Problem Set 2

All parts are due on February 22, 2019 at 6PM. 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 2-1. [25 points] **Heap Practice**

(a) [10 points] For each array below, draw it as a left-aligned complete binary tree 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, 10, 5, 23, 12, 8, 240]`
   Solution:

```
Min Heap    ______0____
           __10__    __5____
           23  12    8   240
```

2. `[17, 7, 16, 5, 6, 2]`
   Solution:

```
Max Heap    _____17_____
           __7__     __16
           5   6     2
```

3. `[7, 12, 7, 12, 14, 18, 10]`
   Solution:

```
Min Heap     _______7______
            ___12___    ___7___
            12    14    18   10
```

4. `[8, 5, 10, 7, 1, 2, 12]`
   Solution:

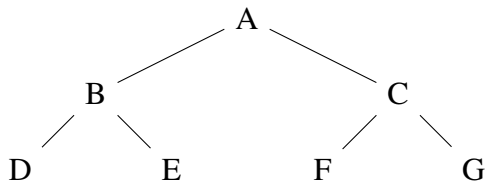
```
Neither    _____8_____            _____8_____
          __5__    _(10)_ =>    _(5)_    __12___
          7   1    2    (12)    (7)  1    2    10

       ___(8)_____            _____12____            _____12_____
=> __7__    _(12)__ => __7__      _(8)__   => __7__    __10__
   5   1    2    10     5   1      2  (10)     5   1    2    8
```

Rubric:

- 2 points for each correct classification
- 2 additional points on part 4 for correct modification to max heap
- Partial credit may be awarded

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



Each node stores a key from the multiset $X = \{6, 5, 3, 2, 6, 1, 4\}$ so as to satisfy the **max-heap property** (there are two 6s in X, so exactly two nodes will store a 6). For each node in {A, B, C, D, E, F, G}, list the key values that could be stored there.

Solution:

- A: 6
- {B, C}: 3, 4, 5, 6
- {D, E, F, G}: 1, 2, 3, 4, 5

By the max heap property, the root must contain the largest key 6 and no other key can be at the root. Since 6 is repeated the other 6 must be at B or C. 6 cannot occur at a leaf node because then a smaller key would be above it at B or C. Aside from the root, keys 3, 4, and 5 can exist anywhere. The two smallest keys, 1 and 2, cannot be at B or C or else at least one of their children would be bigger.

Rubric:

- 2 points each for correct keys at nodes A, B, C
- 1 point each for correct keys at nodes D, E, F, G

Problem 2-2. Shifty Array [5 points]

Describe a data structure that maintains a sequence of items and supports the following operations, each in $O(1)$ time. For each operation, specify whether the running time achieved by your data structure is worst-case or amortized.

- `at(i)`: return the ith item in the sequence
- `insert(x)`: add x as the first item in the sequence
- `shift()`: move the first item in the sequence to the back of the sequence

Solution: Many possible solutions. A valid solution to this problem is to reduce to using a double-ended sequence as discussed in R03, or use similar techniques to solve this problem directly. That double-ended sequence already supports `at(i)` in worst case constant time, and `insert_left(x)`, `insert_right(x)`, `delete_left(x)`, and `delete_right(x)`, all in amortized constant time. To implement `insert(x)`, simply `insert_left(x)` in amortized constant time. To implement `shift()`, simply `insert_right(delete_left())`, also in amortized constant time. Note that there is no way for a linked list to support `at(i)` in constant time, so solution must use a modified array of some kind.

Rubric:

- 1 point for each correct operation implementation (3)
- $\max(k-1, 0)$ points for k operations correctly designated worst-case/amortized

Problem 2-3. [30 points] **Consulting**

(a) [10 points] **Picky Produce:** High-end grocery store chain Fole Whoods has hired you to purchase watermelons from a wholesale distributor that sells watermelons at one price regardless of size. Fole Whoods on the other hand sells watermelons to customers by weight, at a price of d dollars per ounce, so heavier watermelons sell for a higher price. However, due to restrictive shopping safety regulations, Fole Whoods is not allowed to sell watermelons that weigh more than x ounces, even though their distributor sells them. Each week, you are given a list of p pairs (n_i, o_i) representing the ID number and integer weight in ounces of each watermelon available for purchase. Describe an $O(p \log w)$-time algorithm to determine the IDs of the w most profit-maximizing watermelons available on any given week.

Solution: We will use a min-heap of size w in order to store the pairs (n_i, o_i) corresponding to the w largest watermelons with weight up at most x, ordered by their o_i weights. First, loop through all watermelon pairs in $O(p)$ time, and if $o_i > x$, set o_i to zero, as no money can be made from that watermelon. Then for each watermelon pair (n_i, o_i), insert the pair into the min-heap in $O(\log w)$ time. If after insertion the min-heap contains more than w pairs, remove the pair with minimum o_i stored at the root in $O(\log w)$ time. Repeat until all n watermelon pairs have been processed. At termination, the min-heap contains the k largest watermelon pairs with weight at most x. This is true because the loop maintains the invariant that after processing pair i (one-indexed from 1 to n), the min-heap contains the $\min\{i, k\}$ most-profitable watermelons from the first i pairs. Then, simply loop through the pairs in the min-heap and return the ID number associated with each pair. Processing each pair takes at most $O(\log w)$ time, so processing all n pairs takes at most $O(p \log w)$ time, as desired.

Note that there is an algorithm which runs in $O(p + w \log p)$ time, which is asymptotically equivalent to $O(p \log w)$ when $w = \Omega(p)$, but is asymptotically faster when $w = o(p)$ (strict upper bound). The algorithm starts the same as the previous one, reducing weights of watermelons with $o_i > x$ in $O(p)$ time. Then, a max-heap can be

built from all p pairs in $O(p)$ time. Then, removing and returning the largest w from the max-heap yields a $O(p + w \log p)$-time algorithm as claimed.

Rubric:

- 2 points for description of a correct algorithm
- 2 points for a correct argument of correctness (even if inefficient)
- 2 points for a correct analysis of running time (even if inefficient)
- 4 points if correct algorithm is $O(p \log w)$
- Partial credit may be awarded

(b) [10 points] **Sorting Students:** 0.660 is a popular algorithms course at a very organized university. Each semester, the course splits its students (sometimes unevenly) into recitations, with each student is in exactly one TA's recitation. At the end of the term, each TA assigns an integer grade to each of their students, and submits to the instructors a list of (name, grade) pairs, one for each student in their recitation. The instructors must then combine grades from all the recitations into one sorted list in order to assign letter grade cutoffs. As the TAs are very organized, each TA submits their list of students to the instructors ordered increasing by grade. Given the sorted list from each of the r TAs, describe an $O(s \log r)$ time algorithm to construct a list containing the names of all s students, sorted by their grade in the course.

Solution: This problem requires us to merge r sorted lists into a single sorted list. To do this, we generalize the merge step of merge sort: keep track of the unprocessed student pairs from each of the r lists, and then repeatedly process the pair with lowest grade, placing it at the end of a final list. For each of the r recitations, insert the triple (grade, name, recitation) corresponding to the first student in the recitation having lowest grade, into a min-heap, processing each in $O(\log r)$ time. Then, repeat the following until every student has been processed: remove the triple (g_i, n_i, r_i) corresponding to the student with lowest grade from the root of the min-heap in $O(\log r)$ time, append name n_i to an output list in amortized constant time, and then check the student list of recitation r_i. If recitation list r_i contains any unprocessed students, process the next lowest grade student from that list and add its corresponding triple to the min-heap in $O(\log r)$ time. This loop maintains the invariant that after the ith removal from the heap (one indexed), the output list contains the names of the lowest scoring i students in sorted order. Each of the s students takes $O(\log r)$ time to process, so this algorithm runs in $O(s \log r)$ time, under the weak assumption that $r = O(s)$; otherwise, sort the students directly in $O(s \log s)$ time, which is $O(s \log r)$ when $r = \omega(s)$.

Rubric:

- 2 points for description of a correct algorithm
- 2 points for a correct argument of correctness (even if inefficient)
- 2 points for a correct analysis of running time (even if inefficient)
- 4 points if correct algorithm is $O(s \log r)$

- Partial credit may be awarded

(c) [10 points] **Unstable Stable:** Dirgah is a gamekeeper of fantastic beasts at the wizarding school of Wogharts. He has a steady flow of new animals arriving from around the world, which he houses in various pens in his stable; space is limited, so he often houses many animals in the same pen. Each animal is associated with a positive integer representing its *pedagogical value* to the school, whereas the *value of a pen* is the average value of animals housed in that pen. Whenever a new animal arrives, Dirgah houses it in any pen with lowest value. Students often come to Dirgah to request a pet. Even though Dirgah is not supposed to give out animals to students, he has a soft heart. If a lowest valued pen contains any animals, he will give the student whatever animal from that pen has lowest value. Describe a database to help Dirgah keep track of the animals in his pens supporting the following operations; p is the number of pens in the stable and n is the number of stabled animals at the time of the operation.

- `initialize(P)`: given a list `P` of p lists containing the values of animals in each pen, make a database to keep track of them in $O(k+p)$ time, where k is the number of animals in `P`.
- `add_animal(v)`: house an animal with value `v` into the lowest value pen in $O(\log n + \log p)$ time.
- `remove_pet()`: Remove a lowest value animal from the lowest valued pen in $O(\log n + \log p)$ time (if one exists).

Solution: We implement the requested database by using a min-heap of min-heaps. For each pen, we maintain an **animal** min-heap on the animals in that pen, keyed by animal value. Then we maintain one more **pen** min-heap on pairs (v_i, t_i), one for each pen p_i, keyed by pen value v_i, where t_i is a pointer to the animal min-heap associated with the pen. To perform the `initialize(P)` operation, we first build each animal min-heap in linear time (as shown in R04). Each animal exists in at most one pen, so this step takes $O(k+p)$ time. Then, compute the value of each pen, also in $O(k+p)$ time. Lastly, build the pen min-heap on (pen value, pointer) pairs keyed by pen value, which can be built in $O(p)$ time. The root of the pen min-heap corresponds to the pen with lowest value and links to its corresponding animal min-heap. To implement `add_animal(v)`, remove the lowest valued pen pair from the pen min-heap in $O(\log p)$ time, insert value v into the animal min-heap linked at the root in $O(\log n)$ time, recompute the pen value, and then reinsert the pen pair into the pen min-heap in $O(\log p)$ time. To implement `remove_pet()`, find the lowest valued pen pair in constant time (at root), remove the lowest valued animal from its linked animal min-heap (if one exists) in $O(\log n)$ time, and then recompute the pen value in constant time. Note that removing an animal cannot increase the value of the pen, so the minimum pen stays the minimum, so `remove_pet()` can be supported in $O(\log n)$ time, with no dependence on p. Operation correctnesses follow directly from the properties of min-heaps.

Rubric:

- 1 point for description of a correct data structure
- 1 point for each correct operation implementation (3)
- 1 point for each correct operation running time analysis (3)
- 1 point for each correct operation meets requested asymptotic bound (3)
- Partial credit may be awarded

Problem 2-4. [45 points] **Bear Bowl Buying**

Mama Bear is very disappointed: Silvribraids broke into her house, ate her porridge, and even stole her favorite bowl! So Mama Bear is off to the store to buy a new one. But Mama Bear can't have just any bowl; she needs one that is not too big and not too small, but just right. As Mama Bear walks around the store, she will look at each bowl and enter its positive integer size into a database on her phone. At any point during her shopping, she would like to know the **best** bowls she's seen so far: specifically, k bowls that are neither the $\lceil \frac{n-k}{2} \rceil$ largest nor the $\lfloor \frac{n-k}{2} \rfloor$ smallest of the n bowls she has seen so far (or all n bowls when $n \leq k$). In this problem, you will design a database to help Mama Bear shop, supporting the following two operations:

- `record_bowl(s)`: add a bowl of size s to the database
- `best_bowls()`: return the sizes of the k best bowls in the database

(a) [5 points] Mama Bear can't find her heaps anywhere! Describe in a data structure that **does not use heaps**, supporting `record_bowl(s)` in $O(n)$ time and `best_bowls()` in $O(k)$ time.

Solution: Simply maintain the set of bowl sizes in a sorted dynamic array. To insert a bowl, insert it into the array in $O(n)$ time by appending to the end and swapping it down. To retrieve the k best bowl, either return all the bowls if $n < k$, or go to index $\lfloor \frac{n-k}{2} \rfloor$ (zero indexed) and return the k bowls starting from that location in $O(k)$ time.

Rubric:

- 1 point for description of a correct data structure
- 1 point for each correct operation implementation (2)
- 1 point for each correct operation running time analysis (2)
- (operations must acheive requested bounds to receive points)
- Partial credit may be awarded

(b) [15 points] Luckily, Mama Bear has found some heaps in the cupboard! Describe a data structure that **uses heaps**, supporting `record_bowl(s)` in $O(k + \log n)$ time and `best_bowls()` in $O(k)$ time.

Solution: We will maintain a data structure to separately maintain (1) the best bowls, (2) the bowls smaller than the best, and (3) the bowls bigger than the best. We will use a sorted array A_{best} to store the best bowls, and then two max-heaps A_{small} and

A_{large} to store the small and large bowls respectively (by storing the negative of bowl sizes in A_{large} means that we only need to implement a max-heap, not a min-heap). If we can maintain this data structure, we can implement `best_bowls()` directly by reading off the bowls from A_{best} in $O(k)$ time. To implement `record_bowl(s)`, first check whether A_{best} already has k bowls; if not, insert s into A_{best} by appending to the end and swapping down in $O(k)$ time (we refer to this operation as `insert_sorted`. Otherwise, there are three cases: s is either smaller, larger, or within the range of current best bowls. If s is smaller, insert s into heap A_{small} in $O(\log n)$ time, remove the maximum from A_{small} in $O(\log n)$ time, and then `insert_sorted` that maximum at the front of A_{best} in $O(k)$ time. If s is larger, insert $-s$ into heap A_{large} in $O(\log n)$ time, remove the maximum from A_{large} in $O(\log n)$ time, and then `insert_sorted` the negative of that maximum at the back of A_{best} in amortized $O(1)$ time. If s is within the range of current best bowls, simply `insert_sorted` s into A_{best} in $O(k)$ time. After any of these cases, because of the correctness of max-heap `delete_max`, $k+1$ bowls exist in A_{best}, with no bowl in $A_{smaller}$ larger than any bowl in A_{best}, and no bowl in A_{larger} smaller than any bowl in A_{best}. Then to maintian the definition of best, if the number of bowls in A_{small} and A_{large} are equal, remove the largest bowl from A_{best} in amortized $O(1)$ time and insert its negative into A_{large}; otherwise, remove the smallest bowl from A_{best} in $O(k)$ time, and insert it into A_{small}. This algorithm performs a constant number of heap operations and at most a constant number of linear time operations on a sorted array of size $O(k)$, so runs in $O(k + \log n)$ time.

Rubric:

- 3 points for description of a correct data structure
- 1 point for a correct implementation of `best_bowls()`
- 1 point for a correct analysis running time for `best_bowls()`
- 1 point if correct implementation of `best_bowls()` is $O(k)$
- 4 points for a correct implementation of `record_bowl(s)`
- 2 point for a correct analysis running time for `record_bowl(s)`
- 3 point if correct implementation of `record_bowl(s)` is $O(k + \log n)$
- Partial credit may be awarded

(c) [25 points] Implement your `record_bowl(s)` and `best_bowls()` methods from part (b) in a Python class `MamaBearDB`. You can download a code template containing some test cases from the website. 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`.

Rubric:

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

Solution:

```
class MamaBearDB:

    def __init__(self, k):
        "Initialize database"
        self.k = k
        self.small = [] # max heap
        self.best  = [] # sorted array
        self.large = [] # max heap

    def record_bowl(self, s):
        "Add bowl with size s"
        if len(self.best) < self.k:
            insert_sorted(self.best, s)
            return
        if s < self.best[0]:
            insert_max_heap(self.small, s)
            insert_sorted(self.best, delete_max_heap(self.small))
        elif s > self.best[-1]:
            insert_max_heap(self.large, -s)
            insert_sorted(self.best, -delete_max_heap(self.large))
        else:
            insert_sorted(self.best, s)
        if len(self.small) == len(self.large):
            insert_max_heap(self.large, -self.best.pop(-1))
        else:
            insert_max_heap(self.small, self.best.pop(0))

    def best_bowls(self):
        "Return the self.k best bowls"
        return tuple(self.best)

def insert_sorted(A, s):
    i = len(A)
    A.append(s)
    while i > 0 and A[i] < A[i - 1]:
        A[i], A[i - 1] = A[i - 1], A[i]
        i = i - 1

def insert_max_heap(A, x):
    A.append(x)
    max_heapify_up(A, len(A) - 1)

def delete_max_heap(A):
    A[0], A[-1] = A[-1], A[0]
    out = A.pop()
    if len(A) > 0:
        max_heapify_down(A, 0)
    return out

```

```
50 def max_heapify_up(A, c):
51     p = (c - 1) // 2
52     if c > 0 and A[p] < A[c]:
53         A[c], A[p] = A[p], A[c]
54         max_heapify_up(A, p)
55 
56 def max_heapify_down(A, p):
57     l = 2 * p + 1 if 2 * p + 1 < len(A) else p
58     r = 2 * p + 2 if 2 * p + 2 < len(A) else p
59     c = l if A[r] < A[l] else r
60     if A[p] < A[c]:
61         A[c], A[p] = A[p], A[c]
62         max_heapify_down(A, c)
```