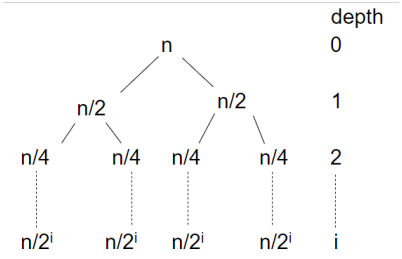
Project Deliverable 6

Alex Abrahamson - Mitchell Baker - Brock Ellefson - Yue Hou - Seth Severa

4 December 2017

Introspective sort is a sorting algorithm that combines the quick average time of Quicksort with the stable worst case performance of Heapsort developed by Dr. David Musser in 1997. Sorting algorithms often need to find a compromise between fast execution in the common case and a reliable performance when it comes to problematic inputs. We wanted to take this algorithm and compare its runtime to the runtime of the two algorithms it was constructed from, Quicksort and Heapsort.

A rudimentary overview of the methods these algorithms use makes clear the advantages of both. Quicksort is in most cases the fastest sorting algorithm, utilizing recursion and running with a recurrence relation of $T(n) = 2T(n/2) + O(n)$. It's average runtime is $O(nlogn)$, however in the worst case of Quicksort our subproblems do not increase efficiently. As seen below, the number of subproblems approaches n, giving us $O(n^2)$ time.



Heapsort is an in-place comparison algorithm that runs in $nlogn$ no matter the input it runs on. Since it creates a heap tree with a logarithmic depth, we can guarantee that searching and replacing happen quickly, even on already sorted inputs.

In Introsort, we combine these two implementations. When we limit our depth to $2logn$, we know that we will have at most 2^{2logn} subproblems, each being solved by the Heapsort in $nlogn$ time. Since we are at the level $2logn$, Heapsort will actually run in $(n/2^i)logn(2^i)$ time. This means that we have n^2 subproblems

solved in $(1/n)log(1/n)$ time which simplifies to $nlog(1/n)$ which is $O(nlogn)$. So Introsorts runtime will always be $O(nlogn)$, regardless of the input.

With the problem stated, we provide pseudocode for further clarity on the implementation of the algorithm Introsort.

Algorithm 1 INTROSORT

```
1:  Procedure Sort( A)
2:  maxdepth = 2⌊log‖A‖⌋
3:  Introsort( A, maxdepth)
4:
5:  Procedure Introsort( A, maxdepth)
6:  in: A, maxdepth
7:  n ← |A|
8:
9:  if n ≤ 1 then
10:     return
11: else if maxdepth = 0 then
12:     HEAPSORT(A)
13: else
14:     p ← PARTITION(A)
15:     INTROSORT(A[0:p], maxdepth - 1)
16:     INTROSORT(A[p + 1:n], maxdepth −1)
17: end if
```

Loop invariant

In the recursive procedure of INTROSORT, the loop invariant L is

L = { A is the concatenation of the return values of INTROSORT(A[0:p], maxdepth $-$ 1) and INTROSORT(A[0:p], maxdepth $-$ 1, up until the maxdepth value reaches 0. }

Following this, in the HEAPSORT procedure the loop invariant L becomes

L = { At the start of each iteration of the loop, the subarray A$[1, ..., i]$ is a max heap containing the i^{th} smallest elements of A, and the subarray A$[i + 1, ..., n]$ contains the $n - i$ largest elements of A$[1, ..., n]$ sorted. } (CLRS, 160).

Introsort and the '+1 element'

For our additional elements regarding Introspective Sort, we chose to provide clear and intuitive representations of the algorithm's time complexity, space complexity, and running method. Time complexity analysis is provided above via recurrence tree. Discussion and presentation on the performance of the algorithm is provided by a short video, the contents summarized in part above, and further here.

Our video is a walk-through of a few common sorting algorithms. Bubblesort, Heapsort, and Quicksort are shown and their methods of operating explained. Bubblesort is highly iterative, sweeping the entire array multiple times, which Heapsort aims to create a better structure from which to repopulate the array in order. Quicksort finishes these algorithms off by being a recursive divide-and-conquer algorithm but with a drawback- there are times when it performs as bad as our first algorithm.

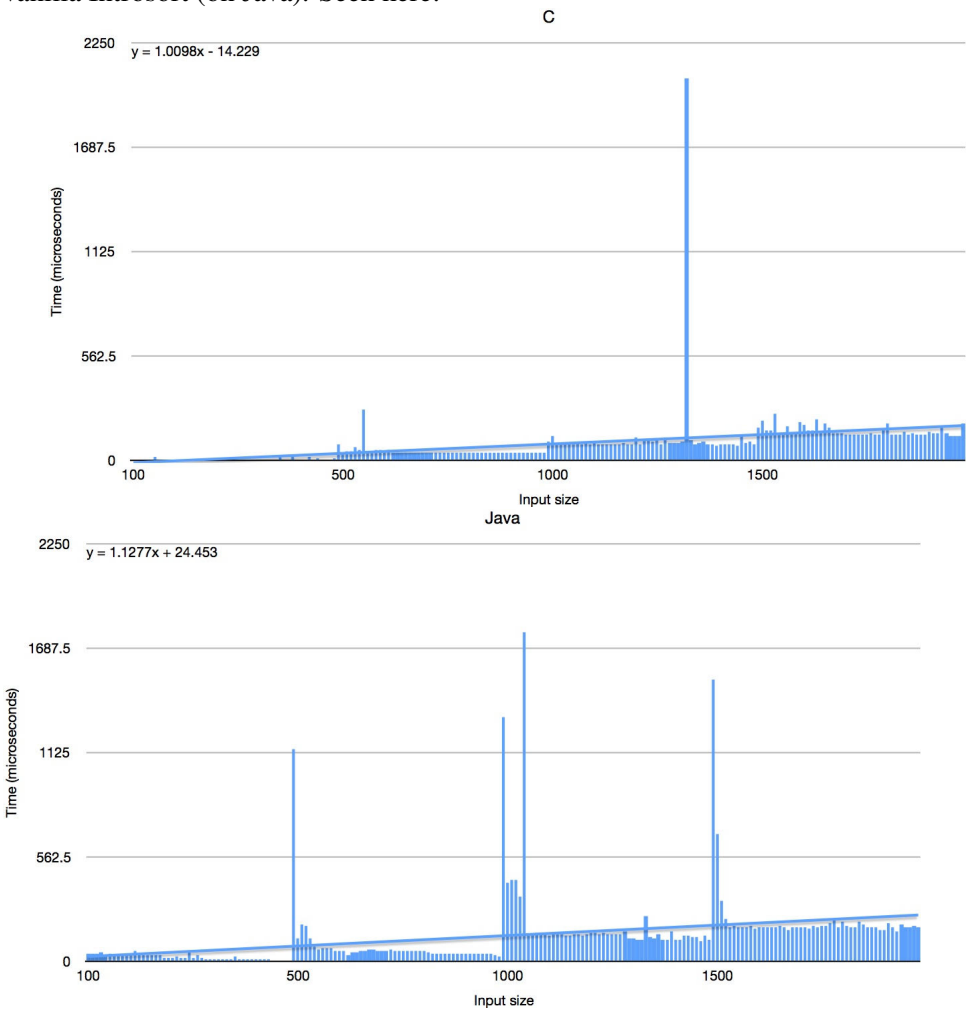
Introsort is introduced as a superior alternative, and discussed while showing comparisons of the algorithms in action; visualized and highly intuitive. We cover the benefits of the maximum depth branching, while retaining the quick average case of Quicksort. Narration and tactile approach make our video easy to comprehend and pleasing to engage with. Using a python library called py-game, we then implemented a visual representation of how these sorting algorithms actually sort a given array.

We then implemented a better version of Introsort, using references to speed up the sorting process even further. We also tested Introsort against other sorting alorithms. We testing runtime and counted operations of each algorithm.

Table 1: Time to sort given size n

n	IntroSort	QuickSort
1,000	.0018	.0024
10,000	.021	.023

Our implementation of Introsort (on C++) on average seemed to do about 10 percent better than just vanilla Introsort (on Java). Seen here:



This approach has led us to a better and grander understanding of the world of sorting algorithms, and the realization at how widely used and appreciated introspective sort is, though we way not have known it. The github repository that contains these files can be found here,

or by using this link https://github.com/brockellefson/IntrospectiveSortResearch