst case quadratic time but *average case linear time*, assuming random choice of an initial pivot. Can you find it?)

(6) Recall the majority problem from the previous homework. The task was to write an algorithm which returned the majority element of an array when one existed.

(a) Write a second algorithm for majority which makes use of hash tables rather than divide and conquer, and which works in time $O(n)$.

(b) Justify this runtime theoretically, and verify that it is faster than your divide and conquer algorithm from the previous homework. Towards doing this, write a random input generator function which generates random lists of a given size which consists of the characters a, b, and c. Then have a testing function record the times of both algorithms being run on the same lists of sizes between 10 and 500 (just one list per size is fine). Submit all functions as well as the generated plot.

(c) Is the divide and conquer solution strictly worse? Not exactly. State and justify with words the *space* complexity of both algorithms.