Mergesort

1 Algorithm

- Input: array of n numbers, unsorted.
- Assume: all elements distinct

Approach:

```
def MergeSort(A, p, r):
    if p < r:
        q = (p+r)/2
        MergeSort(A, p, q)
        MergeSort(A, q+1, r)
        merge(A, p, q, r)

def Merge(A, p, q, r)
    C = [0] * (r - p + 1)
    i, j, k = p, q, 1

    while i < q and j <= r:
        if A[i] < A[j]:
            C[k++] = A[i++]
        else
            C[k++] = A[j++]
    Append leftovers and copy C -> A
```

2 Analysis

For the merge subroutine, the best-case number of comparisons for this version is n, and the worst case is $2n-1$ when given two lists of size n to merge. There can be some practical improvements made, such as using a binary search when one side to merge has size 1, but the worst case for **any** merge algorithm will always be $2n-1$ because there will always exist at least 1 case such that the algorithm must iterate through all elements to determine the ordering.

At each level of the algorithm, the merge subroutine merges two lists of size $n/2$ each, so the work done at each level of the recurrence is $n-1$. Because there are $\lg n$ levels in the recursive, tree, the number of comparisons is $\Theta(n \lg n)$.

$$\sum_{i=0}^{\lg n} 2^i \left(\frac{n}{2^i} - 1\right) = \sum_{i=0}^{\lg n - 1} (n - 2^i) \tag{1}$$

$$= n \lg n - n + 1 \tag{2}$$

3 Practical Improvements

- Use insertion sort for small subarrays, at a cutoff of ≈ 7. Eliminates unecessary overhead, improves $\approx 20\%$.
- Stop if already sorted: is the biggest item in first half $\leq$ smallest item in second half?

4 Bottom-up Mergesort

Basic plan:

1. Pass through array, merging subarrays of size 1
2. Repeat for size 2, 4, 8, 16, etc...