Project 3: Constant Factor vs Efficiency Class

by Dion Wayne Pieterse & Michael Lindwall

Mathematical Analysis – List Merge Sort

Sorting Problem:
Input: A linked list V of n comparable elements.
Output: A linked list S containing the elements of V in non-decreasing order.

Pseudo Code:

```
def merge_sort_list(V):

    if |V| <= 1:                                  (1)
        return V                                  (1)
    else:                                         (1)
        HALF_LENGTH = V.size() / 2                                (1)
        iter::begin_it = V.begin()                                (1)
        iter::half_way_it = V.begin()                             (1)
        advance(half_way_it, HALF_LENGTH)                         (n)
        iter::end_it = V.end()                                    (1)

        //create left and right sublists
        left_half = LinkedList()                                  (1)
        right_half = LinkedList()                                 (1)

        iter::begin_it_l = left_half.begin()                      (1)
        iter::begin_it_r = right_half.begin()                     (1)

        //populate left sublist
        left_half.insert(begin_it_l, begin_it, half_way_it)       (n)
        //populate right sublist
        right_half.insert(begin_it_r, half_way_it, end_it)        (n)

        return merge_l(merge_sort_list(left_half), merge_sort_list(right_half))    (n + 2T(n/2))
```

Note: Due to Master's Theorem, the else section always executes.
Efficiency = 1 + 1 + 1 + n + 1 + 1 + 1 + 1 + 1 + n + n + n + 2T(n/2)
= 8 + 4n + 2T(n/2)
= **2T(n/2) + 4n + 8**

r = 2, d = 2,
c(n) = 4n + 8 E O(n^1); therefore, k = 1
Two base cases: |V| = 0 and |V| = 1; therefore, t >= 1 (there exists a base case)

Therefore, r, d, t and k E O(1) and r >= 1, t >= 1, d > 1, k >= 0 so Master Theorem may be applied.
$r = d^k$
$2 = 2^1$; therefore, case 2 is satisfied from Master's Theorem
Efficiency = $O(n^k \log n) = O(n^1 \log n) =$ **O(nlogn)**

Helper Function:

Merge Problem:
Input: Two linked lists L and R in non-decreasing order, of length n_L and n_R respectively.
Output: A linked list S in non-decreasing order containing all the elements of both L and R.

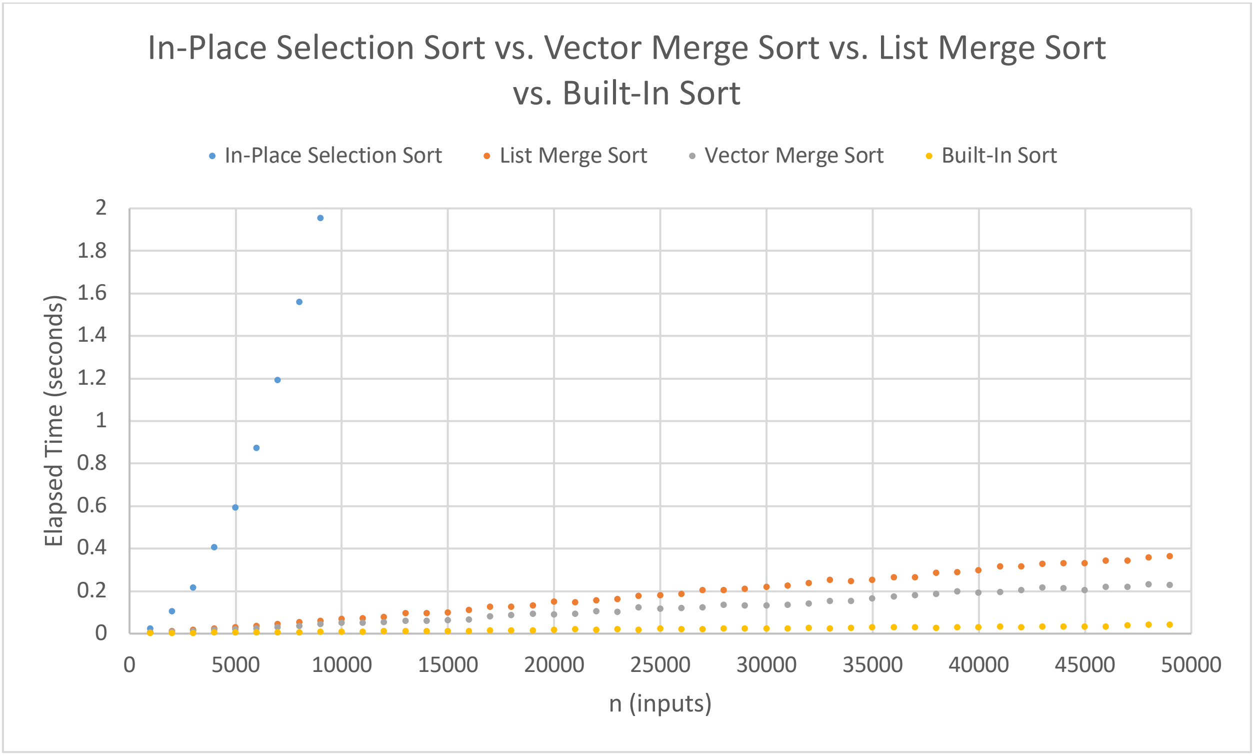
```
def merge_l(L, R):
        S = LinkedList()        (1)
        li = 0                  (1)
        ri = 0                  (1)

        iter::li_it = L.begin() (1)
        iter::ri_it = R.begin() (1)

        while li < L.size() && ri < R.size():      (1)     n iterations

                if(li_it <= ri_it):                (1)
                        S.add_back(li_it)          (1)
                        advance(li_it, 1)          (1)
                        li += 1                    (1)   max(4, 5)
                else:                              (1)
                        S.add_back(ri_it)          (1)
                        advance(ri_it, 1)          (1)
                        ri += 1                    (1)

        for i in range(i = li, L.size()):          (1)
                S.add_back(li_it)                  (1)     n iterations
                advance(li_it, 1)                  (1)

        for i in range(i = ri, R.size()):          (1)
                S.add_back(ri_it)                  (1)     n iterations
                advance(ri_it, 1)                  (1)

        return S                                   (1)
```

Efficiency = 1 + 1 + 1 + 1 + 1 + n(1 + 1 + 1 + 1 + 1 + 1) + n(3) + n(3) + 1
= 5 + n(6) + 6n + 1

= 12n + 6

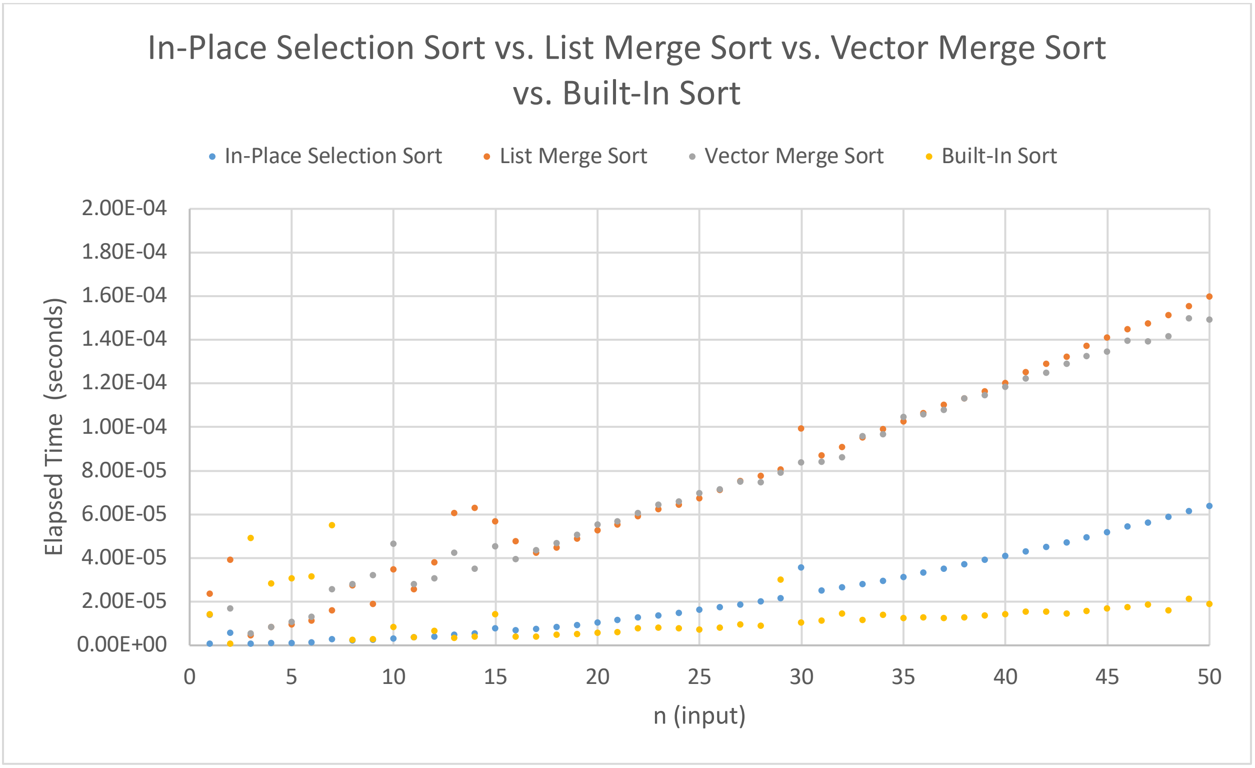
Proof of helper function:
12n + 6 E O(12n + 6) by Trivial Property Theorem
= O(12n) by Dominated Terms Theorem
= **O(n)** by Constant Factor Theorem

Plot #1: All Four Algorithms n < 50 000:



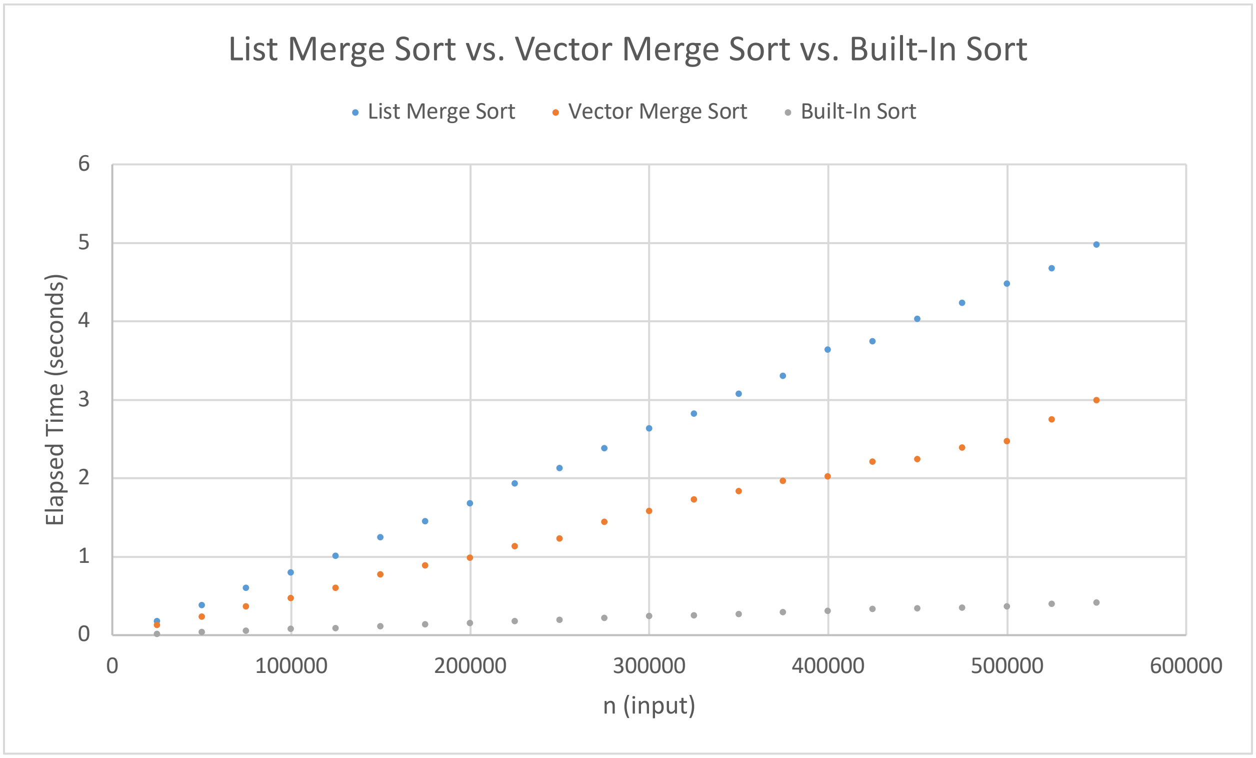
Conclusion: The slowest is clearly the in-place selection sort for large values of n. The fastest is built-in sort, the second fastest is vector merge sort and the third fastest is list merge sort. This plot is in clear support of hypothesis 1: Clearly, the list merge sort is the slower of the three O(n log n) algorithms. Also hypothesis 2 is supported: O(n log n) algorithms outperform O(n^2) algorithms for larger n values.

Extra Plot #2: Scatter Plot For All Four Algorithms For n <= 50



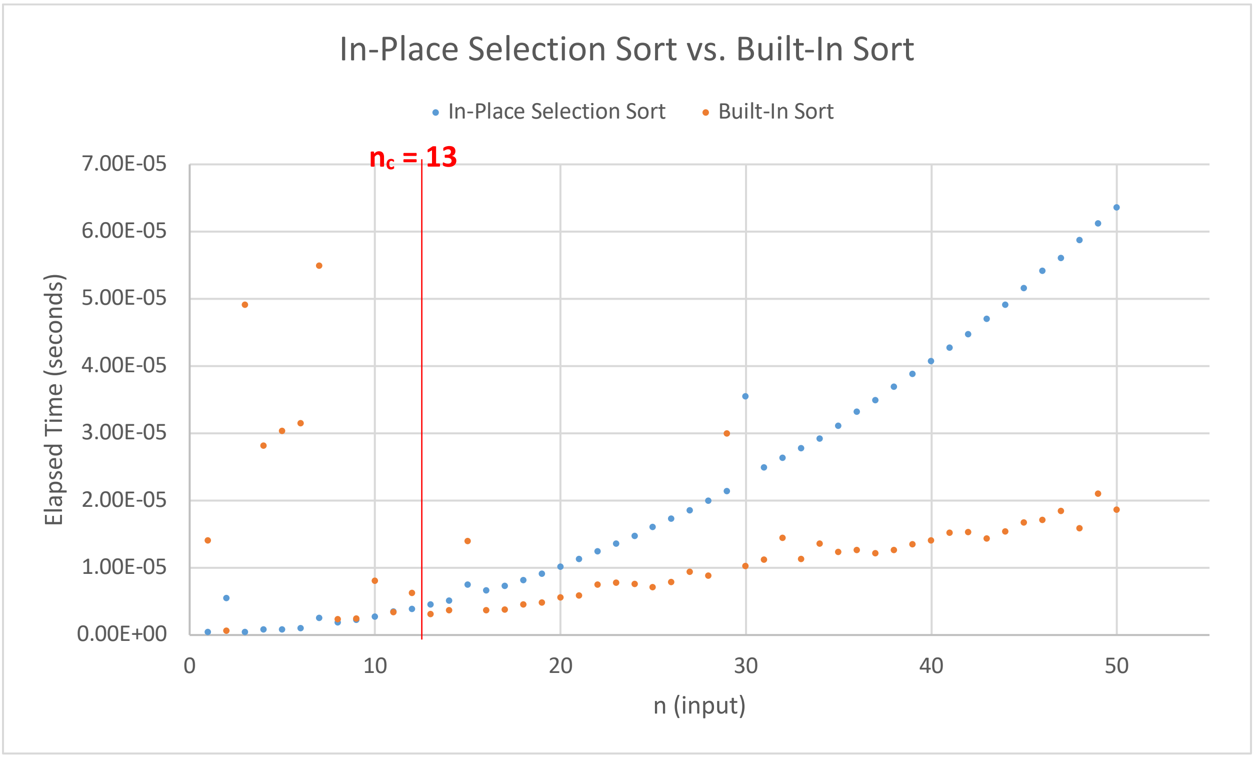
Conclusion: The above graph is to provide a general overview of performance for small values of n <= 50. It is clear that In-Place Selection Sort is fastest for very small values of n. The crossover point between built-in sort and in-place selection sort appears around n = 13, after which built-in sort becomes faster than in-place selection sort. Please refer to plot #4 for further details for the crossover point.

Plot #3: Three O(n logn) Algorithms n <= 550 000:



Conclusion: The above chart shows the performance for all three O(n logn) algorithms. List merge sort is the slowest algorithm. Vector merge sort is the second fastest and the built-in sort is the fastest of the three algorithms. Hypothesis 1 is clearly supported by this scatter plot, list merge sort is the slowest of the three O(n log n) algorithms.

Plot #4: Built-In Sort vs. In-Place Selection Sort Algorithms:



Note: The vertical red line indicates where the crossover (**n_c**) occurs.

Conclusion: The above plot clearly shows that as n input values grows, the built-in sort algorithm becomes much faster than the in-place selection sort algorithm after a certain n value. The red line indicates the crossover where built-in sort becomes faster than in-place selection sort as n input values grows. The crossover point is at **n = 13**.

Analysis and Answers to Questions:

- **Based on your experiments, is it true that selection sort is the fastest algorithm for very small? Is it true that selection sort is the slowest algorithm for large n? What *n* is the crossover point *c_n* for your implementations of algorithms 1 and 2?**

  Yes, selection sort is faster for small values of n up until n_c = 13 in our experiment's case. Yes, selection sort is the slowest algorithm as n grows much larger in value.

  The value of n that signifies the crossover point is **n_c = 13** for algorithms 1 and 2.

- **Based on your experiments, is it true that the built-in sort and two merge sorts all have *O*(*n* log *n*) time, built-in has the best constant factor, vector merge sort has the second-best constant factor, and list merge sort has the worst constant factor?**

  Yes, the built-in sort, vector merge sort and list merge sort all have O(nlog n) time. We found the built-in sort has the best constant factor by far, vector merge sort has the second-best constant factor and the list merge sort has the worst constant factor. Looking at scatter plot #3, you can clearly see how much slower list merge sort is when compared to vector merge sort and built-in sort. Built-in sort is the clear winner.

- **Is the slowest of your *O*(*n* log *n*) sorts faster than selection sort? If so, by what margin? Do you find this surprising, given that selection sort's code is so much simpler?**
  Yes, the slowest O(n logn) sort (list merge sort) is indeed faster than in-place selection sort. Looking at scatter plot #1, between n = 1000 and n = 10000, you can see the rapid rise in time for selection sort. List merge sort is faster by a large margin. Yes, we find it surprising because list merge sort utilizes a recursive approach which would seem slower as well as having to generate a sorted linked list in the helper function, but list merge sort performs faster regardless.

- **Does your data support, or repudiate, hypothesis 1, and why?**

  Hypothesis 1:
  "Sorting algorithms that operate in-place on vectors are faster, by constant factors, than sorting algorithms that operate on linked-lists and/or out-of-place."

  Yes, the data does support the hypothesis that sorting algorithms that operate in-place on vectors are indeed faster by constant factors than sorting algorithms that operate on linked-lists and/or out-of-place. This finding is clearly visible in scatter plot #1.

- **Does your data support, or repudiate, hypothesis 2, and why?**

  Hypothesis 2:
  "O(n logn)-time algorithms outperform O(n^2)-time algorithms for large n, regardless of the implementation choices that are made."

  Yes, our data supports hypothesis 2. Even the less efficient implementation (using lists) O(n logn)-time algorithm outperforms the O(n^2) in-place selection sort, which was implemented using a vector. This is clearly visible in scatter plot #1.

- **Based on these results, which strategy for speeding a bottleneck is better:**
  i. **picking an algorithm with good constant factors, but a mediocre efficiency class, and implementing it carefully using optimizations such as in-placedness (e.g. selection sort); or**
  ii. **picking an algorithm with a good efficiency class, but mediocre constant factors, and implementing simply (e.g. vector merge sort)**

  Why? What are the implications on software development in general?

  Based on our finding, we choose option (ii) as the best choice. When analyzing efficiency classes of algorithms, the constant factor is always dominated by the main efficiency class of the algorithm. Eg. O(5n) becomes O(n) by Constant Factor Theorem. Therefore, the constant factor is less important than the efficiency class. Picking a good efficiency class is the main factor for good performance. The implications on software development are that one's prudent choice of efficiency class can dramatically affect the performance of an algorithm.