Empirical Analysis of Sorting Algorithms

**Motivation:**

For many years it was believed that the fastest algorithm to bring comparable elements of an array into sorted order had asymptotic growth proportional to n2, where n is the size of the data to be sorted. However, sorting algorithms have been invented that have asymptotic growth proportional to n\*lg(n), such as Merge sort and Quick sort. This doesn’t mean that we should never use n2 sorting algorithms. The simplicity of implementation of n2 algorithms is legitimate reason to use them in some cases. Also, on sufficiently small data sizes, n2 algorithms are faster than n\*lg(n) algorithms. We will see an example of how large the data must be for n\*lg(n) algorithms to be faster than n2 algorithms in this experiment. Other than size, the state of the data determines which algorithm is optimal. Each algorithm has a best case and a worst case input state, but the average case is the same for all algorithms that solve the same problem. Some algorithms will react to extreme cases more than others. For sorting algorithms, the average case is randomly generated data. In this experiment, we will guess that the best case is data that is already sorted (nondecreasing) and the worst case is data that is sorted in the reverse order (nonincreasing). However, we will see that for some sorting algorithms this guess is not correct.

We will empirically compare the running time of Timsort, Merge sort, Insertion sort, and Bubble sort. We will count CPU cycles between the beginning and end of each algorithm execution for many different values of n and compare the data on a graph. The graph will reveal how the algorithms perform compared to each other and how the algorithms react to special cases. CPU cycles are a close approximation to the comparisons and assignments in the algorithms, since each comparison and assignment takes a number of CPU cycles. We won’t be able to tell how many CPU cycles each operation takes because this small number is offset by the number of CPU cycles it takes to measure CPU cycles, but we will get to see how many CPU cycles each algorithm takes. CPU cycles are more relevant than the actual number of comparisons and assignments since the whole purpose of algorithms is to run computers. In a world where data sizes are growing exponentially using the faster algorithm is critical.

**Background:**

All algorithms tested in this experiment use the fact that the elements are comparable to sort them.

Merge sort is a sorting algorithm that merges sorted subarrays together until the entire array is sorted. Merge sort starts with subarrays of size 1, which are already sorted. It merges pairs of sorted subarrays into larger sorted subarrays until it merges two sorted subarrays into the entire array.

Insertion sort inserts each element into a sorted array, starting with the left and moving right. Insertion sort uses a linear search from the right to find the correct position at which to insert an element, moving each element one space to the right in the array to make a space for the element being inserted.

Timsort is a hybrid of Merge sort and Insertion sort. Timsort uses an optimized combination of Merge sort and Insertion sort to take advantage of Merge sort’s low asymptotic complexity and Insertion sort’s low complexity on small problem sizes. Timsort is recursive. When called on a sufficiently small problem size, Timsort uses Insertion sort. When called on a problem size above a certain threshold, Timsort begins by applying Insertion sort on sufficiently small subarrays, then merges sorted subarrays together until the entire array is sorted. In this experiment Timsort’s threshold is set at 25.

Bubble sort traverses the array n times, swapping each pair of adjacent elements if they are in reverse order. Each traversal adds an element to the sorted end of the array. After n-1 traversals the entire array is known to be sorted. Some versions of Bubble sort terminate if they make a traversal with no swaps, but the version tested in this experiment does not.

**Procedure:**

A program was written in the language C++. The program empirically compared Insertion sort, Bubble sort, Merge sort, and Timsort. The program tested each algorithm on data sizes that are the closest whole numbers to the powers of the square root of two (sqrt(2)). The intent of using powers of sqrt(2) was to see if powers of 2 affect Merge sort and Timsort. Every other data size was a power of two, so if Merge sort or Timsort was significantly affected by powers of 2 it would show up in the graph. The maximum data size the program is set to test is 10,000. The highest number tested was 8192, which is the highest power of sqrt(2) less than or equal to 10,000. To make the tests fairer, the exact same data was used on all four algorithms.

To ensure that the algorithms weren’t cheating, the program checked to see if the data was sorted after each execution of a sorting algorithm. The program would display an error message and terminate if an algorithm failed to sort the data. The efficiency of each algorithm was compared via the TSC CPU cycle counter.

The following pseudocode describes the procedure of the code.

**Pseudocode:**

enumeration dataState:

randomData,

presorted,

reverseSorted

// Precondition: *A* is an array with the data,

// *start* is the index of the first element in the left division,

// *m* is the index of the first element in the right division,

// and *end* is the index of the last element in the right division.

// The left and right divisions must each be sorted.

// Postcondition: The elements in *A* from *start* to *end*, inclusive, are in

// increasing sorted order.

merge( A, start, m, end ):

size = end – start + 1

// Loop Invariant: at each execution of the guard,

// *L*[0…(*k*-*start*-1)] = *A*[*start*…(*k*-1)]

for k = start to k < m:

L[k - start] = A[k]

L[m – start] = ∞

i = j = 0 // we are indexing from zero

// Loop Invariant: at each execution of the guard, *A*[*start*…(*start*+*k*-1)]

// is sorted

for k = 0 to k < size:

if L[i] < A[j + m] || j > end – m:

A[k + start] = L[i]

i++

else:

A[k + start] = A[j + m]

j++

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// *min* is the index of the first element in the sort;

// *max* is the index of the last element in the sort.

// *A* must be at least of size *max*+1.

// Postcondition: The elements of *A* from position *min* to position *max*, inclusive, are

// in sorted order.

mergesort( A, min, max ):

if max != min:

center = floor( (max + min + 1) / 2 )

mergesort( A, min, center – 1 )

mergesort( A, center, max )

merge( A, min, center, max )

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// *min* is the index of the first element in the sort;

// *max* is the index of the last element in the sort.

// *A* must be at least of size *max*+1.

// Postcondition: The elements of *A* from position *min* to position *max*, inclusive, are

// in sorted order.

insertionsort( A, min, max ):

// Loop Invariant: at each execution of the guard, *A*[*min*...(*i*-1)] is sorted

for i = min + 1 to i <= max:

j = i

// Loop Invariant: at each execution of the guard,

// *A*[*j*] is the minimum element of *A*[*j*...*i*]

while j > min && A[j] < A[j-1]:

swap( A[j-1], A[j] )

j--

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// *min* is the index of the first element in the sort;

// *max* is the index of the last element in the sort.

// *A* must be at least of size *max*+1.

// Postcondition: The elements of *A* from position *min* to position *max*, inclusive, are

// in sorted order.

bubblesort( A, min, max ):

// Loop Invariant: at each execution of the guard, the last *i* elements in *A*

// are the largest *i* elements in sorted order

for i = 0 to i < max - min + 1:

// Loop Invariant: at each execution of the guard, the largest element

// of *A*[*min*...*j*] is *A*[*j*]

for j = min to j < max – i:

if A[j+1] < A[j]:

swap( A[j+1], A[j] )

// Precondition: *A* is an array with the data that is to be

// sorted from *min* to *max* inclusive.

// *min* is the index of the first element in the sort;

// *max* is the index of the last element in the sort.

// *A* must be at least of size *max*+1.

// Postcondition: The elements of *A* from position *min* to position *max*, inclusive, are

// in sorted order.

timsort( A, min, max ):

if max - min < TIM\_SORT\_THRESHOLD:

insertionsort( A, min, max )

else:

center = (max + min + 1) / 2

timsort( A, min, center - 1 )

timsort( A, center, max )

merge( A, min, center, max )

// Precondition: The size of *A* is at least *size*.

// Postcondition: *A* is filled with integers that are either nondecreasing,

// nonincreasing, or random, depending on the value of *state*.

generateData( A, size, state = randomData ):

// Loop Invariant: at each execution of the guard, *A*[0…(*k*-1)] contains

// randomized data

for k = 0 to k < size:

A[k] = rand() % MAX\_DATUM\_VALUE

if state == presorted:

mergesort( A, 0, size - 1 )

else if state == reverseSorted:

mergesort( A, 0, size - 1 )

// Loop Invariant: at each execution of the guard, *A*[0…*j*]

// and *A*[(*size*-1-*j*)…(*size*-1)] are reverse sorted

for j = 0 to j < size / 2:

swap( A[j], A[size - j - 1] )

// Precondition: *data* must be at least size *n*.

// Postcondition: If *data* satisfies the specified *direction*,

// then true is returned. Otherwise false is returned.

isSorted( data, n, direction = presorted )

// Loop Invariant: at each execution of the guard, *data*[0…(*k*-1)] is

// sorted in increasing order if *direction* = *presorted*, or sorted in

// decreasing order if *direction* = *reverseSorted*.

for k = 1 to k < n:

if ( direction == presorted && data[k] < data[k-1] )

|| ( direction == reverseSorted && data[k-1] < data[k] ):

return false

return true

// Precondition: *array1* and *array2* must be of at least size *n*.

// Postcondition: If *array1*[0…*n*-1] = *array2*[0…*n*-1] then true is returned.

// Otherwise false is returned.

isEqual( array1, array2, n )

eq = true

// Loop Invariant: at each execution of the guard,

// *eq* = (*array1*[0…*k*-1] = *array2*[0…*k*-1])

// and the loop terminates if *eq* is false.

k = 0

while eq && k < n:

if array1[k] != array2[k]:

eq = false

k++

return eq

// Precondition: none.

// Postcondition: Timing data for Timsort, Merge sort,

// Insertion sort, and Bubble sort is outputted along with the exponent on sqrt(2)

// and the data size. This is done for each of 26 sizes of data for

// each of 3 data states.

main():

// Loop Invariant: at each execution of the guard, the timing data

// for all previous values of *ds* are outputted.

foreach ds in dataState:

exponent = 1

// Loop Invariant: at each comparison of the guard,

// the approximate number of CPU clock cycles it takes to run each of

// the sorting algorithms on sqrt(2) ^ (1…*exponent*–1) items

// has been outputted

while sqrt(2) ^ exponent <= MAX\_DATA\_SIZE:

n = floor( (sqrt(2) ^ exponent) + 0.5 )

// generate data and check to make sure it is generated correctly

if ds == randomData:

generateData( original\_data, n, randomData )

if n > RANDOMNESS\_THRESHOLD

&& ( isSorted( original\_data, n )

|| isSorted( original\_data, n, reversesorted )

) ):

Error()

else if dsIndex == 1:

generateData( original\_data, n, presorted )

if( !isSorted( original\_data, n ) ):

Error()

else:

generateData( original\_data, n, reversesorted )

if( !isSorted( original\_data, n, reversesorted ) ):

Error()

data = original\_data

t = currentTime()

timsort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted ):

Error()

output( duration )

data = original\_data

t = currentTime()

mergesort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted ):

Error()

output( duration )

data = original\_data

t = currentTime()

insertionsort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted ):

Error()

output( duration )

data = original\_data

t = currentTime()

bubblesort( data, 0, n-1 )

duration = currentTime() – t

if !isSorted( data, n, presorted ):

Error()

output( duration )

output( exponent )

output( n )

exponent++

The C++ program outputted the data through STDOUT. The program was compiled into the Unix executable file a.out and the Unix command ./a.out > CPUcounts.txt was used to collect the data into a text file. This was done on a MacBook Pro with a single core processor, which may have helped the accuracy of the CPU cycle counts. There are 2 billion CPU clock cycles in a second on this computer. The data was imported into Microsoft Excel. First, the data was plotted on a linear scale. The linearly scaled graph is included in this report to give the viewer a feel for what the data really looks like. Then the data was plotted on a logarithmic scale. The logarithmically scaled graph allows for deep analysis of the data. After the data was put on the graph, lines representing the asymptotic complexity classes were added. Timsort is represented by the color green, Merge sort by blue, Insertion sort by orange, and Bubble sort by gray. Test cases of randomly sorted data are represented by circles, presorted by squares, and reverse sorted by triangles.

**Problems encountered:**

The biggest problem was noise in the data. We don’t know what is causing this noise. Our first attempt to solve this problem was to use CPU clock cycle counts instead of microsecond timer readings. This gave us a lot more precision, but there was still some noise. Our second attempt to remove the noise in the data was to account for the possibility that the launching of the program put some tasks on the CPU’s task list which slowed down the execution of the program. If this were the case, then the tasks would go away given sufficient time. So we put some sleep statements in the code. But the sleep statements did not affect the data in the output.

**Tests of the code:**

Throughout the code we placed checks using the isSorted() and isEqual() functions. We used the isSorted() function to see if the data was sorted or reversely sorted when it was supposed to be sorted or reversely sorted. isEqual() was used to make sure two arrays had equivalent values when they were supposed to have equivalent values. If a test failed, then the program would throw an Error object with a message, which would be caught and the message outputted to STDOUT. In order to make sure that our tests were correct we put tried putting test errors in the code. When the program threw what it was supposed to throw we removed the test errors.

The normal program output was six-column space-separated data, with the CPU cycle count data from Timsort, Mergesort, Insertionsort, and Bubblesort in that order, followed by the exponent of sqrt(2) that produced the data size, and the data size itself. These columns were rows long. There were 26 rows for random data, 26 for presorted data, and 26 for reverse sorted data. The coherence of the data makes us sure that the program is correct.

**Conclusion:**

For data sizes larger than about 100 elements, Merge sort and Timsort are faster than both Insertion sort and Bubble sort. 100 elements can be thought of as an n0 for asymptotic complexity notation. At the data size 10,000, our empirical analysis shows Merge sort and Timsort around 30 times faster than Insertion sort and Bubble sort. At 10,000, Bubble sort takes about half a second. For data sizes smaller than about 100 elements, Insertion sort, Bubble sort, and Timsort are all much faster than Merge sort. Insertion sort is faster than Bubble sort for all data sizes of random data, but not by much. Timsort is close to Insertion sort for small data sizes and close to Merge sort for large data sizes.

Presorted data appears to be the best case for all four algorithms in the experiment. Insertion sort is extremely fast on presorted data. On small data sizes, Timsort takes advantage of its Insertion sort parts and is also extremely fast on presorted data. On large data sizes, Timsort does not handle presorted data like Insertion sort although it still handles presorted data better than Merge sort. Merge sort performs better on reverse sorted data than random data which shows that reverse sorted data is not the worst case for Merge sort. For data sizes larger than about 1,000, Timsort also performs better on reverse sorted data than random data. However, for Insertion sort and Bubble sort reverse sorted data is worse than random data. The logarithmically scaled graph makes the reverse sorted data case look nearly equivalent for Insertion sort and Bubble sort, but the linearly scaled graph shows that Insertion sort’s case is a little faster than Bubble sort’s. Finally, there is no noticeable effect of powers of 2 on Merge sort or its derivate Timsort.

Comparing the data to the complexity classes we see that the data is can be modeled by the complexity class expression for large data sizes. For small data sizes, the algorithms tend to take longer than the expression of their complexity classes. This makes sense because the complexity class expression is the term of the exact run time expression with the largest asymptotic growth, with scalars removed. The exact number of operations of an algorithm usually has some lower-order terms which make a big difference for small data sizes but make insignificant difference when the data size is large.

The time complexity of each algorithm came out in the empirical analysis. In some cases, one algorithm was 30 times faster than another, which shows that there is more to algorithm selection than satisfying the postcondition. We may want to extrapolate the data collected in this experiment to larger sizes than 10,000. By extrapolating along the n2 curve we can calculate that with 100,000,000 randomly generated data, Bubble sort would take a number of months to finish. But by extrapolating along n\*lg(n), Merge sort would handle 100,000,000 randomly generated data in a number of minutes. Algorithm selection can completely change the functionality of software.