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.

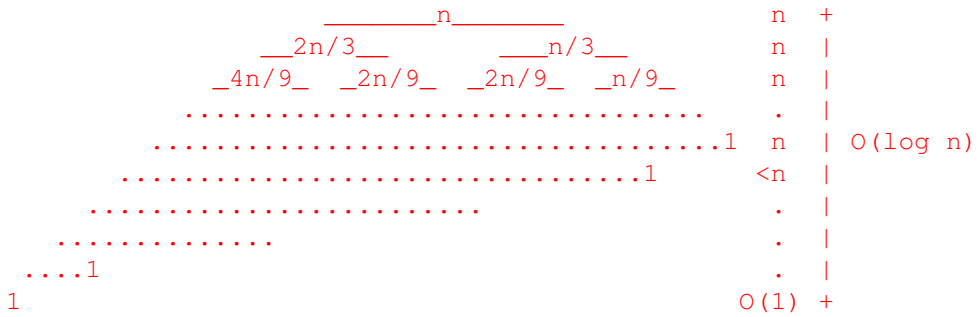
Solution: $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

Rubric:

- 2 points for correct recurrence relation

(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.

Solution:



The root does no more than cn work for some positive constant c. Then the second level does no more than $c(n/3)+c(2n/3) = cn(1/3+2/3) = cn$ work, the third level no more than $c(n/9) + c(2n/9) + c(2n/9) + c(4n/9) = cn(1/3 + 2/3)^2 = cn$, and so on. In fact, each level requires at most cn work, though may do significantly less work near the bottom of the unbalanced tree. The longest root to leaf path has height

$\log_{3/2} n = O(\log n)$, leading to a $O(n \log n)$ upper bound. The work is also lower bounded by $\Omega(n \log n)$, because the subtree above and including level $h = \log_{3/1} n$ is a complete binary tree with height h, for which each level does at least $c'n$ for some positive constant c'.

Rubric:

- 3 points for tree drawing
- 5 points for upper bound argument
- 5 points for lower bound argument

(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 $1 < a$?

Solution: If we divide into $n/4$ and $3n/4$, each level i requires at most $cn(1/4 + 3/4)^i = O(n)$ work, while the height of the tree is $\log_{4/3} n$ which is $O(\log n)$. In fact for division into n/a and $(1-a)n/a$, each level requires at most $cn(1/a + (a-1)/a)^i = O(n)$ work, while the height of the tree is $\log_{a/(a-1)} n$ which is $O(\log n)$ for any constant $a > 0$. So the asymptotic runtime is the same for both cases.

Rubric:

- 2 points for 4 spilt argument
- 3 points for a split argument

(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.

Solution: Here, the amount of work done at the root (level 0) is cn, while the work done at the next level is $ca + c(n-a) = cn$. Then the work done at level $i+1$ is $c(n-ia)$. Then the total runtime of the algorithm is:

$$cn + \sum_{i=0}^{n/a} c(n - ia) = cn + cn\left(\frac{n}{a} + 1\right) - ca\frac{(n/a)(n/a+1)}{2} = \frac{c}{a}\left(n^2 - \frac{n^2}{2a} + 2an - \frac{a^2}{2}\right) = \Theta(n^2).$$

Rubric:

- 3 points for correct argument that runtime is larger than $O(n \log n)$
- 2 point for if they argue $\Theta(n^2)$

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.

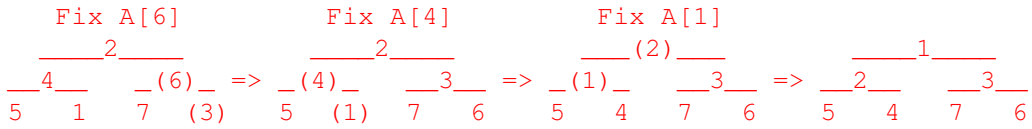
- $a_0 = [7, 6, 4, 1, 5, 2, 3]$ **Solution:** Max heap
- $a_1 = [2, 4, 6, 5, 1, 7, 3]$ **Solution:** Neither
- $a_2 = [6, 7, 3, 1, 5, 2, 4]$ **Solution:** Neither
- $a_3 = [1, 2, 5, 3, 4, 7, 6]$ **Solution:** Min heap
- $a_4 = [7, 3, 6, 1, 5, 2, 4]$ **Solution:** Neither

Rubric:

- 2 points for each correct classification

(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).

Solution: Check each node for mistakes in reverse array order, swapping at most two elements at each node. Assuming the array is zero indexed, errors occur at array positions 6, 4, and 1:



Rubric:

- 2 points drawing of tree
- 1 point for each correct swap (out of order is OK)

(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.

Solution: The heap has the maximum number of elements when it is a complete binary tree of height h:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

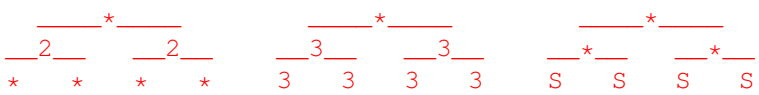
Alternatively, a heap has the minimum number of elements when only a single element is at the bottom level. There are 2^h elements on the bottom level of a complete binary tree, so the minimum number of elements is $(2^{h+1} - 1) - 2^h + 1 = 2^h$.

Rubric:

- 2 points for maximum
- 2 points for minimum

(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.

Solution: The second largest key must exist on level 2; if it were at the root, it would be the maximum key, and if it were lower, it would have more than one ancestor with larger keys. By similar reasoning, the third largest key must exist in a child of either the node containing the second largest or maximum key. Lastly, the node containing the smallest key must exist in a leaf; if had a child, its child would have a lower key.



Rubric:

- 2 points for correct second largest locations
- 2 points for correct third largest locations
- 2 points for correct smallest locations

(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.

Solution: MAXIMUM simply queries and returns a reference to the element at the root of the heap, which will always take constant time. EXTRACTMAX on the other hand removes the root from the heap in constant time, but may require $\Omega(\log n)$ swaps to ensure the heap maintains the max heap property, i.e. by running MAXHEAPIFY. So the running time of EXTRACMAX will increase with the number of items contained in the heap, while MAXIMUM will not.

Rubric:

- 2 points for correct analysis of MAXIMUM
- 3 points for correct analysis of EXTRACTMAX

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.

Solution: See below.

Rubric: This problem is graded by ALG.

(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.

Solution: Implement a Median Priority Queue using two auxiliary Priority Queues: a Max Priority Queue to store the $\lceil \frac{n}{2} \rceil$ smallest integers, and a Min Priority Queue to store the rest. We will call this the *MPQ Property*. If the MPQ Property is ever violated, we can transfer integers from one Priority Queue to the other using their respective EXTRACTBEST and INSERT methods until the property is satisfied. We call this the FIX procedure.

Storing the data in this way, BEST can simply look at BEST of the Max Priority Queue as the median is the largest of the $\lceil \frac{n}{2} \rceil$ smallest. This algorithm is clearly $O(1)$.

EXTRACTBEST will call EXTRACTBEST on the Max Priority Queue. As n decreases by 1, the MPQ property may not be satisfied, so we fix it using FIX. Because the change is only off by one, FIX only needs to transfer at most one integer between Priority Queues. Since INSERT may need to be called on auxiliary Priority Queues once, and EXTRACTBEST may need to be called twice, the running time is $3O(\log n) = O(\log n)$.

To INSERT integer v, compare it to the BEST of the Max Priority Queue. If it is smaller, INSERT into the Max Priority Queue; otherwise, INSERT into the Min Priority Queue. Again, the modification may violate the MPQ property, so a FIX operation

may be needed. A similar analysis as EXTRACTBEST shows that this algorithm is also $O(\log n)$.

Rubric:

- 4 points for using Min and Max Priority Queues
- 4 points for BEST algorithm and analysis
- 6 points for EXTRACTBEST algorithm and analysis
- 6 points for INSERT algorithm and analysis

(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, $0 < r \leq 1$.

Solution: Use the same structure as the Median Priority Queue except change the MPQ property to ensure that the auxiliary Max Priority Queue contains the $\lceil rn \rceil$ smallest elements. Everything else is the same.

Rubric:

- 5 points for a correct argument

(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`.

Rubric: This problem is graded by ALG.

```
##
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
##
```

Solution:

```
##
class MaxPriorityQueue:
    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 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[parent] < self.A[node]):      # THIS LINE CHANGED
            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
        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[best] < self.A[right]:  # THIS LINE CHANGED
                best = right
            if left  < len(self.A) and self.A[best] < self.A[left]:   # THIS LINE CHANGED
                best = left
            if node != best:
                self.A[best], self.A[node] = self.A[node], self.A[best]
                node = best
            else:
                return out

class RPriorityQueue:
    def __init__(self, r):  # Impliment me
        self.r = r          # r should not change after initialization
        self.maxPQ = MaxPriorityQueue() # stores lowest ceil(r * n) integers
        self.minPQ = MinPriorityQueue() # stores the rest

    def best(self):         # Implement me
        return self.maxPQ.best()

    def insert(self, v):    # Implement me
        if len(self.maxPQ.A) == 0 or v <= self.maxPQ.best():
            self.maxPQ.insert(v)
        else:
            self.minPQ.insert(v)
        self.balance()

    def extract_best(self): # Implement me
        out = self.maxPQ.extract_best()
        self.balance()
        return out
```

```
    def balance(self):
        maxN = len(self.maxPQ.A)
        minN = len(self.minPQ.A)
        if   maxN < ceil(self.r * (maxN + minN)):
            self.maxPQ.insert(self.minPQ.extract_best())
        elif maxN > ceil(self.r * (maxN + minN)):
            self.minPQ.insert(self.maxPQ.extract_best())
##
```