Problem Set 2

All parts are due on September 26, 2017 at 11:59PM. 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.csail.mit.edu .

Problem 2-1. [25 points] **Unbalanced Sort**

Consider an algorithm UNBALANCEDSORT, identical to MERGESORT except that, instead of dividing an array of size n into two arrays of size $n/2$ to recursively sort, we divide into two arrays with sizes roughly $n/3$ and $2n/3$. For simplicity, assume that n is always divisible by divisors (i.e. you may ignore floors and ceilings).

(a) [2 points] **Recurrence:** Write down a recurrence relation for UNBALANCEDSORT. Assume that merging takes $\Theta(n)$ time.

(b) [13 points] **Analysis:** Show that the solution to your recurrence relation is $\Theta(n \log n)$ by drawing out a recursion tree, assuming $T(1) = O(1)$. Note, you need to prove both upper and lower bounds.

(c) [5 points] **Generalization:** Suppose that UNBALANCEDSORT is instead divided into two arrays of size $n/4$ and $3n/4$. How does the asymptotic runtime of the algorithm change? What about dividing into n/a and $(a-1)n/a$ for arbitrary constant $1 < a$?

(d) [5 points] **Limitation:** Now suppose that UNBALANCEDSORT divided into two arrays of size a and $n-a$ for some positive constant integer a. How does the asymptotic runtime of the algorithm change? Assume that merging still takes $\Theta(n)$ time, and $T(a) = O(a)$. It may help to draw a new recursion tree.

Problem 2-2. [25 points] **Heap Practice**

(a) [10 points] For each array, state whether it is a max heap, a min heap, or neither.

- $a_0 = [7, 6, 4, 1, 5, 2, 3]$
- $a_1 = [2, 4, 6, 5, 1, 7, 3]$
- $a_2 = [6, 7, 3, 1, 5, 2, 4]$
- $a_3 = [1, 2, 5, 3, 4, 7, 6]$
- $a_4 = [7, 3, 6, 1, 5, 2, 4]$

(b) [5 points] Draw the complete binary tree representation for array a_1, and show how to turn it into a min heap by repeatedly swapping adjacent nodes in tree (i.e. run BUILDMINHEAP).

(c) [4 points] What are the minimum and maximum number of elements that can be contained in a heap with height h? By height, we mean the length of the longest root to leaf path.

(d) [6 points] Draw a complete binary tree on 7 nodes. If the nodes in your tree have unique keys that satisfy the max heap property, draw a triangle around each node that could contain the second largest key, a square around each node that could contain the third largest key, and a circle around each node that could contain the smallest key.

(e) [5 points] Given a max heap on n elements, MAXIMUM returns a reference to the maximum element in the heap, while EXTRACTMAX extracts the maximum element from the heap so that the heap remains a max heap. Briefly describe the asymptotic running time for each of these procudures, and argue why they are different in the worst case.

Problem 2-3. [50 points] **Generalized Priority Queue**

For this problem, a Priority Queue is a data structure supporting three operations:

- INSERT inserts an integer into the queue in $O(\log n)$ time,
- BEST returns a copy of the best integer in $O(1)$ time, and
- EXTRACTBEST removes the best integer in $O(\log n)$ time,

where n is the number of integers in the data structure at the time of the operation. Different Priority Queues have different definitions of best. For example, a Min Priority Queue is a Priority Queue where best means the minimum integer contained in the Queue.

(a) [5 points] **Modify:** The code template provided on the website includes an implementation of a `MinPriorityQueue`. Modify that code to implement a Python class `MaxPriorityQueue` that supports operations: `best`, `insert`, and `extract_best`, where the best integer is the maximum integer contained in the Queue.

(b) [20 points] **Algorithm:** A Median Priority Queue is a Priority Queue where best means the $\lceil \frac{n}{2} \rceil$th smallest integer contained in the Queue. Describe how to implement a Median Priority Queue using other Priority Queues, and describe algorithms for INSERT, BEST, and EXTRACTBEST, and confirm their asymptotic running time.

(c) [5 points] **Generalize:** A r-Priority Queue is a Priority Queue where best means the $\lceil rn \rceil$th smallest integer contained in the Queue, for a fixed $0 < r \leq 1$. Then a $\frac{1}{n}$-Priority Queue is a Min Priority Queue, a 0.5-Priority Queue is a Median Priority Queue, and a 1-Priority Queue is a Max Priority Queue. Modify the Median Priority Queue you described in part (b) to implement an r-Priority Queue for any fixed r, $0 < r \leq 1$.

(d) [20 points] **Implement:** Write a Python class `RPriorityQueue` that implements a r-Priority Queue.

Please submit all code for this problem in a single file to alg.csail.mit.edu; be sure your submission includes class definitions for:

`MinPriorityQueue`, `MaxPriorityQueue`, and `RPriorityQueue`.

```
##
from math import ceil

class MinPriorityQueue:
    def __init__(self):     # initialize
        self.A = []         # list where heap is stored

    def best(self):         # returns best, without modifying heap
        if len(self.A) < 1:
            return None     # return None if queue empty
        return self.A[0]

    def insert(self, v):    # inserts v, maintaining heap
        self.A.append(v)
        node = len(self.A) - 1
        parent = (node - 1) // 2
        while (0 <= parent) and (self.A[node] < self.A[parent]):
            self.A[parent], self.A[node] = self.A[node], self.A[parent]
            node = parent
            parent = (node - 1) // 2

    def extract_best(self): # removes best, maintaining heap
        if len(self.A) < 1:
            return None     # return None if queue empty
        node = 0
        out = self.A[node]
        self.A[node] = self.A[-1]
        self.A.pop()
        while True:
            left  = 2 * node + 1
            right = 2 * node + 2
            best = node
            if right < len(self.A) and self.A[right] < self.A[best]:
                best = right
            if left  < len(self.A) and self.A[left]  < self.A[best]:
                best = left
            if node != best:
                self.A[best], self.A[node] = self.A[node], self.A[best]
                node = best
            else:
                return out

class MaxPriorityQueue:
    def __init__(self):     # Implement me
        pass
    def best(self):         # Implement me
        pass
    def insert(self, v):    # Implement me
        pass
    def extract_best(self): # Implement me
        pass

class RPriorityQueue:
    def __init__(self, r):  # Implement me
        self.r = r          # r should not change after initialization
        pass
    def best(self):         # Implement me
        pass
    def insert(self, v):    # Implement me
        pass
    def extract_best(self): # Implement me
        pass
##
```