Setup

I setup the tests so that they would quicksort 100 times for each test. There were two separate types of tests setups, one that was from `[0,20]` and incremented by one each iteration, and a different setup that went from `[0, 2^16]` and incremented the using `f(x) = 2^x` where x went up by one until `x = 16 (it doubled).` Each of these tests will be compared/sorted using several different methods including pure quicksort and a quicksort/insertion sort hybrid. Unless specified, each result is computed using compare cost and not time cost. Each section will be a test case. Some sections may depend on work done in a previous section.

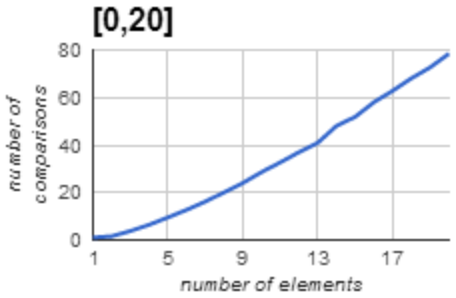
Sections

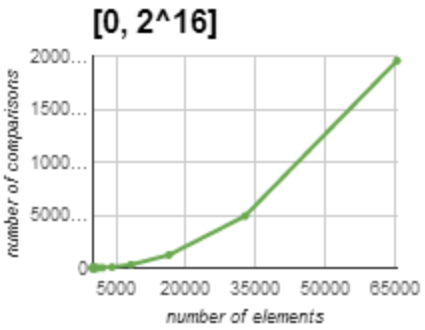
1. In this test we are to compare how many element comparisons quicksort does for an input array of size *n.* The partition code has been implemented to specifications. The actual sort has been implemented similar to the following pseudocode:

```
sort (data, start, end)
      part = partition(data, start, end)

      if end - start < 2 or start > end
            return array;

      if start < part - 1
            return quickSort(data, start, part - 1)

      else if part < end
            return quickSort(array, part + 1, end);

      return array;
```

This is an *in-place* sort. No extra arrays are created.

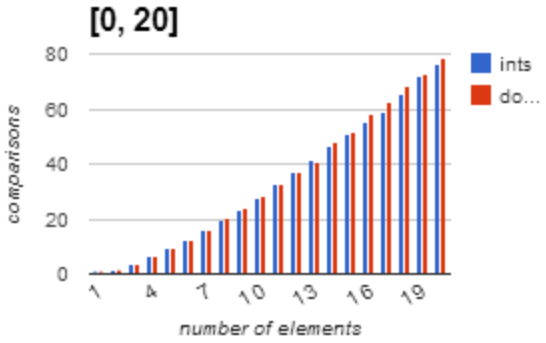
# of comparisons goes to ~200,000,000

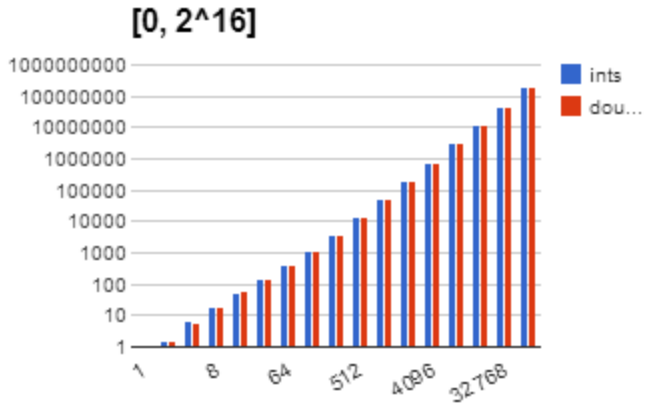




These are the number of comparisons it takes per sort of an array of size *n*. This particular sort costs about $\Omega(n^2)$. That is the upper bound of the function, meaning that this is the most the function will ever take in terms of comparisons.

2. In the previous test, the data was of type double with a value of [0, 1). In this test we will change it to be of value [0, 10). Integer generated, but still stored in a double. This is to keep consistency across the code.



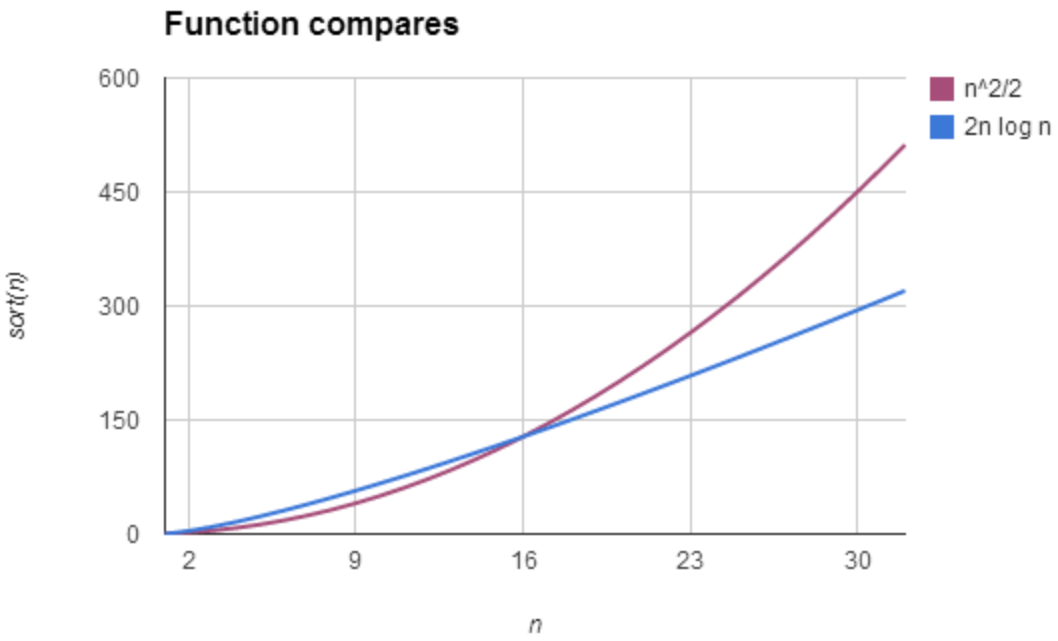


When sorting data with sizes ranging [0, 20], the lowest number of comparisons usually came from the integers. About 7 out of 10 of the best cases belonged to the integers. However, when getting higher and higher it evened out to be almost 50/50. This may just be my test, however I would guess that it would make no difference, in terms of comparison costs. The way the test is currently set up, it shouldn't make a difference whether it is an integer or not. However, if we were to change how they were compared, we could make the function get closer to a `O(n log n)`.

3. In this test, I changed the partition function to compare for data at right to be just greater than the pivot point. When making this change, the compare counts decreased *immensely*. Almost 10000%. This is because now, it is not comparing the right side of the partition including the pivot. It is now just comparing the right partition. This brought us down closer to the lower bound of the quick sort algorithm. $O(n \log n)$. The values of the function are now between $O(n \log n)$ and $\Omega(n^2)$.

4. It is known that quicksort has a lower bound of about $O(n \log n)$. However, this is just the proportional asymptote. The actual formula is approximately $O(2n \log n)$ in its average case. It is also known that the lower bound of an insertion sort is around $O(n^2)$. But when computed out it comes to about $O(n^2 / 2)$. Again this is just the proportional asymptote, eventually the division of 2 will have little effect on the complexity of the function, as the amount of elements rises.

When we graph out both the approximate functions for both of these sorts, we get a graph like this:



This is nifty because, just before 16 we get that $n^2 / 2$ is smaller than $2n \log n$.
So when there are about 16 elements, insertion sort should have less than or equal the amount of comparisons that quicksort has. If implemented correctly, a quick sort/ insertion sort hybrid could significantly lower the cost of the overall sort.

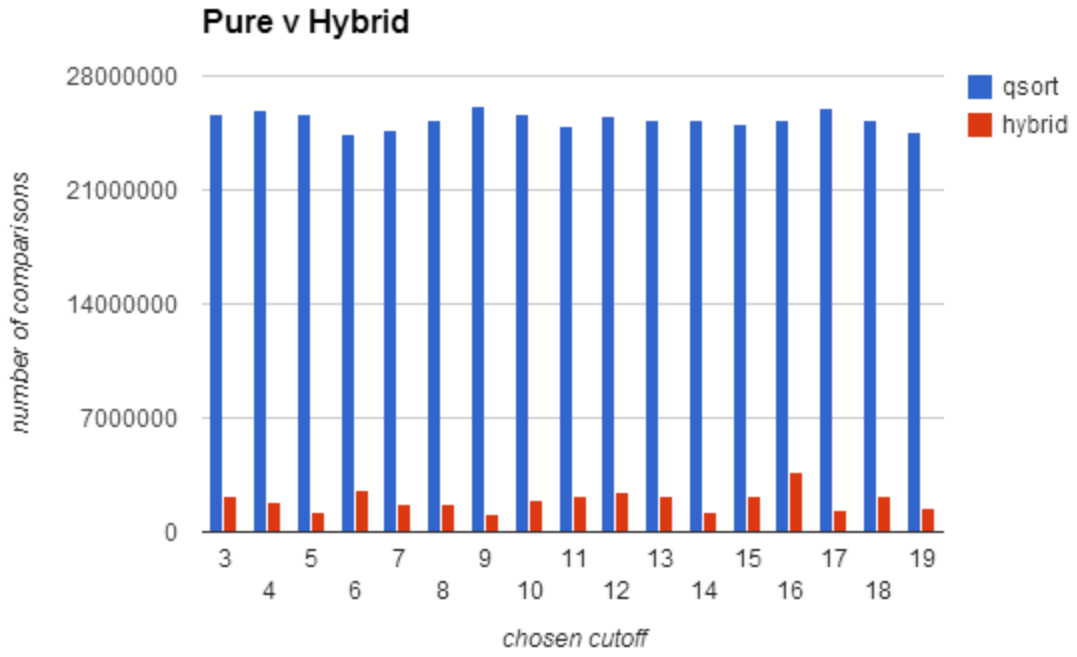
5. In this test, a hybrid quicksort / insertion sort is to be implemented. The implementation goes as follows:

```
if subarray size < 2
   return

else if subarray size < cutoff
   use insertion sort on this subarray

else
   do quicksort partition and sorting on this subarray
```

To calculate the best possible cutoff point for data of size 1000 to size 1000000, I used the following pseudocode:

```
highestMargin = 2 ^ 64;
bestCutoff = -1;
from 3 to 20 increment 1 // We choose low numbers
      fill = generateData(size)
      // returns the number of compares
      comparesWithout = quickSort(fill)

      fill = generateData(size);
      comparesWith = quickSortAndInsert(fill);

      if comparesWithout > comparesWith
            margin = comparesWithout - comparesWith
            if margin > highestMargin
                  bestCutoff = cutoff;
                  highestMargin = margin;
```

I attempted to calculate the best value for the cutoff. The cutoff is the best possible place to stop using quick sort and finish off with insertion sort.



The average cutoff given ranged from 7 - 14. The means of the averages came to about 11. After running the tests several times, the best cutoff for 1000000 elements was 9. It had only 1065242 comparisons when sorting in a hybrid sort compared to 2.6143963E7 comparisons in a pure quicksort.

In the previous section, it was stated that anything after 16 would reduce the amount of comparisons. This is because those formulas where just an approximation. There are some missing steps in the cycle. But overall, it was a very close guess.

6. When making a *growable* array, there is a specific cost associated with appending elements to an array that is full. Normally when inserting/appending, the cost is about O(1). But when the array is filled, there is a need to expand. The only way to do that is create a new array of a bigger size, then fill the new array with the old filled array. This has a hefty cost associated with this transaction, about O(n + 1). Because it needs to now, take all elements from the previous array *n*, and move them to the new array (Threw in a +1 because that is the cost of appending/inserting an element into the array).

Normally when there is a need for a new array, the size of the fresh array is doubled the previous one. So if the previous array had only ten slots, the new one would be doubled to twenty slots. But for this experiment, there will be three different methods to expanding the array sizes. Instead of doubling the sizes the new sizes are going to be:

- Tripled
- Increased by 10%
- Size of previous + 1000

The sizes of these arrays will be identified by the following formulas (in order of appearance):

- len = f(3n)
- len = f(n + n*.1) *or* f(n*1.1)
- len = f(n + 1000)

These are the sizes of the new array when it comes time to grow. Surprisingly enough, they all have the same big-o notation. O(n). Meaning that if the size of elements is big enough, it won't matter how you grow the array. However, if we are talking about 1 - 100 elements or so, the best method to choose is the smallest allocation, which is increase by 10%. However, with that in mind, it may not be the best if its grown frequently.