CS 5720 Design and Analysis of Algorithms

Project #2

Submission requirements:

- A .zip file containing your source code. You may use any language you would like.
- A PDF (submitted separately to the Canvas assignment) containing each item below that is listed as a **Deliverable.** For each item contained in your PDF, clearly mark which deliverable it is associated with. Plots should be clearly labeled and have descriptive captions.

Hybrid Sorting Methods. In class, we've seen that divide-and-conquer methods can be used to create blazingly fast sorting algorithms such as Mergesort and Quicksort with average-case (sometimes worst-case) time complexity $\Theta(n \log n)$. What we haven't discussed as much is the fact that *for small arrays,* the elementary sorting algorithms (insertion sort, bubble sort, selection sort) aren't really all that slow, even though they have worst-case time complexity of $\Theta(n^2)$. In fact, it can happen that the best elementary sorting algorithms are actually faster than Mergesort on very small arrays. What can we do with this fact? We can use Mergesort to handle large arrays, and then when the arrays in Mergesort's recursion get small enough, pass them to an elementary sorting algorithm like Insertion sort to finish up. This is known as *hybrid sorting* since we're combining two types of sorting algorithm. If the implementations are good, the combination of the two can be faster in general than either algorithm individually. This project will explore this phenomenon.

Your task: Create a new sorting algorithm `HybridSort(A[n],K)` that takes two inputs:

- `A[n]`: A numerical array of length n, and
- `K`: A length threshold; note that you must have $K \geq 1$. If $n \leq K$, your algorithm should sort it using Insertion Sort. If $n > K$, your algorithm should sort it using Mergesort.

Since it is a sorting algorithm your algorithm should return or compute a new array that contains exactly the same elements as the input array and is sorted ascending. Your algorithm may operate in-place or return a new sorted array; the choice is yours.

The high-level operation of your algorithm should work as follows, given correctly-implemented subroutines `InsertionSort(A[n])` and `Merge(B[p],C[q])`:

Algorithm 1

1: **function** HybridSort($A[0..n-1], K$)
2: **if** n < K **then return** InsertionSort($A[0 .. n-1]$)
3: **else return** Merge(HybridSort($A[0 .. \lfloor n/2 \rfloor - 1], K$), HybridSort($A[\lfloor n/2 \rfloor .. n-1], K$))
4: **end if**
5: **end function**

Assignment:

1. Implement the above algorithm in a language of your choice, also providing your own implementations of `InsertionSort` and `Merge`. Test your implementation thoroughly for correctness (e.g., use your programming language's built-in sorting algorithm and make sure it agrees with your algorithm on every input). Provide verification that you have tested your algorithm and that it always sorts correctly.

   Deliverable 1: Your implementation's code and your verification of correctness.

2. Generate a plot (or plots) which depicts your algorithm's average running time as a function of K and n on input arrays that you generate randomly (i.e., input arrays for this deliverable should *not* be pre-sorted in any way). Your "mental model" here should be that K will be relatively small (under 100) and n will be relatively large (as large as your implementation can handle in a reasonable amount of time).

   As you are running the tests, select values of K and n which showcase interesting phenomena. For example, you might expect very low and very high values of K to have poor performance, but moderate values to have good performance. If that intuition holds, make sure you plot your running times in such a way that this phenomenon is clearly depicted in your plot.

   Finally, a note on "average running time": for a specific value of K and n, a single run of your algorithm is not enough to tell you anything about its average running time. To compute an average for a specific value of n, you'll need to generate several random arrays of length n, feed them all to the algorithm for your fixed value of K, then compute the average of the running times you obtained.

   Deliverable 2: A plot showing the average run time of your algorithm as a function of K, with a separate trace for at least 5 representative values of n.

3. Identify the optimal value of K as a function of array length n. This should be informed by (or answered by) your analysis in the previous step. How much does the best choice of K depend on n? Note: depending on your implementation, it could happen that this optimal K will be the same for all n. If that's the case, report it anyway. Try to understand any relationship (or lack of relationship) that you find.

   Deliverable 3: A plot showing the optimal value of K as a function of array length n. Explain why you think the relationship between n and optimal K is the way that it is.

4. **Deliverable 4:** Repeat deliverables 2 and 3; however, this time, test your algorithm only on *sorted* arrays. How do the results differ from what you reported in Deliverables 2 and 3? Explain these differences to the best of your ability.

Further Reading: The default sorting algorithm used in Python is called "Timsort" after its creator Tim Peters, who created it for Python. A version of Timsort is still used as Python's native sorting algorithm to this day. It is hybrid, and also highly optimized (it does advanced things like looking for already-sorted subarrays so it can adaptively skip things it doesn't need to do). Read more here: `https://en.wikipedia.org/wiki/Timsort`