CPSC 331 — Assignment #3

Improving the Performance of `Merge Sort`

About This Assignment

While completing this assignment you will implement an algorithm that is based on `Merge Sort`. If you implement this with care then it will be somewhat faster than the version of the algorithm described in class. It can be completed by groups of up to three students and is due by 11:59pm on Thursday, December 5.

More About Sorting Algorithms

Recall that `Merge Sort` is a ***divide and conquer*** algorithm for sorting: If its input is not an array with length one[1] then an array is sorted by ***splitting*** its input to form two smaller arrays, sorting these recursively, and then combining the resulting arrays by a ***merge*** process to produce a sorted array that is returned as output corresponding to the given input array.

While this "asymptotically faster" than classical sorting algorithms like `Insertion Sort`, it actually requires *more* steps to sort the input array when the length of this is very small!

When this happens (and something like this often happens, when "divide and conquer" algorithms are being considered) a ***hybrid*** algorithm is considered. This combines the use of the asymptotically fast algorithm with the algorithm that is asymptotically slower but simpler, and faster (in the worst case) when executed on *very small* inputs.

There are (at least) two ways to produce a hybrid algorithm. Both involve a positive integer ***threshold***:

- The input size (in this case, the length of the input array) is compared to the threshold. If the size is less than or equal to the threshold then the asymptotically slower algorithm

[1] . . . or length zero, if empty input arrays are allowed. . .

(that is faster on very small inputs) is used to solve this instance of the problem; the asymptotically faster algorithm is used otherwise.

This the input size is only compared to the threshold once, and one of the two algorithms being considered is used, without change, after that.

- The ***hybrid*** algorithm is, itself, recursive. It begins by comparing the input size (again, the length of the input array, for this case) to the threshold. If the input size is less than or equal to the threshold then the asymptotically slower algorithm (that is faster on very small inputs) is used.

  On the other hand, if the input size is greater than the threshold then the hybrid algorithm behaves like the asymptotically faster one — right up until the time that the algorithm would call itself recursively: The "hybrid" algorithm now calls itself recursively, instead of calling the asymptotically faster algorithm.

  The effect is that the input size is being compared to the threshold every time the hybrid algorithm is called — not just once — until the (decreasing) input size becomes less than or equal to the threshold, and the asymptotically slower algorithm is used.

Even though it might seem otherwise — because of all the extra comparisons of input sizes to the "threshold" that are included — the *second* way to do this results in an algorithm that is somewhat ***faster*** than both of the algorithms that one starts with — provide that the value for the "threshold" is chosen carefully.

Pseudocode for a `sort` algorithm that corresponds to the second version of a "hybrid algorithm" is shown in Figure 1 on page 3. The ***threshold*** is given in this code by the value of a variable called `THRESHOLD`. Four other methods are called by this one:

- A method called `copy` receives an ArrayList `A` with base type `T` (where `T` is a type with a total order), whose entries are assumed not to be `null`. This method returns, as output, another ArrayList `B` with the same base type, size, and contents — so that $\texttt{A.get(i)} = \texttt{B.get(i)}$ for every integer `i` such that $0 \leq \texttt{i} < \texttt{A.size()} = \texttt{B.size()}$.

- A method called `insertionSort`, which does not return output, receives an ArrayList with base type `T`, such that `T` has a total order, as its input. It is assumed that this ArrayList has positive size and that none of its entries are `null`. The method sorts the elements of its input array in nondecreasing order, by an application of the `Insertion Sort` algorithm.

- A method called `split`, which also does not produce output, receives three ArrayLists — `A`, $\texttt{B}_1$ and $\texttt{B}_2$ — that all have the same base type `T` as input. As above, it is assumed that `T` has a total order, `A` has positive size and that none of the entries of `A` is `null`. On the other hand, it is also assumed that both $\texttt{B}_1$ and $\texttt{B}_2$ are empty, that is, they initially have size zero.

```
ArrayList<T> sort (ArrayList<T> A) {
 1. if (A.size() ≤ THRESHOLD) {
 2.   ArrayList<T> B = copy(A)
 3.   insertionSort(B)
 4.   return B
    } else {
 5.   ArrayList<T> B_1 = new ArrayList<T>()
 6.   ArrayList<T> B_2 = new ArrayList<T>()
 7.   split(A, B_1, B_2)
 8.   ArrayList<T> C_1 = sort(B_1)
 9.   ArrayList<T> C_2 = sort(B_2)
10.   return merge(C_1, C_2)
    }
}
```

Figure 1: Second Version of a Hybrid Sorting Algorithm

On termination of the application of this method, the array A has not been changed. If A has size n (so that n is a positive integer) then, on termination, $\mathtt{B}_1$ and $\mathtt{B}_2$ have sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ respectively (so that the sum of their sizes is equal to n). For $0 \leq i \leq \lceil n/2 \rceil - 1$, $\mathtt{B}_1.\mathtt{get}(i) = \mathtt{A}.\mathtt{get}(i)$ and, for $\lceil n/2 \rceil \leq i \leq n-1$, $\mathtt{B}_2.\mathtt{get}(i - \lceil n/2 \rceil) = \mathtt{A}.\mathtt{get}(i)$. Thus the final versions of $\mathtt{B}_1$ and $\mathtt{B}_2$ have been obtained by "spitting" A up into two ArrayLists of approximately the same size, as the name of the method suggests.

- A method called `merge` solves the "Merging" problem that was discussed during the lecture on the Merge Sort algorithm — with ArrayList's replacing arrays as inputs and outputs. It is assumed that none of the entries of either input array are `null`.

1. Consider a modified version of the "Sorting" problem in which the input and output are an ArrayList instead of an array and the precondition asserts that `THRESHOLD` is a positive integer, and that each entry in the input ArrayList is not `null`. Sketch a proof that, if the methods "`copy`", "`insertionSort`", "`split`" and "`merge`" are as described above, then the `sort` algorithm shown in Figure 1 correctly solves this version of the "Sorting" problem.

   Note: This should not be hard! Consider a modification of the proof of correctness of the "Merge Sort" algorithm.

2. Complete the program `BetterMergeSort.java` (that is now available) to implement the hybrid algorithm that has now been described.

In order to bound the worst-case running time of this algorithm, one can form a recurrence for the number of steps executed in the worst case.

As noted in class the `insertionSort` algorithm uses a quadratic number of steps in the worst case — so that there exist real constants a_I, b_I and c_I such that if the input array has positive length n then the number of steps used by this algorithm, when executed on this array, is at most $a_I n^2 + b_I n + c_I$. Indeed, as described in the notes for Lecture #15, one can set $a_I = \frac{5}{2}$, $b_I = \frac{5}{2}$, and $c_I = -3$.

It should not be hard to give pseudocode for an implementation of the `copy` method described above, establish its correctness using the methods introduced at the beginning of this course, and prove that it uses a number of steps linear in the length of the input array in the worst case — so that there exist real constants b_C and c_C such that if the input array has positive length n then the number of steps used by this algorithm, when executed on this array, is at most $b_C n + c_C$.

Similarly, information included in Lecture #16 can be used to design and prove the correctness of a `split` method that also uses a number of steps linear of the length of the input array in the worst case — so that there exist real constants b_S and c_S such that if the input array has positive length n then the number of steps used this algorithm when executed on this array, is at most $b_S n + c_S$.

As noted in Lecture #16, there is a `merge` algorithm such that the number of steps executed is at most linear in the sum of the lengths of the input arrays. In particular, there exist real constants b_M and c_M such that if the input arrays have positive lengths n_1 and n_2 respectively, then the number of steps used by this algorithm when executed on these arrays is at most $b_M(n_1 + n_2) + c_M$. In particular, one can set $b_M = 5$ and $c_M = 10$.

Once again, consider the algorithm shown in Figure 1. Let $T_{\text{THRESHOLD}}(n)$ be the maximum number of steps executed by this algorithm when it is executed on an input array with positive length n. Then it can be shown that there exist real constants α, β, γ, δ and ζ such that $\alpha > 0$, $\beta > 0$, $\gamma > 0$, and

$$T_{\text{THRESHOLD}}(n) \leq \begin{cases} \alpha n^2 + \beta n + \delta & \text{if } n \leq \text{THRESHOLD}, \\ T_{\text{THRESHOLD}}(\lceil n/2 \rceil) + T_{\text{THRESHOLD}}(\lfloor n/2 \rfloor) + \gamma n + \zeta & \text{if } n > \text{THRESHOLD}. \end{cases} \tag{1}$$

Consider a function $T : \mathbb{N}^2 \to \mathbb{R}$ such that, for positive integers n and k,

$$T(n,k) = \begin{cases} \alpha n^2 + \beta n + \delta & \text{if } n \leq k, \\ T(\lceil n/2 \rceil, k) + T(\lfloor n/2 \rfloor, k) + \gamma n + \zeta & \text{if } n > k. \end{cases} \tag{2}$$

The next claim might seem so “obvious” that you might wonder what you have to prove it.

However, it is not hard to prove and describing a proof of it might help you to get ready for the problems that follow it.

3. Prove that if THRESHOLD is a positive integer then $T_{\text{THRESHOLD}}(n) \leq T(n, \text{THRESHOLD})$ for every positive integer n.

4. Prove that if k is a positive integer, the function $T : \mathbb{N}^2 \to \mathbb{R}$ is as shown at line (2), above, $\delta + \gamma + \zeta > 0$,

   $$A = \max\left(\alpha + \beta + \delta + \zeta + \gamma, k \cdot \alpha + \beta + \tfrac{1}{k} \cdot (\delta + \zeta + \gamma)\right)$$

   and $B = -(\zeta + \gamma)$, then

   $$T(n, k) \leq \gamma n \log_2 n + An + B$$

   for every positive integer n.

   Note: When proving this, you may use the following without proof:

   (a) Let $\Delta : \mathbb{R} \to \mathbb{R}$ such that
   $$\Delta(x) = \alpha x + \beta + \tfrac{\delta+\zeta+\gamma}{x}$$
   for every positive real number x. Then — if $\delta + \gamma + \zeta > 0$, as given above — then
   $$\Delta(x) \leq \max(\Delta(1), \Delta(k)) = A$$
   for every real number x such that $1 \leq x \leq k$.

   (b) If m is an odd integer such that $m \geq 3$ then $\log_2 \left\lceil \frac{m}{2} \right\rceil \leq \log_2 \left(\frac{m}{2}\right) + \frac{\log_2 e}{m}$.

   (c) $\frac{2 \log_2 e}{3} \leq 1$.

   Proofs of these will be included in the solutions for this assignment, for interested students.

A similar argument — making use of another well-known approximation, namely that

$$\tfrac{x}{1+x} \leq \ln(1 + x)$$

when x is a real number such that $|x| < 1$ — can be used to establish that if

$$\widehat{A} = \min_{\substack{n \in \mathbb{N} \\ 1 \leq n \leq k}} \left(\alpha n - \gamma \log_2 n + B - \tfrac{1}{n}(\delta - \gamma + \zeta)\right)$$

and $\widehat{B} = \gamma - \zeta$, then

$$T(n, k) \geq \gamma n \log_2 n + \widehat{A} n + \widehat{B}$$

for every positive integer n, as well.

For the rest of this assignment, you should assume that the function $T : \mathbb{N}^2 \to \mathbb{N}$ really is the number of steps used by this hybrid algorithm (when the first argument is the input and the second argument is the choice of threshold) in the worst case.[2]

5. The above information can be used to argue that *one* of the following parts of the algorithm should be optimized in order to significantly improve the performance of this algorithm —

   - the implementation of "Insertion Sort", or
   - the implementation of "Merge Sort", including the algorithms to split the input array into two smaller arrays and to merge two sorted array to produce a larger sorted array

   while the *other* part is not as important. Which of these two is more important? Why?

[2] With a bit of extra work it can be shown that, for choices of α, β, δ, γ and ζ corresponding to your implementation of the algorithm, this is true.