Problem Set 2

All parts are due March 3rd, 2016 at 11:59PM.

Name: Your Name

Collaborators: Name1, Name2

Part A

Problem 2-1.

(a)

(b)

(c)

(d)

(e)

Problem 2-2.

(a) Explanation: In order to compute the augmented merge, we basically have to do the normal merge operation we studied in class (we compare the first elements of each list, add the smallest to the result list and keep doing comparing the first not yet added until we reach the end of both lists). The difference is that now we add a counter to know the number of inversions. For this counter to work we use a support variable initialized to the length of the of the first list. Then, we decrease the variable by one unit every time the element from the first list is appended to the list (no inversions are made). Then, we increase the variable every time something from the second list is appended (as an inversion is made) and add the value of the variable to the counter. When we go through the whole list, we just see if the counter is even or odd and then return the final merged list B and the parity bit b.

Pseudocode:

```
Aumented Merge(A_1, A_2):

   k = |A_1|
   count = 0, i = 0, j = 0
   while i < |A_1| or j < |A_2|:
      if A_1[i] < A_2[j]:
         B.append(A_1[I])
         i++
         k--
      else:
         B.append(A_2[j])
         j++
         count += k
   b = count%2
   return B, b
```

(b) The running time is $O(max(n_1, n_2))$, due to the while loop, that happens the max between n_1 and n_2 times (as we increment i and j by 1 everytime we go through the loop till we reach the end of both lists).

(c) Explanation: This is basically the same as the MergeSort algorithm that we studied in class. The only difference is that now we have to keep track of the parity of each part of the recursive problem and take the mod of the total parity (that we find in the end).

Pseudocode:

```
Aumented MergeSort(A):

left = A[1, ..., len(A)/2]
right = A[len(A)/2 +1, ..., len(A)]

#CHECK BASE CASE ON THE BOOK
if len(left) = 1 or len(right) = 1:
           return B, b

(sorted_left, parity_left) = Augmented MergeSort(left)
(sorted_right, parity_right) = Augmented MergeSort(right)

(B, parity_all) = Augmented Merge(left, right)

b = (parity_left + parity_right + parity_all) %2

return B, b
```

(d) To analyse the running time of this algorithm we use the recursion analysis techniques we learned in class. Everytime we divide the problem by 2 and each problem has half of the original problem size. What we do outside of the recursion is the augmented merge sort, which we analysed to be $O(max(n_1, n_2))$. In this case n_1 and n_2 are $(\frac{n}{2})$, so we have a recursion function:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

By Master Theorem we know this is $O(nlogn)$

Problem 2-3.

(a)

(b)

(c)

(d)

Part B

Problem 2-4.

Submit your implemented python script.

(a)

(b)

(c)

(d)

(e)