Problem Set 3

All parts are due on September 27, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 3-1. [25 points] **Heap Practice**

(a) [10 points] For each array below, draw it as a left-aligned complete binary tree (according to the transformation from Lecture 5) and state whether the tree is a max-heap, a min-heap, or neither. If the tree is neither, turn the tree into a max-heap by repeatedly swapping adjacent nodes of the tree. You should communicate your swaps by drawing a sequence of trees, with each tree depicting one swap.

1. `[0, 12, 4, 23, 13, 6, 24]`
   Solution:

```
Min Heap    _____0_____
          __12__    __4__
          23   13   6   24
```

2. `[8, 11, 8, 12, 14, 9, 10]`
   Solution:

```
Min Heap    _____8_____
          __11__    __8__
          12   14   9   10
```

3. `[23, 7, 16, 4, 7, 12, 1]`
   Solution:

```
Max Heap    _____23____
          __7__     __16__
          4   7     12    1
```

4. `[9, 6, 10, 2, 7, 4, 11]`
   Solution:

```
Neither     _____9_____                 _____9_____
          __6__       _(10)_ =>       _(6)_     __11__
          2   7       4    (11)       2   (7)   4     10

       ____(9)____              _____11_____              _____11_____
=> ___7___    _(11)_ => ___7___     _(9)_   => ___7___    ___10___
   2     6    4     10   2     6     4  (10)   2     6    4      9
```

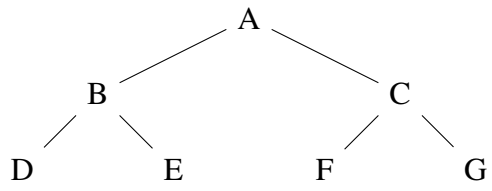
5. `[10, 2, 9, 0, 1, 8, 7]`
   Solution:

```
Max Heap    _____10_____
          __2__       __9__
          0   1       8   7
```

Rubric:

- 2 points for each correct classification

(b) [10 points] Consider the following binary tree on seven nodes labeled A – G.



Assume the tree contains the keys $K = \{1, 2, 3, 4, 5, 6, 7\}$ (each occurring exactly once) such that the min-heap property is satisfied. For each key $k \in K$, list the node(s) that could contain key k.

Solution: 1: A, 2: B, C
(3, 4, 5): B, C, D, E, F, G
(6, 7): D, E, F, G

By the min heap property, the smallest key (1) must be at the root. The node containing the largest key (7) must exist in a leaf; if it were not, its child would have a lower key. The third smallest key must exist in a child of either the node containing the second smallest (B, C) or smallest key (A).

Rubric:

- 1 points for correct locations of 1
- 3 points for correct locations of 2
- 3 points for correct locations of 3, 4, and 5
- 3 points for correct locations of 6 and 7

(c) [5 points] Given a max-heap represented by array A, and an integer key k, describe an algorithm SET_SECOND_LARGEST(A, k) that finds an item currently stored in A having a second-largest key, and modifies it to have key k, while preserving the max-heap property on A. Your algorithm should run in $O(\log n)$ time, where $n \geq 2$ is the length of input heap A. Remember to argue correctness and running time of your algorithm. Your algorithm may make use of any heap operations described in lecture or recitation as a black box, without having to repeat their description, analysis, or correctness arguments. (By second largest, we mean any key that could appear second in some sorted order.)

Solution: The simplest solution is to reduce this problem to existing priority queue operations:

```
def set_second_largest(A, k):
    largest = A.delete_max()
    second_largest = A.delete_max(A)
    second_largest.key = k
    A.insert(largest)
    A.insert(second_largest)
```

We can also solve the problem more directly (and efficiently) via heapify. By part (b), the second largest value must either be in index 1 or 2 of the array. Compare these two values to identify the appropriate index. Compare the new value to the existing value at the index. If the new value is larger, follow the `max_heapify_up` procedure on the modified node, i.e., swap with its parent as long as the value at the parent is smaller than it. Otherwise, if the new value is smaller, follow the `max_heapify_down` procedure on the modified node, i.e. swap values with its largest value child as long as a child's value is larger than it. The running time in either case is proportional to the $O(\log n)$ height of the tree, since we might swap up or down the entire height.

```
def change_second_largest(A, v):
    i = 1
    if len(A) >= 3 and A[2] > A[1]:
        i = 2
    if A[i] < v:
        A[i] = v
        max_heapify_up(A, len(A), i)
    elif A[i] > v:
        A[i] = v
        max_heapify_down(A, len(A), i)
```

Rubric:

- 8 points for a correct algorithm/pseudocode
- Partial credit may be awarded

Problem 3-2. [30 points] **Fishing Frameworks**

Frankie the Fisher frequently fishes in the fjords of Finland. To aid her business, she wants to build a mobile app called Fishr to help keep track of her inventory. For each of the following questions, describe a data structure to support the requested operations. When describing a data structure in this class, you must first describe how to store the data, and then describe an algorithm operating on the data to support each requested operation. As with any algorithm in this class, you should briefly argue correctness and running time for each supported data structure operation. For parts (a), (b), and (c), operation running times depend on the number n of fish stored in the data structure **at the time of the operation**.

(a) [5 points] Whenever Frankie catches a fish, she wants to be able to record its weight and species on the app. In addition, at any point during a fishing trip, she wants to be able to query the species of her heaviest catch on the trip so far. Describe a data structure that supports two operations: recording the weight and species of a catch, and querying the species of the heaviest catch. Both operations should run in worst-case $O(1)$ time per operation.

Solution: Frankie only needs to remember the currently heaviest fish; she can forget about all the others! When recording, if the new fish is heavier than the previous max, she replaces her record with the weight and species of the new fish. Otherwise she records nothing. Querying the heaviest fish simply returns the species of her single recorded fish. This uses $O(1)$ time per operation, and $O(1)$ space in total.

(b) [5 points] Frankie's fishing trips often last for months, so when she gets hungry, she will eat the heaviest fish in her inventory (Frankie dislikes eating small fish). Describe a data structure supporting the same two operations as part (a), plus an operation that removes the heaviest fish from her current inventory. All operations should run in worst-case or amortized $O(\log n)$ time per operation.

Solution: Use a max-heap H built with a dynamic array. Each fish is recorded as an item, where `item.key` is the fish's weight and `item.species` is the species. The species of the heaviest catch can be read in $O(1)$ time with `H.find_max().species`, and the other two operations are the standard `H.insert` and `H.delete_max`, which each use $O(\log n)$ amortized time.

(c) [10 points] Frankie the Fisher has finally figured out that fleshier fish fetch fatter fees! To finance her future funding, Frankie decides to stop eating the heaviest fish in her inventory when she's hungry, and to eat the fish with **median** weight instead. Describe a data structure supporting the same operations as part (b), except that the second and third operations apply to the median weight fish instead of the heaviest. (For even n, the operations may reference either of the two middle fish in the sorted order by weight.) All operations should run in worst-case or amortized $O(\log n)$ time per operation.

Solution: We will store a max-heap H_1 that holds the smallest $\lceil n/2 \rceil$ fish, and a min-heap H_2 that holds the largest $\lfloor n/2 \rfloor$ fish. This guarantees that the median is always

$\max(H_1)$ if n is odd, or $\max(H_1)$ AND $\min(H_2)$ if n is even.

To insert a new item (which records weight and species as before), first compare `item.key` with `H1.find_max().key`; if `item.key` is larger, then insert the item into H_2; otherwise insert into H_1. Then, restore the sizes of H_1 and H_2 if necessary with `H1.insert(H2.delete_min())` (or vice-versa). This preserves the invariant stated above. These heap operations take $O(\log n)$ time, amortized.

Deleting and returning the median fish can be accomplished with `H1.delete_max()` followed, if necessary, by the same size correction as above. These heap operations take $O(\log n)$ time, amortized.

(d) [10 points] Frustration! Many of Frankie's fish tanks have flung overboard in a furious storm, and she now only has enough space to store at most k fish on her boat at any given time, even though she may catch $n \gg k$ fish during her trip. She resolves to continue fishing, but to only keep the k heaviest fish that she sees on the trip. After catching the first k fish, each time she catches a new fish, she must choose some fish to throw back (or eat), which may or may not be the fish she just caught. Describe a data structure supporting two operations: adding a new catch to her inventory, and identifying a fish to discard, in order to maintain the heaviest k fish on her boat at all times. Be sure to argue why Frankie ends up with the k heaviest fish at the end of her trip! All operations should run in worst-case or amortized $O(\log k)$ time per operation.

Solution: Frankie can arrange her fish by weight in a min-heap that contains the k largest fish she has caught so far. When she catches a new fish, she compares it to the fish at the root of her heap, which is the lightest fish she currently has. If the new fish is lighter, then she tosses it back to the river. Otherwise, she swaps the two fish, tosses the lightest fish, and max heapifies the new fish down the heap to restore the heap's order. With this algorithm, Frankie can find the lightest fish to toss in constant time, and inserting new fish to her heap takes $O(\log k)$ time. Since she always tosses the lightest fish out, she guarantees that she'll end up with the heaviest k fish she can find.

Rubric:

- 4 points (parts a,b) / 8 points (parts c,d) for a correct data structure and argument
- 1 point (parts a,b) / 2 points (parts c,d) for runtime analysis
- Partial credit may be awarded

Problem 3-3. [45 points] **Proximate Sorting**

An array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted, i.e., if the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$. In this problem, we will show how to sort a k-proximate array faster than $\Theta(n \log n)$.

(a) [5 points] Prove that insertion sort (as presented in this class, without any changes) will sort a k-proximate array in $O(nk)$ time.

Solution: To prove $O(nk)$, we show that each of the n insertion sort rounds swap an item left by at most $O(k)$. Below are lots of ways to prove this claim!

1. In the original ordering, entries that are $\geq 2k$ slots apart must already be ordered correctly: indeed, if $A[s] > A[t]$ but $t - s \geq 2k$, there is no way to reverse the order of these two items while moving each at most k slots. This means that for each entry $A[i]$ in the original order, fewer than $2k$ of the items $A[0], \ldots, A[i-1]$ are less than $A[i]$. Thus, on round i of insertion sort when $A[i]$ is swapped into place, fewer than $2k$ swaps are required, so round i requires $O(k)$ time.
2. It's possible to prove a stronger bound: that $a_i = A[i]$ is swapped at most k times in round i (instead of $2k$). This is a bit subtle: the final sorted index of a_i is at most k slots away from i (from the k-proximate assumption), but a_i may not move to its final position immediately—it may move **past** its final sorted position and then be bumped rightward in future rounds. We'll proceed by contradiction. Assuming any rounds perform more than k swaps, let round i be the **latest** round that performs more than k swaps. Then this entry a_i (initially positioned at $A[i]$) moves to index $j < i - k$ during its round, and since all future rounds make at most k swaps, this entry a_i is too far left in the array to be reached in later rounds. But then a_i starts at index i and ends up at index $j < i - k$, violating the k-proximate condition.
3. Here's another way to prove the stronger bound. Suppose for contradiction a loop swaps the pth largest item $A[i]$ to the left by more than k to position $p' < i - k$, past at least k items larger than $A[i]$. Since A is k-proximate, $i - p \leq k$, i.e. $i - k \leq p$, so $p' < p$. Thus at least one item less than $A[i]$ must exist to the right of $A[i]$. Let $A[j]$ be the smallest such item, the qth largest item in sorted order. $A[j]$ is smaller than $k+1$ items to the left of $A[j]$, and no item to the right of $A[j]$ is smaller than $A[j]$, so $q \leq j - (k+1)$, i.e. $j - q \geq k+1$. But A is k-proximate, so $j - q \leq k$, a contradiction.
4. A final argument that does not use contradiction uses the Fact™ that if A is k-proximate, then the subarray $A[:\ i+1]$ is also k-proximate. (This is proved below.) Assuming $A[:\ i+1]$ is k-proximate, we conclude that the last element $A[i]$ is among the $k+1$ largest elements in $A[:\ i+1]$ and therefore gets swapped at most k times during round i. To prove the Fact™, it's enough to chop off one element at a time: assuming $A = [a_0, \ldots, a_n]$ is k-proximate, let's prove that $A' = [a_0, \ldots, a_{n-1}]$ is also k-proximate. Let m be the sorted index of a_n, and we know $|n - m| \leq k$ because A is k-proximate. Consider another item a_s in A (which is also the sth item in A'), and say it has sorted index t in A and t' in A'. Note that $t' = t - 1$ if $t > m$ (deletion affected this element) and otherwise $t' = t$. We have $|s - t| <= k$ since A is k-proximate, so if a_s violates the k-proximate condition in A', then we must have $|s - t'| = |s - t| + 1 = k + 1$. This only happens if

$t > m$ (so that $t' = t - 1$) and $s > t$. But then s and t are both strictly between m and n, meaning $|s - t| \leq k - 2$, so $|s - t'| \leq k - 1$ is still within allowable bounds. So a_s does not violate k-proximity in A', as desired.

Rubric:

- 5 points for a correct argument
- Partial credit may be awarded

(b) [15 points] $\Theta(nk)$ is asymptotically faster than $\Theta(n^2)$ when $k = o(n)$, but is not asymptotically faster than $\Theta(n \log n)$ when $k = \omega(\log n)$. Describe an algorithm to sort a k-proximate array in $O(n \log k)$ time, which is always nonstrictly faster than $\Theta(n \log n)$. (Remember to argue the correctness and running time of your algorithm.)

Solution: We perform a variant of heapsort, where the heap only stores $k + 1$ items at a time. Build a min-heap H out of $A[0], \ldots, A[k - 1]$. Then, repeatedly, insert the next item from A into H, and then store `H.delete_min()` as the next entry in sorted order. So we first call `H.insert(A[k])` followed by `B[0] = H.delete_min()`; the next iteration calls `H.insert(A[k+1])` and `B[1] = H.delete_min()`; and so on. (When there are no more entries to insert into H, do only the `delete_min` step.) B is the sorted answer.

This algorithm works because the ith smallest entry in array A must be one of $A[0]$, $A[1], \ldots, A[i+k]$ by the k-proximate assumption, and by the time we're about to write $B[i]$, all of these entries have already been inserted into H (and some also deleted). Assuming entries $B[0], \ldots, B[i - 1]$ are correct (by induction), this means the ith smallest value is still in H while all smaller values have already been removed, so this ith smallest value is in fact `H.delete_min()`, and $B[i]$ gets filled correctly.

Each heap operation takes time $O(\log k)$ because there are at most $k + 1$ items in the heap, so the n insertions and n deletions take $O(n \log k)$ total.

Rubric:

- 12 points for a correct algorithm and argument
- 3 points for runtime analysis
- Partial credit may be awarded

(c) [25 points] Write a Python function `proximate_sort` that implements your algorithm from part (b). You can download a code template from the webiste containing some test cases. You may adapt any code presented in lecture or recitation, but for this problem, **you may NOT import external packages** and **you may NOT use Python's built-in sort functionality** (the code checker will remove `List.sort` and `sorted` from Python prior to running your code). Submit your code online at `alg.mit.edu`.

Solution:

```
def min_heapify_up(A, c):
    if c > 0:
        p = (c - 1) // 2
        if A[c] < A[p]:
            A[c], A[p] = A[p], A[c]
            min_heapify_up(A, p)

def min_heapify_down(A, p):
    if len(A) > 0:
        l = 2 * p + 1
        r = 2 * p + 2
        if l > len(A) - 1: l = p
        if r > len(A) - 1: r = p
        c = l if A[l] < A[r] else r
        if A[c] < A[p]:
            A[c], A[p] = A[p], A[c]
            min_heapify_down(A, c)

def proximate_sort(A, k):
    heap = []
    B = [None] * len(A)
    for i in range(len(A) + k):
        if i < len(A):      # insert A[i] into heap
            heap.append(A[i])
            min_heapify_up(heap, len(heap) - 1)
        if i >= k:          # extract min from heap
            heap[0], heap[-1] = heap[-1], heap[0]
            B[i - k] = heap.pop()
            min_heapify_down(heap, 0)
    return B
```

Rubric:

- This part automatically graded at `alg.mit.edu`.