Problem Set 2

All parts are due on February 22, 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 `py.mit.edu/6.006`.

Problem 2-1. [30 points] **Counting Inversions**

Given array `A` containing n integers, an **inversion** is a pair of indices (i, j) with $i < j$ for which `A[i] > A[j]`, i.e., a larger number `A[i]` appears *before* a smaller one `A[j]` in the array. The total number of inversions is a measure of how far an array is from being sorted.

(a) [3 points] List all inversions in the array below (assume zero-based indexing).

```
[4,5,3,1,6,3,6,8,4]
```

Solution:

```
(0,2),(0,3),(0,5),
(1,2),(1,3),(1,5),(1,8),
(2,3),
(4,5),(4,8),
(6,8),
(7,8)
```

Rubric:

- $\lfloor k/4 \rfloor$ points for k correct inversions
- $-\lceil k/4 \rceil$ points for k incorrect inversions

(b) [5 points] Give an $O(n^2)$ algorithm to count the number of inversions in an array.

Solution: There are only $\binom{n}{2} = O(n^2)$ pairs (i, j) where i is less than j. Then an $O(n^2)$ brute force solution would be to loop through each pair and check in constant time whether it is an inversion.

Rubric:

- 4 points for a correct algorithm
- Partial credit may be awarded
- 1 point for arguing running time

(c) [8 points] Given array `A` whose left and right halves, `A[:n/2]` and `A[n/2:]`, are each sorted, give a linear time algorithm to count the number of inversions in `A`.

Solution: There are no inversions in the first half or second half as they are sorted. Thus the only inversions occur between some element i in the first half and element j in the second half. Scan left pointer i and right pointer j across each half of the array element by element in order from left to right, similar to the merge procedure in merge sort. During the scan, we maintain the invariant that all inversions involving elements $j' < j$ in the second half of the array have been counted. This is exhaustively true at the start when j points to the left-most element of the right half of the array.

Now assume that we have counted all inversions involving $j' < j$. If $A[i] < A[j]$, element i does not form an inversion with j or any element to j's right, so we increment i and maintain the invariant. Alternatively, if $A[i] > A[j]$, then element j forms an inversion with i and every element to i's right in the first half of the array. We then add this number to a count and increment j, thus preserving the invariant. Because we increment a pointer over each element of the array exactly once, performing constant work each time, this algorithm runs in linear time.

Rubric:

- 4 points for a correct algorithm
- 3 points for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

(d) [8 points] Give an $O(n \log n)$ algorithm to count the number of inversions in an array.

Solution: We augment the merge sort algorithm by separating the array into halves, sorting and counting inversions in each half inductively, counting inversions across the halves using the counting algorithm described in the previous part, completing the sort by merging them as in merge sort, and returning the sum of the three counts. Correctness is argued in the section below. This algorithm remains $O(n \log n)$ because the counting function and the merge step can both be done in linear time, leading to the same asymptotic running time as merge sort.

Rubric:

- 6 points for a correct algorithm
- 2 points for analysis of running time
- Partial credit may be awarded

(e) [6 points] Prove that your $O(n \log n)$ algorithm is correct, i.e., returns the correct number of inversions.

Solution: We prove that the algorithm returns the correct number of inversions **and** sorts the array by inducting on the size of the array. When the array contains only one element which is sorted, it cannot participate in an inversion, and the algorithm correctly returns a count of zero. Now consider an array of length n, assuming that

the claim is true for the algorithm run on arrays of length $< n$. Each inversion exists between two elements i and j, which can either both be in the left half, both in the right half, or one is in each. Sorting and counting inversions in each half correctly counts the first two types of inversion by induction. Then the counting algorithm and merging sorts and correctly counts the inversions across the halves by part (c), so summing these three correctly counts all three types of inversions, completing the proof.

Rubric:

- 6 points for a correct proof
- Partial credit may be awarded

Problem 2-2. [30 points] **Heap Practice**

(a) [10 points] For each array below, draw it as a binary tree and state whether it is a max heap, a min heap, or neither. If an array is neither, turn it into a max heap by repeatedly swapping adjacent nodes of the tree (i.e. run `build_max_heap`). You should communicate your swaps by drawing a sequence of trees, with each tree depicting one swap.

1. `[11, 2, 9, 1, 0, 3, 8]`
   Solution:

   ```
   Max Heap    _____11_____
              __2__     __9__
              1   0     3   8
   ```

2. `[1, 3, 2, 4, 5, 7, 6]`
   Solution:

   ```
   Min Heap    _____1_____
              __3__     __2__
              4   5     7   6
   ```

3. `[2, 4, 6, 5, 9, 7, 8]`
   Solution:

   ```
   Min Heap    _____2_____
              __4__     __6__
              5   9     7   8
   ```

4. `[5, 6, 7, 1, 8, 2, 9]`
   Solution:

```
Neither     _____5_____          _____5_____
          __6__     _(7)_ => _(6)_     __9__
          1   8     2   (9)    1   (8)   2   7

       ____(5)____          _____9_____          _____9_____
=> ___8___    _(9)_ => ___8___    _(5)_  => ___8___    ___7___
   1    6    2   7     1    6    2   (7)     1    6    2    5
```

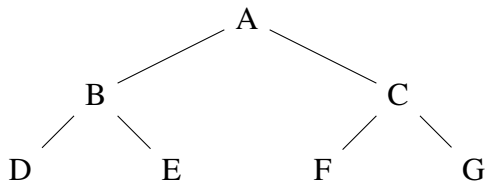
5. `[10, 3, 9, 1, 2, 7, 4]`
   Solution:

```
Max Heap    _____10_____
          __3__     __9__
          1   2     7   4
```

Rubric:

- 2 points for each correct classification

(b) [12 points] Consider the following binary tree on seven nodes labeled A – G.



Assume each tree node contains a unique key that satisfies the max heap property. For each of the following, list the node(s) that could contain the key described.

1. The heap's largest key
   Solution: A
2. The heap's second largest key
   Solution: B, C
3. The heap's third largest key
   Solution: B, C, D, E, F, G
4. The heap's smallest key
   Solution: D, E, F, G

Solution: By the max heap property, the largest key must be at the root. 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 it were not, its child would have a lower key.

Rubric:

- 2 points for correct largest locations
- 3 points for correct second largest locations
- 4 points for correct third largest locations
- 3 point for correct smallest locations

(c) [8 points] Given integer array `A` satisfying the max heap property, index `i`, and value `v`, write pseudocode[1] for a function `change_value(A, i, v)` which replaces the value at `A[i]` with `v`, and then swaps at most $O(\log n)$ pairs of elements to ensure the array maintains the max heap property after the change. The only operations you are allowed to perform on the array are comparisons and swaps between pairs of the integers, i.e., `A[i] < A[j]` and `A[i], A[j] = A[j], A[i]`. You are **not allowed** to store array elements outside of the array.

Solution:

Compare the new value to the existing value at the given 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_value(A, i, v):
    if A[i] < v:
        A[i] = v
        max_heapify_up(A, len(A), i)
    if 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 2-3. [40 points] **Cow Party**

Bessie the cow wants to throw a party! She has F cow friends, and each cow friend C_i lives in a plane pasture, X_i meters east and Y_i meters north of Bessie's house. Bessie would like to invite all her friends to the party, but her house only has room to accommodate her and at most $G < F$ cow guests. Because Bessie is a pragmatic cow, she decides to invite the G cow friends that live closest to her so they don't need to travel far (you may assume that her distance to each cow friend is unique). Let's help Bessie compile her guest list!

[1] Psuedocode may be lines of English sentences, Python code, or a combination of the two.

(a) [5 points] Given an array of cow friend locations, describe an $O(F \log F)$ time algorithm to return Bessie's guest list.

Solution: Use any $O(F \log F)$ sorting algorithm to sort the list of points by their distance to the origin, and then tell Bessie the G smallest elements by reading from the left. The total time is $O(F \log F + G) = O(F \log F)$.

Rubric:

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

(b) [10 points] Bessie thinks your $O(F \log F)$ time algorithm is too slow. Describe an $O(F + G \log F)$ time algorithm to return Bessie's guest list. Your algorithm should also be **in-place**, i.e., use no more than $O(1)$ extra space beyond the input array.

Solution: Use `build_min_heap` (by modifying max heap code) to turn A into a min heap in-place. This takes $O(F)$ time and $O(1)$ extra space. Now, run `extract_min` G times and tell Bessie the last G elements of A in reverse order. Takes $O(G \log F)$ time and $O(1)$ extra space.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(c) [10 points] Bessie asks you to demonstrate your $O(F + G \log F)$ time algorithm for her. In doing so, she realizes that your algorithm involves reading **and writing** to the cow friend location list that she gave you. She is angry, and tells you that she does not want you writing on her list. She says that your algorithm may use $O(G)$ extra space, so long as you only read from her list. Describe a $O(F \log G)$ time algorithm to return Bessie's guest list using at most $O(G)$ extra space, provided only read access to the cow friend location array.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

Solution: The key insight here is that to find the minimum G elements of a list, it is not necessary to pick them off in order. Rather, as we scan through the list of F elements (this part is necessary, so at least $O(F)$ time is needed), we simply have to maintain a set of the smallest G elements seen so far. That's where the $O(G)$ space comes from.

It is clear that as we scan through the first G elements of the list, we can simply insert each one into the set. When we inspect the $(G+1)$st element, though, we have to compare it against the largest element in our set. If $A[G+1]$ is smaller, then we kick out the largest element and insert $A[G+1]$. This process repeats up through $A[F]$. Therefore, we need an efficient means of finding the max element, removing it, and inserting new element. This is exactly what max heaps are designed for!

It follows that by maintaining a **max** heap of the *smallest* G elements seen so far, as we scan through all F elements, we can find the minimum G elements $O(F \log G)$ time. Note, however, that this is actually asymptotically worse than our second solution. If G is not very small, then $F \log G > F$. More importantly, $F \log G$ is a factor of $\log G/G$ of FG, whereas $G \log F$ is a factor of $\log F/F$. The function $\log F/F$ is monotonically decreasing, so for $G < F$, $F \log G$ is larger than $G \log F$.

Once the scan is complete, the heap contents can be sorted (via multiple calls to `extract_max`) and the now-sorted array can be returned.

(d) [15 points] Write a Python function `close_cow_friends` that implements your $O(F \log G)$ time algorithm. You can download a code template containing some test cases and useful functions presented in recitation from the website. Submit your code online at `py.mit.edu/6.006`.

```
def close_cow_friends(locations, g):
    '''
    Return g locations closest to origin in increasing order.
    Input:  locations | generator of location tuples (x, y)
            g         | number locations to return
    '''
    ##################
    # YOUR CODE HERE #
    ##################
    return []
```

Solution:

```
def max_heapify_down(A, n, i):
    l = 2 * i + 1
    r = 2 * i + 2
    l = l if l < n else i
    r = r if r < n else i
    c = l if A[r] < A[l] else r
    if A[i] < A[c]:
        A[i], A[c] = A[c], A[i]
        max_heapify_down(A, n, c)

def max_heapify_up(A, n, i):
    p = (i - 1) // 2
    if 0 < i and A[p] < A[i]:
        A[i], A[p] = A[p], A[i]
        max_heapify_up(A, n, p)

def close_cow_friends(locations, g):
    A = [None] * g
    i = 0
    for l in locations:
        d = (l[0] * l[0]) + (l[1] * l[1])
        if i < g:
            A[i] = (d, l)
            i = i + 1
            max_heapify_up(A, i, i - 1)
        elif d < A[0][0]:
            A[0] = (d, l)
            max_heapify_down(A, g, 0)
    for i in range(g - 1, -1, -1):
        A[0], A[i] = A[i], A[0]
        max_heapify_down(A, i, 0)
    return [a[1] for a in A]
```

Rubric:

- This part automatically graded at `py.mit.edu/6.006`.