Section 4 Overview

Agenda
- Heaps and Deques
- Sorting - Insertion and Selection Sort Pseudocode
- Master Theorem Review
- Pseudocode Review
- Optional Problems

Heap Clarifications
- Remember you will be implementing an adaptable priority queue. This means that we must be able to remove any internal nodes, not just the min or the max
- What makes that different? You must reorganize the heap. This means upheaping and downheaping when necessary

Deques
Deques are a data structure that will become very useful to you (especially while implementing heap!). They are essentially stacks, but you can push and pop from either side (like you could a deck of cards).

Using Deques to Add to the Heap

![Diagram showing the process of adding a node to a deque and then to a heap.](image)
Using Deques to Remove from the Heap

**DELETE 1**

Root: 1

If the parent has a right child, remove right child

Pop the node off the back of the deque, and check the parent of the removed node

**ADD 2**

Root: 1

If the parent has no left child, add left child

Add the node to the back of the deque, and check the front of the deque to get the parent node

**ADD 3**

Root: 1

If the parent has a left child, add right child

Add the node to the back of the deque, and check the front of the deque to get the parent node, and then pop the parent node off front of the deque because it has no space for new child nodes
Sorting and Master Theorem

Take a look at lecture slides for the pseudocode on insertion and selection sort as well as for the rules for Master Theorem! No new information here, just review :)
**Editing Pseudocode**

We went over how to write “good” pseudocode using the following example:

```plaintext
function FUNction():
    aIndex = bIndex = 0
    while aIndex < A.length:
        while bIndex < B.length:
            if A[aIndex] <= B[bIndex]:
                result add A[aIndex]
                aIndex = aIndex+1
            else:
                result add B[bIndex]
                bIndex = bIndex +1
        if aIndex < A.length:
            result = result + A[aIndex...end]
        if bIndex < B.length:
            result = result + B[bIndex...end]
    return result
```

What’s wrong with this pseudocode?
- No input parameters (what if I create array C and array D and want to manipulate those?)
- Bad name (the name should tell us what the function is doing; this one is far too vague)
- Result is never defined
- Double while loops (runtime runtime runtime)
- Not specific (where will A[aIndex] be added to result?)
- Not concise (should use aIndex++ instead of aIndex = aIndex+1)

Now here’s some improved pseudocode:

```plaintext
function merge(A, B):
    // Input: two sorted lists
    // Output: 1 sorted list
    result = []
    aIndex = bIndex = 0
    while aIndex < A.length and
        bIndex < B.length:
        if A[aIndex] <= B[bIndex]:
            result.append(A[aIndex++])
        else:
            result.append(B[bIndex++])
    if aIndex < A.length:
        result = result + A[aIndex...end]
    if bIndex < B.length:
        result = result + B[bIndex...end]
    return result
```
Optional Problems

1. Given a list of N comparable items, return a list of the M largest items from that list.

Hint: Use a minimum priority queue/heap

Solution (SPOILER ALERT! No peeking! Give the problem your best shot before scrolling): Build min-heap of capacity M that you populate as you go, removing the minimum and replacing it with the next element if that element is greater than the smallest item of the min-heap

function getMlargest(array, M):
    pq = priority queue
    for element in array:
        if (pq.size < M):
            pq.insert(element, element)
        else:
            smallest = pq.peek()
            if smallest < element
                pq.removeMin()
                pq.insert(element, element)

    largest = build array from pq
    return largest

Unrelated Note: You can use built-in data structures along with their functions! You don’t have to make and use data structures from scratch! Ex: “h = Hashset(), pq = Priority Queue” in pseudocode

2. Space vs. Runtime efficiency: Find the duplicates in a list.

Multiple solutions (no scrolling until you’ve tried it!):

○ Add to hashtable which is linear for setup and space usage but constant for each duplicate searched
○ Sort the list which is constant in space usage and O(nlogn) setup, but linear for each duplicate searched

● In what cases is it better to use each solution? If you have 100 items in the set vs. 100,000,000? How about if you’re making searching for the duplicate of 1 term vs. 500? If you’re constrained by space but not by processing speed?