CS 374 Spring 2018
Homework 5

Nathaniel Murphy (njmurph3@illinois.edu)
Tanvi Modi (tmodi3@illinois.edu)
Marianne Huange (mhuang46@illinois.edu)

```
1  def merge_count_double(A, B):
2      count = 0
3      ret = []
4      temp_B = [2*i for i in B]       ## Double every value in array B
5
6      while len(A) > 0 and len(temp_B) > 0:
7          if A[0] <= temp_B[0]:
8              A.pop(0)
9          else:
10             temp_B.pop(0)
11             count += len(A)
12     return count
13
14 def mergesort_count_double(A, lo, hi):
15     if lo - hi >= 0:
16         return 0
17     mid = (lo+hi) // 2              ## Find middle element
18     a = mergesort_count_double(A, lo, mid)
19     b = mergesort_count_double(A, mid+1, hi)
20     return a + b + merge_count_double(A[lo:mid+1], A[mid+1:hi+1])
```

The language that I test this algorithm and that is very intuitive (almost looks like pseudocode) is Python. Note that we use variables with capital letters (i.e. A, B) to denote arrays. This algorithm looks almost identical to the algorithm discussed in Lab 10 except we multiply array B in the `merge_count_double` function by 2 which will ensure that $a_i > 2a_j$ for $i < j$. The `.pop(0)` notation used in `merge_count_double` represents removing and returning the first element of the array. Assuming that `.pop(0)` takes $\mathcal{O}(1)$ time and instantiation of `temp_B` takes $\mathcal{O}(\frac{n}{2})$ time which means that `merge_count_double` runs in approximately $\mathcal{O}(\frac{3n}{2}) = \mathcal{O}(n)$ time which is identical to the regular mergesort algorithm. Thus, `mergesort_count_double` runs in $\mathcal{O}(n \log n)$